

OS/390



# C/C++ Language Reference



OS/390



# C/C++ Language Reference

**Note!**

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

**Fourth Edition, September 1998**

This edition applies to Version 2 Release 6 of OS/390 C/C++ (5647-A01) and to all subsequent releases and modifications until otherwise indicated in new editions or other updated documentation. Make sure that you use the correct edition for the level of the program listed above. Also, ensure that you apply all necessary PTFs for the program.

Technical changes in the text since the last release of this book are indicated by a vertical line (|) to the left of the change.

Order publications through your IBM representative or the IBM branch office serving your location. Publications are not stocked at the address below. Note that the OS/390 C/C++ publications are available through the OS/390 Library page at: <http://www.s390.ibm.com/os390/bkserv>.

IBM welcomes your comments. You can send your comments electronically to the network ID listed below. Be sure to include your entire network address if you wish a reply.

Internet: [torrcf@ca.ibm.com](mailto:torrcf@ca.ibm.com)  
IBMLink: [toribm\(torrcf\)](#)  
IBM/PROFS: [torolab4\(torrcf\)](#)  
IBMMAIL: [ibmmail\(caibmwt9\)](#)

To send your comments by facsimile (attention: RCF coordinator) use the following FAX numbers:

United States and Canada: 416-448-6161  
Other Countries: (+1)-416-448-6161

Alternatively, you can use the Reader's Comment Form that is provided at the back of this publication, or mail your comments directly to:

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

If you send comments, include the title and order number of this book, and the page number or topic related to your comment. When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

© Copyright International Business Machines Corporation 1996, 1998. 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> . . . . .	<b>ix</b>
Standards . . . . .	ix
Trademarks . . . . .	x

---

## Part 1. Introduction . . . . . 1

### Chapter 1. About This Book . . . . . 3

Who Should Use This Book . . . . .	3
A Note about Examples. . . . .	3
IBM OS/390 C/C++ and Related Publications . . . . .	4
Hardcopy Books . . . . .	8
Softcopy Books . . . . .	9
Softcopy Examples . . . . .	9
OS/390 C/C++ on the World Wide Web . . . . .	10
C/C++ News... . . . .	10
How to Read the Syntax Diagrams . . . . .	10

### Chapter 2. About IBM OS/390 C/C++ 13

Changes for Version 2 Release 6 . . . . .	13
The C/C++ Compilers . . . . .	14
The C Language . . . . .	14
The C++ Language . . . . .	14
Common Features of the OS/390 C and C++ Compilers . . . . .	15
OS/390 C Compiler Specific Features . . . . .	16
Features That Are Specific to the OS/390 C++ Compiler . . . . .	16
Utilities . . . . .	17
Class Libraries. . . . .	17
Class Library Source. . . . .	18
The Debug Tool . . . . .	18
OS/390 Language Environment . . . . .	19
The Program Management Binder . . . . .	19
OS/390 UNIX System Services (OS/390 UNIX)	20
OS/390 C/C++ Applications with OS/390 UNIX	
C/C++ Functions. . . . .	21
Input and Output. . . . .	22
I/O Interfaces . . . . .	22
File Types . . . . .	23
Additional I/O Features . . . . .	23
The System Programming C Facility . . . . .	24
Interaction with Other IBM Products . . . . .	24
Additional Features of OS/390 C/C++ . . . . .	25

---

## Part 2. The C and C++ Languages 29

### Chapter 3. Introduction to C and C++ 31

Overview of the C Language . . . . .	31
C Source Programs . . . . .	32
CBC3RAAA . . . . .	33
C Source Files . . . . .	33
CBC3RAAB - Source File 1 . . . . .	34
CBC3RMAX - Source file 2 . . . . .	34
Program Execution . . . . .	35

Scope in C . . . . .	35
Block Scope. . . . .	35
Function Scope . . . . .	36
File Scope . . . . .	36
Function Prototype Scope . . . . .	36
Example of Scope in C . . . . .	36
Related Information . . . . .	37
Program Linkage . . . . .	37
Internal Linkage . . . . .	38
External Linkage . . . . .	38
No Linkage. . . . .	39
Storage Duration . . . . .	39
Name Spaces . . . . .	39
Related Information . . . . .	40
Command-Line Arguments . . . . .	40
Under OS/390 Batch. . . . .	41
Under IMS . . . . .	41
Under CICS . . . . .	41
Under TSO Command . . . . .	41
Under TSO Call . . . . .	41
Under OS/390 UNIX Shell. . . . .	41
Related Information . . . . .	42
Overview of the C++ Language . . . . .	42
C++ Support for Object-Oriented Programming	42
Data Abstraction . . . . .	42
Encapsulation . . . . .	43
Inheritance . . . . .	43
Dynamic Binding and Polymorphism . . . . .	44
Other Features of C++ . . . . .	44
C++ Programs . . . . .	44
CBC3X02D . . . . .	46
Scope in C++ . . . . .	46
Local Scope. . . . .	46
Function Scope . . . . .	47
File Scope . . . . .	47
Class Scope. . . . .	47
Simple C++ Input and Output . . . . .	47
CBC3X02F . . . . .	48
Output (cout, cerr, and clog) . . . . .	48
Input (cin) . . . . .	49
Linkage Specifications — Linking to non-C++	
Programs . . . . .	50
CBC3X02J . . . . .	50

### Chapter 4. Lexical Elements of C and C++ 51

Tokens . . . . .	51
Source Program Character Set . . . . .	51
Trigraph Sequences . . . . .	53
Digraph Sequences . . . . .	53
Additional Keywords . . . . .	54
Comments . . . . .	54
C++ Comments . . . . .	56
Identifiers . . . . .	56
Special Characters in Identifiers . . . . .	57

Case Sensitivity in Identifiers . . . . .	57
Significant Characters in Identifiers . . . . .	57
Keywords . . . . .	57
OS/390 C/C++ External Name Mapping . . . . .	58
OS/390 Long Name Support . . . . .	59
Constants . . . . .	60
Integer Constants . . . . .	60
Floating-Point Constants . . . . .	62
Fixed-Point Decimal Constants (C Only) . . . . .	63
Character Constants . . . . .	64
String Literals . . . . .	65
Escape Sequences . . . . .	67
<b>Chapter 5. Declarations . . . . .</b>	<b>69</b>
Declarations Overview . . . . .	69
Block Scope Data Declarations . . . . .	70
Initialization . . . . .	71
Storage . . . . .	71
Related Information . . . . .	71
File Scope Data Declarations . . . . .	71
Initialization . . . . .	72
Storage . . . . .	72
Related Information . . . . .	72
Objects . . . . .	72
Storage Class Specifiers . . . . .	73
auto Storage Class Specifier . . . . .	73
extern Storage Class Specifier . . . . .	75
register Storage Class Specifier . . . . .	81
static Storage Class Specifier . . . . .	82
typedef . . . . .	84
Examples of typedef Declarations . . . . .	84
Related Information . . . . .	85
Type Specifiers . . . . .	85
Characters . . . . .	86
Floating-Point Variables . . . . .	87
Fixed-Point Decimal Data Types (C Only) . . . . .	88
Integer Variables . . . . .	89
Enumerations . . . . .	90
Pointers . . . . .	94
void Type . . . . .	99
Arrays . . . . .	100
Structures . . . . .	106
Unions . . . . .	113
Incomplete Types . . . . .	119
Declarators . . . . .	119
volatile and const Qualifiers . . . . .	120
__packed Qualifier (C Only) . . . . .	122
__cdecl Keyword (C++ Only) . . . . .	123
__export Keyword . . . . .	125
Example Declarators . . . . .	126
Initializers . . . . .	127
Related Information . . . . .	128
C/C++ Data Mapping . . . . .	129
C++ Function Specifiers . . . . .	129
C++ References . . . . .	129
Initializing References . . . . .	130
Related Information . . . . .	130

## Chapter 6. Expressions and Operators 133

Operator Precedence and Associativity . . . . .	133
Examples of Expressions and Precedence . . . . .	135

Operands . . . . .	135
lvalues . . . . .	136
Examples of lvalues . . . . .	136
Related Information . . . . .	136
Primary Expressions . . . . .	136
C++ Scope Resolution Operator (::) . . . . .	137
Parenthesized Expressions ( ) . . . . .	137
Constant Expressions . . . . .	138
Function Calls ( ) . . . . .	139
Array Subscript [ ] (Array Element Specification) . . . . .	140
Dot Operator (.) . . . . .	141
Arrow Operator (->) . . . . .	141
Unary Expressions . . . . .	142
Increment (++). . . . .	142
Decrement (--). . . . .	143
Unary Plus (+). . . . .	143
Unary Minus (-). . . . .	143
Logical Negation (!). . . . .	144
Bitwise Negation (~). . . . .	144
Address (&). . . . .	144
Indirection (*). . . . .	145
Cast Expressions . . . . .	145
sizeof (Size of an Object) . . . . .	146
digitsof and precisionof (C Only) . . . . .	147
C++ new Operator . . . . .	147
C++ delete Operator . . . . .	151
C++ throw Expressions . . . . .	152
Binary Expressions . . . . .	152
Multiplication (*). . . . .	152
Division (/). . . . .	153
Remainder (%). . . . .	153
Addition (+). . . . .	153
Subtraction (-). . . . .	154
Bitwise Left and Right Shift (<< >>) . . . . .	154
Relational (< > <= >=) . . . . .	155
Equality (== !=) . . . . .	156
Bitwise AND (&). . . . .	157
Bitwise Exclusive OR (^) . . . . .	157
Bitwise Inclusive OR ( ) . . . . .	158
Logical AND (&&). . . . .	158
Logical OR (  ) . . . . .	159
C++ Pointer-to-Member Operators (* ->*) . . . . .	160
Conditional Expressions . . . . .	160
Type of Conditional C Expressions . . . . .	161
Type of Conditional C++ Expressions . . . . .	161
Examples of Conditional Expressions . . . . .	161
Assignment Expressions . . . . .	162
Simple Assignment (=) . . . . .	162
Compound Assignment . . . . .	164
Comma Expression (,) . . . . .	165

## Chapter 7. Implicit Type Conversions 167

Integral Promotions . . . . .	167
Standard Type Conversions . . . . .	167
Signed-Integer Conversions . . . . .	168
Unsigned-Integer Conversions . . . . .	168
Floating-Point Conversions . . . . .	168
Pointer Conversions . . . . .	168
Reference Conversions . . . . .	169
Pointer-to-Member Conversions . . . . .	169

Function Argument Conversions. . . . .	170
Other Conversions . . . . .	170
Arithmetic Conversions. . . . .	170

## Chapter 8. Functions . . . . . 173

Functions Overview . . . . .	173
C++ Enhancements to C Functions . . . . .	173
Function Declarations . . . . .	174
C Function Declarations . . . . .	174
C++ Function Declarations. . . . .	175
Examples of Function Declarations . . . . .	176
Function Definitions . . . . .	178
Related Information . . . . .	184
The main() Function . . . . .	184
Arguments to main . . . . .	184
Example of Arguments to main . . . . .	185
Calling Functions and Passing Arguments . . . . .	185
Passing Arguments in C++ . . . . .	187
Examples of Calling Functions . . . . .	187
Passing Arguments by Reference . . . . .	188
Default Arguments in C++ Functions . . . . .	190
CBC3X06B . . . . .	190
Restrictions on Default Arguments . . . . .	191
Evaluating Default Arguments . . . . .	191
Function Return Values. . . . .	192
Using References as Return Types . . . . .	193
Pointers to Functions . . . . .	193
C++ Inline Functions . . . . .	195

## Chapter 9. Statements . . . . . 197

Labels . . . . .	197
Examples . . . . .	197
Related Information . . . . .	198
Block . . . . .	198
Initialization within Block Statements . . . . .	198
Example. . . . .	199
Related Information . . . . .	199
break . . . . .	200
Restrictions. . . . .	200
Examples . . . . .	200
Related Information . . . . .	201
continue . . . . .	202
Restrictions. . . . .	202
Examples . . . . .	202
Related Information . . . . .	203
do . . . . .	203
Example. . . . .	204
Related Information . . . . .	204
Expression . . . . .	205
Examples . . . . .	205
Resolving Ambiguous Statements in C++ . . . . .	205
for . . . . .	206
Examples . . . . .	207
Related Information . . . . .	208
goto . . . . .	208
Example. . . . .	209
if . . . . .	209
Examples . . . . .	210
null . . . . .	210
Example. . . . .	211
return . . . . .	211

Value of a return Expression and Function Value . . . . .	211
Examples . . . . .	212
Related Information . . . . .	212
switch . . . . .	212
Restrictions . . . . .	214
Examples . . . . .	214
Related Information . . . . .	216
while . . . . .	216
Example . . . . .	217
Related Information . . . . .	217

## Chapter 10. Preprocessor Directives 219

Preprocessor Overview . . . . .	219
Preprocessor Directive Format . . . . .	220
Phases of Preprocessing. . . . .	220
Macro Definition and Expansion (#define) . . . . .	221
Object-Like Macros . . . . .	221
Function-Like Macros . . . . .	222
Scope of Macro Names (#undef) . . . . .	225
Examples of #undef Directives . . . . .	225
Single Number Sign Operator (#) . . . . .	225
Examples of the # Operator . . . . .	226
Related Information . . . . .	226
Macro Concatenation with the ## Operator . . . . .	226
Double Number Sign Operator (##). . . . .	227
Preprocessor Error Directive (#error) . . . . .	228
Related Information . . . . .	228
File Inclusion (#include) . . . . .	228
Predefined Macro Names . . . . .	229
ANSI/ISO Standard Predefined Macro Names . . . . .	230
OS/390 C/C++ Predefined Macro Names . . . . .	231
Examples of Predefined Macros . . . . .	236
Conditional Compilation Directives. . . . .	237
#if, #elif . . . . .	238
#ifdef . . . . .	239
#ifndef . . . . .	239
#else . . . . .	240
#endif . . . . .	240
Examples of Conditional Compilation Directives . . . . .	240
Line Control (#line) . . . . .	241
Example of #line Directives . . . . .	242
Null Directive (#) . . . . .	242
Pragma Directives (#pragma) . . . . .	243
Restrictions on #pragma Directives . . . . .	245
IPA Considerations . . . . .	247
chars . . . . .	247
checkout. . . . .	248
comment . . . . .	248
convlit . . . . .	249
csect . . . . .	250
define (C++ Only) . . . . .	251
disjoint (C Only) . . . . .	251
environment (C Only) . . . . .	252
export . . . . .	253
filetag . . . . .	253
hdrstop . . . . .	254
implementation (C++ Only) . . . . .	255
info (C++ Only) . . . . .	255
inline (C Only) - also see noinline . . . . .	255

isolated_call . . . . .	257
langlvl . . . . .	259
linkage . . . . .	260
longname . . . . .	261
map . . . . .	262
margins . . . . .	264
noinline (C and C++) - also see inline . . . . .	265
options (C Only) . . . . .	266
pack . . . . .	267
page (C Only) . . . . .	270
pagesize (C Only). . . . .	270
priority (C++ Only) . . . . .	270
runopts . . . . .	271
sequence. . . . .	272
skip (C Only) . . . . .	273
strings . . . . .	273
subtitle (C Only) . . . . .	274
target (C Only) . . . . .	274
title (C Only) . . . . .	275
variable . . . . .	275
wsizeof . . . . .	275

## Part 3. C++ Language Elements . . . 279

### Chapter 11. C++ Classes . . . . . 281

C++ Classes Overview . . . . .	281
Classes and Structures . . . . .	281
Aggregate Classes . . . . .	282
Declaring Class Objects . . . . .	282
Class Names . . . . .	283
Using Class Objects . . . . .	284
Scope of Class Names . . . . .	286
CBC3X10E . . . . .	286
Incomplete Class Declarations . . . . .	287
Nested Classes. . . . .	287
Local Classes . . . . .	288
Local Type Names . . . . .	289

### Chapter 12. C++ Class Members and Friends . . . . . 291

Class Member Lists . . . . .	291
Data Members. . . . .	292
Class-Type Class Members. . . . .	292
Member Functions . . . . .	293
const and volatile Member Functions . . . . .	293
Virtual Member Functions . . . . .	294
Special Member Functions . . . . .	294
Inline Member Functions . . . . .	294
Member Function Templates . . . . .	295
Member Scope. . . . .	295
CBC3X11A . . . . .	295
Pointers to Members. . . . .	297
CBC3X11B . . . . .	297
The this Pointer . . . . .	298
CBC3X11C . . . . .	298
CBC3X11D . . . . .	299
Static Members . . . . .	300
Using the Class Access Operators with Static Members . . . . .	301
Static Data Members. . . . .	302

Static Member Functions . . . . .	303
Member Access . . . . .	304
Classes and Access Control . . . . .	304
Access Specifiers . . . . .	305
Friends . . . . .	306
CBC3X11I . . . . .	306
CBC3X11J . . . . .	307
Friend Scope . . . . .	308
Friend Access . . . . .	309

### Chapter 13. C++ Overloading . . . . . 311

Overloading Functions . . . . .	311
CBC3X12A . . . . .	311
Declaration Matching . . . . .	312
Restrictions on Overloaded Functions . . . . .	312
Argument Matching in Overloaded Functions . . . . .	312
Sequence of Argument Conversions . . . . .	313
Trivial Conversions . . . . .	314
Overloading Operators . . . . .	315
CBC3X12B . . . . .	315
General Rules for Overloading Operators. . . . .	316
Operands of Overloaded Operators. . . . .	316
Restrictions on Overloaded Operators . . . . .	317
Overloading Unary Operators . . . . .	317
Overloading Binary Operators . . . . .	318
Special Overloaded Operators . . . . .	319
Overloaded Assignment . . . . .	319
Overloaded Function Calls. . . . .	319
Overloaded Subscripting . . . . .	320
Overloaded Class Member Access . . . . .	320
Overloaded Increment and Decrement. . . . .	321
Overloaded new and delete . . . . .	322

### Chapter 14. Special C++ Member Functions . . . . . 325

Constructors and Destructors Overview . . . . .	325
Constructors . . . . .	326
Default Constructors. . . . .	326
Copy Constructors . . . . .	327
Construction Order of Class Objects . . . . .	327
Explicitly Constructing Objects . . . . .	328
Destructors . . . . .	328
Free Store . . . . .	330
Temporary Objects . . . . .	333
Related Information . . . . .	334
User-Defined Conversions . . . . .	334
Conversion by Constructor . . . . .	335
Conversion Functions . . . . .	335
Initialization by Constructor . . . . .	336
Explicit Initialization. . . . .	336
Initializing Base Classes and Members. . . . .	338
Construction Order of Derived Class Objects . . . . .	339
Copying Class Objects . . . . .	340
Copy Restrictions. . . . .	340
Copy by Assignment . . . . .	341
Copy by Initialization . . . . .	341

### Chapter 15. C++ Inheritance . . . . . 343

Inheritance Overview . . . . .	343
Multiple Inheritance . . . . .	344



The Inheritance Design Process . . . . .	345
Direct and Indirect Base Classes . . . . .	345
Polymorphism . . . . .	346
Derivation . . . . .	346
CBC3X14A . . . . .	347
CBC3X14B . . . . .	347
CBC3X14C . . . . .	348
Syntax of a Derived Class Declaration . . . . .	348
Inherited Member Access . . . . .	349
Protected Members . . . . .	350
Derivation Access of Base Classes . . . . .	350
Access Declarations . . . . .	351
Access Resolution . . . . .	353
Access Summary . . . . .	355
Multiple Inheritance . . . . .	356
Virtual Base Classes . . . . .	357
Multiple Access . . . . .	357
Accessible Base Classes . . . . .	358
Ambiguous Base Classes . . . . .	358
Virtual Functions . . . . .	359
Ambiguous Virtual Function Calls . . . . .	361
Virtual Function Access . . . . .	362
Abstract Classes . . . . .	363

## Chapter 16. C++ Templates . . . . . 365

Templates Overview . . . . .	365
CBC3X15A . . . . .	367
Structuring Your Program Using Templates . . . . .	367
File stack.h . . . . .	368
File stackdef.h . . . . .	368
Class Templates . . . . .	369
Class Template Declarations and Definitions . . . . .	370
Reference and Uniqueness . . . . .	371
Nontype Template Arguments . . . . .	371
Explicitly Defined Template Classes . . . . .	373
Function Templates . . . . .	373
Example of a Function Template . . . . .	373
Overloading Resolution for Template Functions . . . . .	374
Defining Template Functions . . . . .	375
Explicitly Defined Template Functions . . . . .	375
Function Template Declarations and Definitions . . . . .	376
Differences between Class and Function Templates . . . . .	377
CBC3X15B . . . . .	377
Member Function Templates . . . . .	377
Friends and Templates . . . . .	379
Static Data Members and Templates . . . . .	380

## Chapter 17. C++ Exception Handling 381

C++ Exception Handling Overview . . . . .	381
Formal and Informal Exception Handling . . . . .	382
Using Exception Handling . . . . .	382
Transferring Control . . . . .	384
CBC3X16A . . . . .	385
CBC3X16F . . . . .	386
Catching Exceptions . . . . .	387
Matching Exceptions Thrown and Exceptions Caught . . . . .	387
Order of Catching . . . . .	388
Nested Try Blocks . . . . .	388
Rethrowing an Exception . . . . .	389

Using a Conditional Expression in a Throw Expression . . . . .	390
Constructors and Destructors in Exception Handling . . . . .	391
CBC3X16D . . . . .	391
Exception Specifications . . . . .	393
Exception Specification Syntax . . . . .	393
Empty Exception Specifications . . . . .	394
Functions without an Exception Specification . . . . .	394
Other Exception Specifications . . . . .	394
Special Exception Handling Functions . . . . .	395
unexpected() . . . . .	395
terminate() . . . . .	395
set_unexpected() and set_terminate() . . . . .	395
Example of Using the Exception Handling Functions . . . . .	396

## Part 4. Appendixes . . . . . 399

### Appendix A. C and C++ Compatibility 401

C++ Constructs Not Found in ANSI/ISO C . . . . .	401
Constructs Found in Both C++ and ANSI/ISO C . . . . .	401
Character Array Initialization . . . . .	401
Character Constants . . . . .	402
Class and typedef Names . . . . .	402
Class and Scope Declarations . . . . .	402
const Object Initialization . . . . .	403
Definitions . . . . .	403
Definitions within Return or Argument Types . . . . .	403
Enumerator Type . . . . .	403
Enumeration Type . . . . .	403
Function Declarations . . . . .	403
Functions with an Empty Argument List . . . . .	404
Global Constant Linkage . . . . .	404
Jump Statements . . . . .	404
Keywords . . . . .	404
main() Recursion . . . . .	404
Names of Nested Classes . . . . .	404
Pointers to void . . . . .	405
Prototype Declarations . . . . .	405
Return without Declared Value . . . . .	405
__STDC__ Macro . . . . .	405
typedefs in Class Declarations . . . . .	405
Interactions with Other Products . . . . .	406

### Appendix B. Common Usage C Language Level. . . . . 407

### Appendix C. Conforming to POSIX 1003.1 . . . . . 409

### Appendix D. Conforming to ANSI/ISO Standards . . . . . 411

Implementation-Defined Behavior . . . . .	411
Identifiers . . . . .	411
Characters . . . . .	412
String Conversion . . . . .	413
Integers . . . . .	413
Floating-Point . . . . .	413

Arrays and Pointers . . . . .	414	COBOL FOR MVS & VM Release 2. . . . .	449
Registers. . . . .	414	COBOL for OS/390 & VM Version 2 Release 1 . . . . .	450
Structures, Unions, Enumerations, Bit Fields . . . . .	414	PL/I for MVS & VM Release 1 Modification 1 . . . . .	450
Declarators . . . . .	415	OS PL/I Version 2 Release 3 . . . . .	450
Statements . . . . .	415	VS FORTRAN Version 2 Release 6 . . . . .	450
Preprocessing Directives . . . . .	415	CICS/ESA Version 4 Release 1 . . . . .	450
Library Functions. . . . .	416	CICS Transaction Server for OS/390 Release 2 . . . . .	450
Error Handling . . . . .	416	DB2 Version 3 Release 1 . . . . .	451
Signals . . . . .	417	DB2 Version 4 Release 1 . . . . .	451
Translation Limits . . . . .	417	DB2 Version 5 Release 1 . . . . .	451
Streams, Records, and Files . . . . .	418	IMS/ESA Version 4 Release 1. . . . .	451
Memory Management . . . . .	419	IMS/ESA Version 5 Release 1. . . . .	451
Environment . . . . .	419	IMS/ESA Version 6 Release 1. . . . .	451
Localization . . . . .	420	QMF Version 3 Release 2 . . . . .	452
Time . . . . .	420	VSAM . . . . .	452
<b>Glossary. . . . .</b>	<b>421</b>	<b>INDEX. . . . .</b>	<b>453</b>
<b>Bibliography . . . . .</b>	<b>449</b>	<b>Readers' Comments — We'd Like to Hear from You . . . . .</b>	<b>463</b>
OS/390 . . . . .	449		
VS COBOL II Release 4. . . . .	449		

---

## 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 the IBM Director of Licensing, IBM Corporation, 500 Columbus Avenue, Thornwood, NY, 10594, USA.

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.

This publication documents *intended* Programming Interfaces that allow the customer to write OS/390 C/C++ programs.

Any interfaces, including service component interfaces, that are not documented in the OS/390 C/C++ publications are not formal interfaces. You should not build any dependencies on these interfaces, as IBM can change or remove interfaces at any time, without notice.

Any pointers in this publication to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of these Web sites. IBM accepts no responsibility for the content or use of non-IBM Web sites specifically mentioned in this publication or accessed through an IBM Web site that is mentioned in this publication.

---

## Standards

Extracts are reprinted from IEEE Std 1003.1—1990, IEEE Standard Information Technology—Portable Operating System Interface (POSIX)—Part 1: System Application Program Interface (API) [C language], copyright 1990 by the Institute of Electrical and Electronic Engineers, Inc.

Extracts are reprinted from IEEE P1003.1a Draft 6 July 1991, Draft Revision to Information Technology—Portable Operating System Interface (POSIX), Part 1:

System Application Program Interface (API) [C Language], copyright 1992 by the Institute of Electrical and Electronic Engineers, Inc.

Extracts are reprinted from IEEE Std 1003.2—1992, IEEE Standard Information Technology—Portable Operating System Interface (POSIX)—Part 2: Shells and Utilities, copyright 1990 by the Institute of Electrical and Electronic Engineers, Inc.

Extracts are reprinted from IEEE Std P1003.4a/D6—1992, IEEE Draft Standard Information Technology—Portable Operating System Interface (POSIX)—Part 1: System Application Program Interface (API)—Amendment 2: Threads Extension [C language], copyright 1990 by the Institute of Electrical and Electronic Engineers, Inc.

Extracts from *ISO/IEC 9899:1990* have been reproduced with the permission of the International Organization for Standardization (ISO) and the International Electrotechnical Commission (IEC). The complete standard can be obtained from any ISO or IEC member or from the ISO or IEC Central Offices, Case postale 56, CH - 1211 Geneva 20, Switzerland. Copyright remains ISO and IEC.

Extracts from X/Open Specification, Programming Languages, Issue 4 Release 2, copyright 1988, 1989, February 1992, by the X/Open Company Limited, have been reproduced with the permission of X/Open Company Limited. No further reproduction of this material is permitted without the written notice from the X/Open Company Ltd, UK.

---

## Trademarks

The following terms, which may be denoted by a single asterisk (\*), are trademarks of International Business Machines Corporation in the United States or other countries or both:

AD/Cycle	AFP	AIX
AIX/6000	AT	AS/400
BookManager	C Set ++	C/370
C/MVS	C++/MVS	Common User Access
CICS	CICS/ESA	CICSplex
COBOL/370	CUA	CT
DATABASE 2	DB2	DFSMS
DFSMS/MVS	DFSMSdfp	DRDA
ESCON	GDDM	Hiperspace
IBM	IBMLink	IMS
IMS/ESA	MVS/DFP	MVS/ESA
MVS/SP	MVS/XA	Open Class
OpenEdition	Operating System/2	Operating System/400
OS OPEN	OS/2	OS/390
OS/400	PROFS	PS/2
QMF	RACF	RETAIN
S/370	S/390	SAA
SOM	SOMobjects	SP
SQL/DS	System/370	System/390
System Object Model	Systems Application Architecture	VisualAge
VM/ESA	VSE/ESA	VTAM
3090	3890	400

Microsoft, Windows, Windows NT, and the Windows logo are registered trademarks of Microsoft Corporation.

UNIX is a registered trademark in the United States 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.



---

## Part 1. Introduction

This part describes how to use the OS/390 C/C++ Language Reference, and how to find additional information in the OS/390 C/C++ library. This part introduces the IBM OS/390 C/C++ product.

### **Chapter 1. About This Book**

Describes how to use this book in relation to the OS/390 C/C++ information library and related OS/390 documentation.

### **Chapter 2. About IBM OS/390 C/C++**

Introduces the OS/390 C/C++ product and its key features, related OS/390 environments such as OS/390 UNIX System Services, and other OS/390 tools that are useful when using OS/390 C/C++.





---

## Chapter 1. About This Book

This book describes the IBM C language and C++ language definitions which comply with the POSIX and XPG4 standards, and which the OS/390 Language Environment implements. Use this book if you are a programmer who needs to understand the support that IBM OS/390 C/C++ provides.

---

### Who Should Use This Book

This book is intended for programmers who will write C or C++ applications under the OS/390 operating system. This book is a reference rather than a tutorial. It assumes that you have some experience with writing C or C++ programs and are familiar with the OS/390 operating system.

---

### A Note about Examples

Examples that illustrate the use of the OS/390 C/C++ compiler use a simple style. They are instructional examples, and do not attempt to minimize run time, conserve storage, or check for errors. The examples do not demonstrate all the uses of C/C++ language constructs. Some examples are only code fragments and will not compile without additional code.

---

## IBM OS/390 C/C++ and Related Publications

This section summarizes the content of the IBM OS/390 C/C++ publications and shows where to find related information in other publications.

*Table 1. OS/390 C/C++ Publications*

Book Title and Number	Key Sections/Chapters in the Book
<i>OS/390 C/C++ Programming Guide</i> , SC09-2362	<p>Guidance information for:</p> <ul style="list-style-type: none"><li>• C/C++ input and output</li><li>• Debugging OS/390 C programs that use input/output</li><li>• Using linkage specifications in C++</li><li>• Combining C and assembler</li><li>• Creating and using DLLs</li><li>• Using threads in an OS/390 UNIX<sup>®</sup> application</li><li>• Using threads in an OS/390 UNIX application</li><li>• Reentrancy</li><li>• Using the decimal data type in C and C++</li><li>• Handling exceptions, error conditions, and signals</li><li>• Optimizing code</li><li>• Optimizing your C/C++ code with Interprocedural Analysis</li><li>• Network communications under OS/390 UNIX</li><li>• Interprocess communications using OS/390 UNIX</li><li>• Structuring a program that uses C++ templates</li><li>• Using environment variables</li><li>• Using System Programming C facilities</li><li>• Library functions for the System Programming C facilities</li><li>• Using runtime user exits</li><li>• Using the OS/390 C multitasking facility</li><li>• Using other IBM products with OS/390 C/C++ (CICS*, CSP, DWS, DB2*, GDDM*, IMS*, ISPF, QMF*)</li><li>• Direct-to-SOM support under OS/390 C/C++</li><li>• Internationalization: locales and character sets, code set conversion utilities, mapping variant characters</li><li>• POSIX character set</li><li>• Code point mappings</li><li>• Locales supplied with OS/390 C/C++</li><li>• Charmap files supplied with OS/390 C/C++</li><li>• Examples of charmap and locale definition source files</li><li>• Converting code from code character set IBM-1047</li><li>• Using built-in functions</li><li>• Programming considerations for OS/390 UNIX C/C++</li></ul>
<i>OS/390 C/C++ User's Guide</i> , SC09-2361	<p>Guidance information for:</p> <ul style="list-style-type: none"><li>• OS/390 C/C++ examples</li><li>• Compiler options</li><li>• Binder options and control statements</li><li>• Specifying OS/390 Language Environment runtime options</li><li>• Compiling, IPA Linking, binding, and running OS/390 C/C++ programs</li><li>• Using precompiled headers</li><li>• Utilities (Object Library, DLL Rename, CXXFILT, DSECT Conversion, Code Set and Locale, ar and make, BPXBATCH)</li><li>• Diagnosing problems</li><li>• Cataloged procedures and REXX EXECs supplied by IBM</li><li>• Error messages and return codes</li></ul>

Table 1. OS/390 C/C++ Publications (continued)

Book Title and Number	Key Sections/Chapters in the Book
<i>OS/390 C/C++ Language Reference</i> , SC09-2360	Reference information for: <ul style="list-style-type: none"> <li>• The C and C++ Languages</li> <li>• Lexical elements of OS/390 C and OS/390 C++</li> <li>• Declarations, expressions and operators</li> <li>• Implicit type conversions</li> <li>• Functions and statements</li> <li>• Preprocessor directives</li> <li>• C++ classes, class members, and friends</li> <li>• C++ overloading, special member functions, and inheritance</li> <li>• C++ templates and exception handling</li> <li>• OS/390 C and OS/390 C++ compatibility</li> </ul>
<i>OS/390 C/C++ Run-Time Library Reference</i> , SC28-1663	Reference information for: <ul style="list-style-type: none"> <li>• C header files</li> <li>• C Library functions</li> </ul>
<i>OS/390 C Curses</i> , SC28-1907	Reference information for: <ul style="list-style-type: none"> <li>• Curses concepts</li> <li>• Key data types</li> <li>• General rules for characters, renditions, and window properties</li> <li>• General rules of operations and operating modes</li> <li>• Use of macros</li> <li>• Restrictions on block-mode terminals</li> <li>• Curses functional interface</li> <li>• Contents of headers</li> <li>• The terminfo database</li> </ul>
<i>OS/390 C/C++ Compiler and Run-Time Migration Guide</i> , SC09-2359	Guidance and reference information for: <ul style="list-style-type: none"> <li>• Common migration questions</li> <li>• Application executable program compatibility</li> <li>• Source program compatibility</li> <li>• Input and output operations compatibility</li> <li>• Class library migration considerations</li> <li>• Changes between releases of OS/390</li> <li>• C/370* V1 to V2 compiler changes</li> <li>• Other migration considerations</li> </ul>
<i>OS/390 C/C++ Reference Summary</i> , SX09-1313	Summary tables for: <ul style="list-style-type: none"> <li>• Character set, trigraphs, digraphs, and keywords</li> <li>• Escape sequences, storage classes</li> <li>• Predefined and derived types, type qualifiers</li> <li>• Operator precedence, redirection symbols</li> <li>• fprintf() format, type characters, and flag characters</li> <li>• fscanf() format and type characters</li> <li>• __amrc structure</li> <li>• Hardware exceptions and signals</li> <li>• Compiler return codes</li> <li>• Compiler options</li> <li>• #pragma directives</li> <li>• Library functions</li> <li>• Utilities</li> </ul>

Table 1. OS/390 C/C++ Publications (continued)

Book Title and Number	Key Sections/Chapters in the Book
<i>OS/390 C/C++ IBM Open Class Library User's Guide</i> , SC09-2363	<p>Guidance information for:</p> <ul style="list-style-type: none"> <li>Using the Complex Mathematics Class Library: Review of complex numbers, header files, constructing complex objects, mathematical operators for complex, friend functions for complex, handling complex mathematics errors</li> <li>Using the I/O Stream Class Library: Introduction, getting started, advanced topics, and manipulators</li> <li>Using the Collection Class Library: Overview, instantiating and using, element and key functions, tailoring a collection implementation, polymorphic use of collections, support for notifications, exception handling, tutorials, problem solving, compatibility with previous releases, thread safety</li> <li>Using the Application Support Class Library: Introduction, String classes, Exception and Trace classes, Date and Time classes, controlling threads and protecting data, the IBM Open Class* notification framework, Binary Coded Decimal classes</li> </ul>
<i>OS/390 C/C++ IBM Open Class Library Reference</i> , SC09-2364	<p>Reference information for:</p> <ul style="list-style-type: none"> <li>Complex Mathematics Class Library</li> <li>I/O Stream Class Library</li> <li>Collection Class Library</li> <li>Application Support Class Library</li> </ul>
<i>OS/390 C/C++ SOM-Enabled Class Library User's Guide and Reference</i> , SC09-2366	<p>Guidance and reference information for:</p> <ul style="list-style-type: none"> <li>C++ SOM (RRBC-enabled) versions of Collection and Application Support Class Libraries</li> <li>Cross-language SOM version of the Collection Class Library</li> </ul>
<i>Debug Tool User's Guide and Reference</i> , SC09-2137	<p>Guidance and reference information for:</p> <ul style="list-style-type: none"> <li>Preparing to debug programs</li> <li>Debugging programs</li> <li>Using Debug Tool in different environments</li> <li>Language-specific information</li> <li>Debug Tool reference</li> </ul>
APAR and BOOKS files (Shipped with Program materials)	<p>Partitioned data set CBC.SCBCDOC on the product tape contains the members, APAR and BOOKS, which provide additional information for using the IBM OS/390 C/C++ licensed program, including:</p> <ul style="list-style-type: none"> <li>Isolating reportable problems</li> <li>Keywords</li> <li>Preparing an Authorized Program Analysis Report (APAR)</li> <li>Problem identification worksheet</li> <li>Maintenance on OS/390</li> <li>Late changes to OS/390 C/C++ publications</li> </ul>
<p><b>Note:</b> For complete and detailed information on linking and running with OS/390 Language Environment and using the OS/390 Language Environment runtime options, refer to the <i>OS/390 Language Environment Programming Guide</i>, SC28-1939. For complete and detailed information on using interlanguage calls, refer to <i>OS/390 Language Environment Writing Interlanguage Applications</i>, SC28-1943.</p>	

The following table lists the OS/390 C/C++ and related publications. The table groups the publications according to the tasks they describe.

Table 2. Publications by Task

Tasks	Books
Planning, preparing, and migrating to OS/390 C/C++	<ul style="list-style-type: none"> <li>• <i>OS/390 C/C++ Compiler and Run-Time Migration Guide</i>, SC09-2359</li> <li>• <i>OS/390 Language Environment Customization</i>, SC28-1941</li> <li>• <i>OS/390 UNIX System Services Planning</i>, SC28-1890</li> <li>• <i>OS/390 Planning for Installation</i>, GC28-1726</li> <li>• <i>OS/390 Task Atlas</i>, available on the OS/390 Library page on the World Wide Web (<a href="http://www.s390.ibm.com/os390/bkserv">http://www.s390.ibm.com/os390/bkserv</a>)</li> </ul>
Installing	<ul style="list-style-type: none"> <li>• <i>OS/390 Program Directory</i></li> <li>• <i>OS/390 Planning for Installation</i>, GC28-1726</li> <li>• <i>OS/390 Language Environment Customization</i>, SC28-1941</li> </ul>
Coding programs	<ul style="list-style-type: none"> <li>• <i>OS/390 C/C++ Run-Time Library Reference</i>, SC28-1663</li> <li>• <i>OS/390 C/C++ Language Reference</i>, SC09-2360</li> <li>• <i>OS/390 C/C++ Reference Summary</i>, SX09-1313</li> <li>• <i>OS/390 C/C++ Programming Guide</i>, SC09-2362</li> <li>• <i>OS/390 Language Environment Concepts Guide</i>, GC28-1945</li> <li>• <i>OS/390 Language Environment Programming Guide</i>, SC28-1939</li> <li>• <i>OS/390 Language Environment Programming Reference</i>, SC28-1940</li> <li>• <i>OS/390 C/C++ IBM Open Class Library User's Guide</i>, SC09-2363</li> <li>• <i>OS/390 C/C++ IBM Open Class Library Reference</i>, SC09-2364</li> <li>• <i>OS/390 C/C++ SOM-Enabled Class Library User's Guide and Reference</i>, SC09-2366</li> </ul>
Coding and binding programs with interlanguage calls	<ul style="list-style-type: none"> <li>• <i>OS/390 C/C++ Programming Guide</i>, SC09-2362</li> <li>• <i>OS/390 C/C++ Language Reference</i>, SC09-2360</li> <li>• <i>OS/390 Language Environment Programming Guide</i>, SC28-1939</li> <li>• <i>OS/390 Language Environment Writing Interlanguage Applications</i>, SC28-1943</li> <li>• <i>DFSMS/MVS Program Management</i>, SC26-4916</li> </ul>
Compiling, binding, and running programs	<ul style="list-style-type: none"> <li>• <i>OS/390 C/C++ User's Guide</i>, SC09-2361</li> <li>• <i>OS/390 Language Environment Programming Guide</i>, SC28-1939</li> <li>• <i>OS/390 Language Environment Debugging Guide and Run-Time Messages</i>, SC28-1942</li> <li>• <i>DFSMS/MVS Program Management</i>, SC26-4916</li> <li>• <i>OS/390 Messages Database</i>, available on the OS/390 Library page in the World Wide Web (<a href="http://www.s390.ibm.com/os390/bkserv">http://www.s390.ibm.com/os390/bkserv</a>)</li> </ul>
Compiling and binding applications in the OS/390 UNIX environment	<ul style="list-style-type: none"> <li>• <i>OS/390 C/C++ User's Guide</i>, SC09-2361</li> <li>• <i>OS/390 UNIX System Services User's Guide</i>, SC28-1891</li> <li>• <i>OS/390 UNIX System Services Command Reference</i>, SC28-1892</li> <li>• <i>DFSMS/MVS Program Management</i>, SC26-4916</li> </ul>
Compiling and binding SOM applications with OS/390 SOMobjects*	<ul style="list-style-type: none"> <li>• <i>OS/390 SOMobjects Programmer's Guide</i>, GC28-1859</li> <li>• <i>OS/390 C/C++ Programming Guide</i>, SC09-2362</li> <li>• <i>OS/390 C/C++ User's Guide</i>, SC09-2361</li> </ul>

Table 2. Publications by Task (continued)

Tasks	Books
Debugging programs	<ul style="list-style-type: none"> <li>• README file</li> <li>• <i>Debug Tool User's Guide and Reference</i>, SC09-2137</li> <li>• <i>OS/390 C/C++ User's Guide</i>, SC09-2361</li> <li>• <i>OS/390 C/C++ Programming Guide</i>, SC09-2362</li> <li>• <i>OS/390 Language Environment Programming Guide</i>, SC28-1939</li> <li>• <i>OS/390 Language Environment Debugging Guide and Run-Time Messages</i>, SC28-1942</li> <li>• <i>OS/390 UNIX System Services Messages and Codes</i>, SC28-1908</li> <li>• <i>OS/390 UNIX System Services User's Guide</i>, SC28-1891</li> <li>• <i>OS/390 UNIX System Services Command Reference</i>, SC28-1892</li> <li>• <i>OS/390 UNIX System Services Programming Tools</i>, SC28-1904</li> </ul>
Using shells and utilities in the OS/390 UNIX environment	<ul style="list-style-type: none"> <li>• <i>OS/390 C/C++ User's Guide</i>, SC09-2361</li> <li>• <i>OS/390 UNIX System Services Command Reference</i>, SC28-1892</li> <li>• <i>OS/390 UNIX System Services Messages and Codes</i>, SC28-1908</li> </ul>
Using sockets library functions in the OS/390 UNIX environment	<ul style="list-style-type: none"> <li>• <i>OS/390 C/C++ Run-Time Library Reference</i>, SC28-1663</li> </ul>
Porting a UNIX Application to OS/390	<ul style="list-style-type: none"> <li>• <i>OS/390 UNIX System Services Porting Guide</i> This guide contains useful information about supported header files and C functions, sockets in an OS/390 UNIX environment, process management, compiler optimization tips, and suggestions for improving the application's performance after it has been ported. The <i>Porting Guide</i> is available as a PDF file which you can download, or as web pages which you can browse, at the following URL: <a href="http://www.s390.ibm.com/unix/bpxalpor.html">http://www.s390.ibm.com/unix/bpxalpor.html</a></li> </ul>
Performing diagnosis and submitting an Authorized Program Analysis Report (APAR)	<ul style="list-style-type: none"> <li>• <i>OS/390 C/C++ User's Guide</i>, SC09-2361</li> <li>• CBC.SCBCDOC(APAR) on OS/390 C/C++ product tape</li> </ul>
Quick reference	<ul style="list-style-type: none"> <li>• <i>OS/390 C/C++ Reference Summary</i>, SX09-1313</li> </ul>
Multimedia Tutorial	<ul style="list-style-type: none"> <li>• For a new way of learning C++ programming, you can order the CD-ROM <i>Experience C++: A Multimedia Tutorial</i>, SK2T-1158. This tutorial runs in DOS.</li> </ul>

**Note:** For information on using the prelinker, see the appendix on prelinking and linking OS/390 C/C++ programs in the *OS/390 C/C++ User's Guide*. As of Release 4, this appendix contains information that was previously in the chapter on prelinking and linking OS/390 C/C++ programs in the *OS/390 C/C++ User's Guide*. It also contains prelinker information that was previously in the *OS/390 C/C++ Programming Guide*.

## Hardcopy Books

The following OS/390 C/C++ books are available in hardcopy:

- *OS/390 C/C++ Run-Time Library Reference*, SC28-1663
- *OS/390 C/C++ User's Guide*, SC09-2361
- *OS/390 C/C++ Programming Guide*, SC09-2362
- *OS/390 C/C++ Reference Summary*, SX09-1313
- *OS/390 C/C++ IBM Open Class Library User's Guide*, SC09-2363

- *OS/390 C Curses*, SC28-1907
- *OS/390 C/C++ Compiler and Run-Time Migration Guide*, SC09-2359
- *Debug Tool User's Guide and Reference*, SC09-2137

You can purchase these books on their own, or as part of a set. You receive the *OS/390 C/C++ Compiler and Run-Time Migration Guide*, SC09-2359 at no charge. Feature code 8009 includes the remaining books.

---

## Softcopy Books

All of the OS/390 C/C++ publications (except for the *OS/390 C/C++ Reference Summary*) are available in softcopy book format. The books are available on the tape that accompanies the OS/390 product, and on a CD-ROM called the *IBM Online Library Omnibus Edition: OS/390 Collection*, SK2T-6700.

To read the softcopy books, the BookManager\* Read (Program 5684-062, 5695-046) licensed program must be available on your operating system. BookManager Read provides access to online information as an alternative to hard copy documents. You can read, search, make notes, and select sections of text to print.

Also available are BookManager Read/DOS (Program 73F6-022) for the DOS operating system, and BookManager Read/2 (Program 73F6-023) for the OS/2 operating system. With these products, you can download online books to your workstation and read them.

If your system has BookManager Read installed, you can enter the command BOOKMGR to start BookManager and display a list of books available to you. If you know the name of the book that you want to view, you can use the OPEN command to open the book directly.

**Note:** If your workstation does not have graphics capability, BookManager Read cannot correctly display some characters, such as arrows and brackets.

You can also browse the books on the World Wide Web by clicking on "The Library" link on the OS/390 home page. The URL for this page is:

<http://www.s390.ibm.com/os390/index.html>

---

## Softcopy Examples

Most of the larger examples in the following books are available in machine-readable form:

- *OS/390 C/C++ Language Reference*, SC09-2360
- *OS/390 C/C++ User's Guide*, SC09-2361
- *OS/390 C/C++ Programming Guide*, SC09-2362
- *OS/390 C/C++ IBM Open Class Library User's Guide*, SC09-2363
- *OS/390 C/C++ IBM Open Class Library Reference*, SC09-2364
- *OS/390 C/C++ SOM-Enabled Class Library User's Guide and Reference*, SC09-2366

In the following books, a label on an example indicates that the example is distributed in softcopy. The label is the name of a member in the data sets CBC.SCBCSAM or CBC.SCLBSAM. The labels have the form CBCxyyy or CLBxyyy, where x refers to a publication:

- R and X refer to the *OS/390 C/C++ Language Reference*, SC09-2360
- G refers to the *OS/390 C/C++ Programming Guide*, SC09-2362



- U refers to the *OS/390 C/C++ User's Guide*, SC09-2361
- A refers to the *OS/390 C/C++ IBM Open Class Library User's Guide*, SC09-2363

Examples labelled as CBCxyyy appear in the *OS/390 C/C++ Language Reference*, the *OS/390 C/C++ Programming Guide*, and the *OS/390 C/C++ User's Guide*. Examples labelled as CLBxyyy appear in the *OS/390 C/C++ IBM Open Class Library User's Guide*.

An exception applies to the example names for the Collection Class Library which do not follow a naming convention. These examples are in the *OS/390 C/C++ IBM Open Class Library Reference*, SC09-2364 and in the *OS/390 C/C++ SOM-Enabled Class Library User's Guide and Reference*, SC09-2366. For the *OS/390 C/C++ SOM-Enabled Class Library User's Guide and Reference*, SC09-2366, the label refers to a member name in the data set CBC.SCLBXSM.

---

## OS/390 C/C++ on the World Wide Web

Additional information on OS/390 C/C++ is available on the World Wide Web. The URL for the OS/390 C/C++ home page is:

<http://www.software.ibm.com/ad/c390/index.html>

This page contains late-breaking information about the OS/390 C/C++ product, including the compiler, the class libraries, and utilities. It also contains a tutorial on the source level interactive debugger. There are links to other useful information, such as the OS/390 C/C++ information library and the libraries of other OS/390 elements that are available on the Web. The OS/390 C/C++ home page also contains information on active Beta programs, samples that you can download, C/370 product newsletters, and links to other related Web sites.

---

## C/C++ News...

IBM also publishes the *C/370 Compiler Newsletter*. This free newsletter keeps subscribers up to date on the latest product releases. It also provides coding hints and tips, questions and answers, and news about C/370 products and IBM OS/390 C/C++.

To take advantage of this free publication, send your name, full mailing address, and phone number, as follows:

- Send a message electronically to the following network ID :
  - Internet: [inetc370@ca.ibm.com](mailto:inetc370@ca.ibm.com)
  - IBMMAIL: [ibmmail\(caibmrxz\)](mailto:ibmmail(caibmrxz))
- Mail your request to:
 

EDITOR, C/370 Compiler Newsletter  
 IBM Canada Ltd. Laboratory  
 9/604/895/TOR  
 895 Don Mills Road  
 NORTH YORK ONTARIO CANADA M3C 1W3

---

## How to Read the Syntax Diagrams

This book describes the syntax for commands, directives, and statements, using the following structure:



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

A double right arrowhead indicates the beginning of a command, directive, or statement. A single right arrowhead indicates that it is continued on the next line. In the following diagrams, "statement" represents a command, directive, or statement.



The following indicates a continuation; the opposing arrowheads indicate the end of a command, directive, or statement.



Diagrams of syntactical units other than complete commands, directives, or statements look like this:



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



- Optional items are below the main path.



- If you can choose from two or more items, they are vertical in a stack. If you *must* choose one of the items, one item of the stack is on the main path.



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



- An arrow that returns to the left above the main line indicates an item that you can repeat.

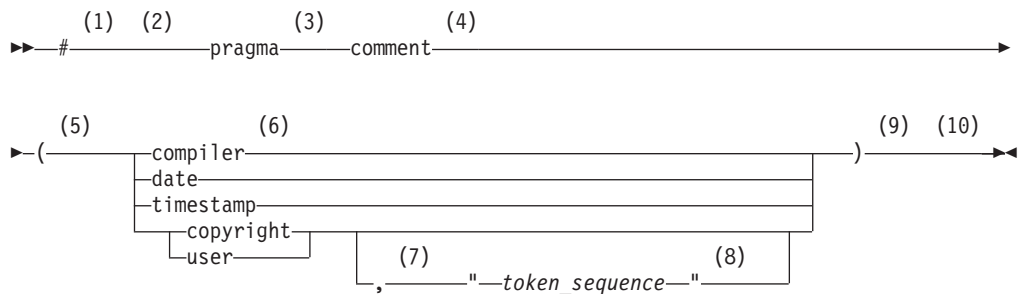


A repeat arrow above a stack indicates that you can make more than one choice from the stacked items, or repeat a single choice.

- Keywords are not italicized, and should be entered exactly as shown (for example, `pragma`). You must spell keywords exactly as shown in the syntax diagram. Variables are in lowercase italics (in hardcopy), for example, *identifier*. They represent user-supplied names or values.
- If the syntax diagram shows punctuation marks, parentheses, arithmetic operators, or other nonalphanumeric characters, you must enter them as part of the syntax.

**Note:** You do not always require the white space between tokens. You should, however, include at least one blank space between tokens unless otherwise specified.

The following syntax diagram example shows the syntax for the `#pragma comment` directive.



#### Notes:

- 1 This is the start of the syntax diagram.
- 2 The symbol `-#` must appear first.
- 3 The keyword `-pragma` must follow the `-#` symbol.
- 4 The keyword `-comment` must follow the keyword `-pragma`.
- 5 An opening parenthesis must follow the keyword `-comment`.
- 6 The comment type must be entered only as one of the following: `-compiler`, `-date`, `-timestamp`, `-copyright`, or `-user`.
- 7 If the comment type is `-copyright` or `-user`, and an optional character string is following, a comma must be present after the comment type.
- 8 A character string must follow the comma. The character string must be enclosed in double quotation marks.
- 9 A closing parenthesis is required.
- 10 This is the end of the syntax diagram.

The following examples of the `#pragma comment` directive are syntactically correct according to the diagram above:

```

#pragma comment(date)
#pragma comment(user)
#pragma comment(copyright,"This text will appear in the module")

```

---

## Chapter 2. About IBM OS/390 C/C++

The C/C++ feature of the IBM OS/390 licensed program provides support for C and C++ application development on the OS/390 platform. The C/C++ feature is based on the C/C++ for MVS/ESA\* product.

IBM OS/390 C/C++ includes:

- A C compiler (referred to as the OS/390 C compiler)
- A C++ compiler (referred to as the OS/390 C++ compiler)
- A set of C++ class libraries
- Application Support Class and Collection Class Library source
- A mainframe interactive Debug Tool (optional)
- A set of utilities for C/C++ application development

IBM offers the C language on other platforms, such as the AIX\*, IBM Operating System/2\* (OS/2\*), IBM Operating System/400\* Version 3 (OS/400\*), Sun Solaris, VM/ESA\*, VSE/ESA\*, and Windows® operating systems. The AIX, OS/2, OS/400, Sun Solaris, and Windows operating systems also offer the C++ language.

---

### Changes for Version 2 Release 6

OS/390 C/C++ has made the following changes for this release:

- Added support for the Institute of Electrical and Electronics Engineers (IEEE) binary floating-point data type, in conformance with the IEEE 754 standard, as applicable to the S/390\* environment. For details on the OS/390 C/C++ support, see the description of the FLOAT option in the *OS/390 C/C++ User's Guide*. In addition, two related sub-options have been introduced, ARCH(3) and TUNE(3). The two sub-options support the new G5 processor architecture, and IEEE binary floating-point data. Refer to the ARCHITECTURE and TUNE compiler options in the *OS/390 C/C++ User's Guide* for details.

Complete IEEE binary floating-point support for OS/390 and its elements requires that you apply small programming enhancements (SPEs) to OS/390 V2R6.0, and to specific releases of some software. These SPEs are delivered as program temporary fixes (PTFs). Consult your System Programmer to ensure that the SPE PTFs you require for IEEE binary floating-point support, as documented in the *OS/390 Planning for Installation* publication, are applied to your system. The *OS/390 Planning for Installation* publication documents the complete software requirements for IEEE binary floating-point support on OS/390.

- Improved the performance of the Binary Coded Decimal (BCD) class library, and its compatibility with the decimal data type in C, and other S/390 languages. For details, see *Using the C++ Decimal Data Type* in the *OS/390 C/C++ Programming Guide*.
- Added support for the long long integer data type. For more details, see "Integer Variables" on page 89 and "Integer Constants" on page 60. The run-time library, including functions such as printf() and scanf(), does not support the long long data type at this time.
- Added a new compiler option, PORT, that enables you to increase the syntax checking for the #pragma pack directive in your code. This option is helpful

when porting code that contains `#pragma pack` directives or packed data from other platforms. For more information on the `PORT` option, see the *OS/390 C/C++ User's Guide*.

- Added a new compiler option, `FASTTEMPINC`, that enables you to improve your compilation time for C++ class templates if you use a large number of recursive templates in an application. For more information on the `FASTTEMPINC` option, see the *OS/390 C/C++ User's Guide*.
- Retroactive to OS/390 Version 1 Release 3, the IBM Open Class Library is licensed with the base operating system. This enables applications to use this library at run time without having to license the OS/390 C/C++ compiler feature(s) or to use the DLL Rename Utility.
- The level of optimization you get when you specify the `OPT(1)`, or `OPT`, compiler option is the same as when you specify `OPT(2)`. For more information on the `OPTIMIZATION` option see the *OS/390 C/C++ User's Guide*.
- The OS/390 C++ class library header files are now distributed in the hierarchical file system (HFS) in directory `/usr/lpp/ioclib/include`.
- As part of the name change of *OpenEdition\** to *OS/390 UNIX System Services*, occurrences of *OpenEdition* have been changed to *OS/390 UNIX System Services* or its abbreviated name, *OS/390 UNIX*, throughout the OS/390 C/C++ information library. *OpenEdition* may continue to appear in messages, panel text, and other code locations.

---

## The C/C++ Compilers

The following sections describe the C and C++ languages and the OS/390 C/C++ compilers.

### The C Language

The C language is a general purpose, versatile, and functional programming language, which allows a programmer to create applications quickly and easily. C provides high-level control statements and data types as do other structured programming languages. It also provides many of the benefits of a low-level language.

### The C++ Language

The C++ language is based on the C language, but incorporates support for object-oriented concepts. Refer to “Appendix A. C and C++ Compatibility” on page 401 for a detailed description of the differences between OS/390 C++ and OS/390 C.

The C++ language introduces classes, which are user-defined data types that may contain data definitions and function definitions. You can use classes from established class libraries, develop your own classes, or derive new classes from existing classes by adding data descriptions and functions. New classes can inherit properties from one or more classes. Not only do classes describe the data types and functions available, but they can also hide (encapsulate) the implementation details from user programs. An object is an instance of a class.

The C++ language also provides templates and other features that include access control to data and functions, and better type checking and exception handling. It also supports polymorphism and the overloading of operators.

## Common Features of the OS/390 C and C++ Compilers

The C or C++ compilers offer many features to help your work:

- Optimization support.
  - Algorithms to take advantage of S/390 architecture to get better optimization for speed and use of computer resources through the OPTIMIZE and IPA compile-time options.
  - The OPTIMIZE compile-time option to instruct the compiler to optimize the machine instructions it generates, to produce faster-running object code, thereby optimizing application performance at run time.
  - Interprocedural Analysis (IPA), to perform optimizations across compilation units, thereby optimizing application performance at run time.
  - The precompiled header facility, to save information from one compilation unit for use in another or to reuse information when re-compiling the source compilation unit, thereby improving performance at compile time.
- DLLs (dynamic link libraries) to reduce application size, and dynamically link to exported variables and functions at run time.

IBM OS/390 C/C++ provides support for generating DLLs in a way similar to the way OS/2 generates DLLs. DLLs allow a function reference or a variable reference in one executable to use a definition located in another executable at run time. You can use both load-on-reference and load-on-demand DLLs. When your program calls a DLL function, or references a DLL, IBM OS/390 C/C++ provides a load-on-reference DLL. Your application code explicitly controls load-on-demand DLLs at the source level.

You can use DLLs to split applications into smaller modules and improve system memory usage. DLLs also offer more flexibility for building, packaging, and redistributing applications.

- Full program reentrancy.

With reentrancy, many users can simultaneously run a program. A reentrant program uses less storage if it is stored in the LPA (link pack area) or ELPA (extended link pack area) and simultaneously run by multiple users. It also reduces processor I/O when the program starts up, and improves program performance by reducing the transfer of data to auxiliary storage. OS/390 C programmers can design programs that are naturally reentrant. For those programs that are not naturally reentrant, C programmers can use constructed reentrancy. To do this, compile programs with the RENT option and use the program management binder supplied with OS/390, or the OS/390 Language Environment Prelinker (prelinker) and program management binder. The OS/390 C++ compiler always ensures that C++ programs are reentrant.
- Locale-based internationalization support derived from the IEEE POSIX 1003.2-1992 standard. Also derived from the X/Open CAE Specification, System Interface Definitions, Issue 4 and Issue 4 Version 2. This allows programmers to use locales to specify language/country characteristics for their applications.
- The ability to call and be called by other languages such as assembler, COBOL, PL/1, and Fortran, to enable programmers to integrate OS/390 C/C++ code with existing applications.
- Exploitation of OS/390 and OS/390 UNIX technology.

OS/390 UNIX is an IBM implementation of the open operating system environment, as defined in the XPG4 and POSIX standards.
- When used with OS/390 UNIX and OS/390 Language Environment, support for the following standards at the system level:

- A subset of the extended multibyte and wide character functions as defined by the Programming Language C Amendment 1. This is ISO/IEC 9899:1990/Amendment 1:1994(E)
- ISO/IEC 9945-1:1990(E)/IEEE POSIX 1003.1-1990
- A subset of IEEE POSIX 1003.1a, Draft 6, July 1991
- IEEE Portable Operating System Interface (POSIX) Part 2, P1003.2
- A subset of IEEE POSIX 1003.4a, Draft 6, February 1992 (the IEEE POSIX committee has renumbered POSIX.4a to POSIX.1c)
- X/Open CAE Specification, System Interfaces and Headers, Issue 4 Version 2
- A subset of IEEE 754-1985 (R1990) IEEE Standard for Binary Floating-Point Arithmetic (ANSI), as applicable to the S/390 environment.
- X/Open CAE Specification, Network Services, Issue 4
- Year 2000 support.

## OS/390 C Compiler Specific Features

In addition to the features common to OS/390 C/C++, the OS/390 C compiler provides you with the following capabilities:

- The ability to write portable code that conforms to the following standards:
  - All elements of the ISO standard ISO/IEC 9899:1990 (E)
  - ANSI/ISO 9899:1990[1992] (formerly ANSI X3.159-1989 C)
  - X/Open Specification Programming Language Issue 3, Common Usage C
  - FIPS-160
- System programming capabilities, which allow you to use OS/390 C in place of assembler
- Additional optimization capabilities through the `INLINE` compile-time option
- Extensions of the standard definitions of the C language to provide programmers with support for the OS/390 environment, such as fixed-point (packed) decimal data support

## Features That Are Specific to the OS/390 C++ Compiler

In addition to the features common to OS/390 C/C++, the OS/390 C++ compiler provides you with the following:

- An implementation based on the definition of the language that is contained in the Draft Proposal International Standard for Information Systems–Programming Language C++ (X3J16/92-00091). The OS/390 C++ compiler also conforms to a subset of the C++ ANSI/ISO (Draft) Standard (X3J16/93-0062).
- System Object Model (SOM) support, through the SOM Interface Definition Language (IDL) compiler available with OS/390 SOMobjects. You can use the IDL compiler and associated emitters to create language-specific bindings that define the interface to a SOM object. This enables OS/390 C++ programs to share SOM objects with other languages. In addition, SOM enables release-to-release binary compatibility.

With Direct-to-SOM (DTS) support in the OS/390 C++ compiler, you can generate SOM objects directly from C++ code. You do not need to create and process the IDL first. You can write virtually the same code you do when creating C++ objects.

**Note:** The OS/390 C++ compiler no longer supports IDL generation through the IDL compile-time option. This option instructed the compiler to generate

IDL. Mixed-language or distributed object applications used IDL. If you need IDL for your applications, you should now code it yourself instead of generating it through the IDL compile option.

- C++ template support and exception handling consistent with VisualAge\* C++ product implementations.

---

## Utilities

The OS/390 C/C++ compilers provide the following utilities:

- The Object Library Utility to update partitioned data set (PDS) libraries of object modules and Interprocedural Analysis (IPA) object modules
- The DLL Rename Utility to make selected DLLs a unique component of the applications with which they are packaged
- The CXXFILT Utility to map OS/390 C++ mangled names to the original source
- The localedef Utility to read the locale definition file and produce a locale object that the locale-specific library functions can use
- The DSECT Conversion Utility to convert descriptive assembler DSECTs into OS/390 C/C++ data structures
- The C/C++ Model Tool to provide online help for C/C++ #pragma directives and runtime library functions. These functions are other than the C Curses functions, and are at the level that is supplied in OS/390 Release 2

---

## Class Libraries

IBM OS/390 C/C++ provides a base set of class libraries, called C/C++ IBM Open Class, which is consistent with that available in other members of the VisualAge C++ product family. These class libraries are:

- The I/O Stream Class Library

The I/O Stream Class Library lets you perform input and output (I/O) operations independent of physical I/O devices or data types that are used. You can code sophisticated I/O statements easily and clearly, and define input and output for your own data types. You can improve the maintainability of programs that use input and output by using the I/O Stream Class Library.

- The Complex Mathematics Class Library

The Complex Mathematics Class Library lets you manipulate and perform standard arithmetic on complex numbers. Scientific and technical fields use complex numbers.

- The Application Support Class Library

The Application Support Class Library provides the basic abstractions that are needed during the creation of most C++ applications, including String, Date, and Time.

The Application Support Class library is available in a C++ SOM version as well as the regular C++ native version.

- The Collection Class Library

The Collection Class Library implements a wide variety of classical data structures such as stack, tree, list, hash table, and so on. Most programs use collections. You can develop programs without having to define every collection. Programmers can start programming by using a high level of abstraction, and later replace an abstract data type with the appropriate concrete implementation. Each abstract data type has a common interface for all of its implementations. The Collection Class Library provides programmers with a consistent set of



building blocks from which they can derive application objects. The library design exploits features of the C++ language such as exception handling and template support.

The Collection Class Library is available in a C++ SOM and a cross-language SOM version, as well as the regular C++ native version.

All of the libraries that are described above are thread-safe, except the cross-language SOM version of the Collection Class Library.

All of the libraries that are described above are available in both static and DLL formats. OS/390 C/C++ packages the Application Support Class and Collection Class libraries together in a single DLL. For compatibility, separate side-decks are available for the Application Support Class and Collection Class libraries, in addition to the side-deck available for the combined library.

**Note:** Retroactive to OS/390 Version 1 Release 3, the IBM Open Class Library is licensed with the base operating system. This enables applications to use this library at run time without having to license the OS/390 C/C++ compiler feature(s) or to use the DLL Rename Utility.

## Class Library Source

The Class Library Source consists of the following:

- Application Support Class Library source code
- Collection Class Library source code (C++ native and C++ SOM only)
- Instructions for building the Application Support Class and Collection Class Libraries in C++ native (static and DLL) versions
- Instructions for building the Application Support Class and Collection Class Libraries in C++ SOM (static and DLL) versions
- Class Library Language Environment message file source
- Instructions for building the Class Library Language Environment message files

---

## The Debug Tool

IBM OS/390 C/C++ supports program development by using a mainframe interactive Debug Tool. This optionally available tool allows you to debug applications in their native host environment, such as CICS/ESA, IMS/ESA\*, DB2, and so on. The Debug Tool provides the following support and function:

- Step mode
- Breakpoints
- Monitor
- Frequency analysis
- Dynamic patching

You can record the debug session in a log file, and replay the session. You can also use the Debug Tool to help capture test cases for future program validation or to further isolate a problem within an application.

You can specify either data sets or hierarchical file system (HFS) files as source files.

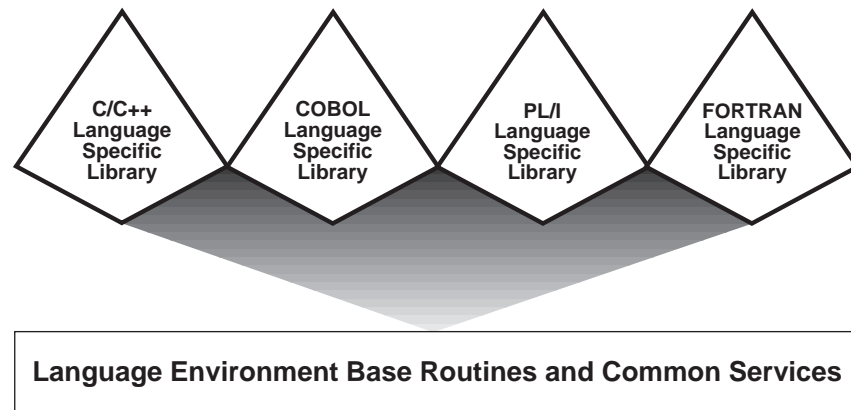


---

## OS/390 Language Environment

IBM OS/390 C/C++ exploits the C/C++ runtime environment and library of runtime services available with OS/390 Language Environment (formerly Language Environment for MVS & VM, Language Environment/370 and LE/370).

OS/390 Language Environment consists of four language-specific runtime libraries, and Base Routines and Common Services; see Figure 1. OS/390 Language Environment establishes a common runtime environment and common runtime services for language products, user programs, and other products.



*Figure 1. Libraries in OS/390 Language Environment*

The common execution environment is composed of data items and services that are included in library routines available to an application that runs in the environment. The OS/390 Language Environment provides a variety of services:

- Services that satisfy basic requirements common to most applications. These include support for the initialization and termination of applications, allocation of storage, interlanguage communication (ILC), and condition handling.
- Extended services that are often needed by applications. OS/390 C/C++ contains these functions within a library of callable routines, and include interfaces to operating system functions and a variety of other commonly used functions.
- Runtime options that help in the execution, performance, and diagnosis of your application.
- Access to operating system services; OS/390 UNIX services are available to an application programmer or program through the OS/390 C/C++ language bindings.
- Access to language-specific library routines, such as the OS/390 C/C++ library functions.

---

## The Program Management Binder

The binder provided with OS/390 combines the object modules, load modules, and program objects comprising an OS/390 application. It produces a single output program object or load module that you can load for execution. The binder supports all C and C++ code, provided that you store the output program in a PDSE (Partitioned Data Set Extended) member or an HFS file.

If you cannot use a PDSE member or HFS file, and your program contains C++ code, or C code that is compiled with any of the RENT, LONGNAME, DLL or IPA compile-time options, you must use the prelinker.

Using the binder without using the prelinker has the following advantages:

- Faster rebinds when recompiling and rebinding a few of your source files
- Rebinding at the single compile unit level of granularity (except when you use the IPA compile-time option)
- Input of object modules, load modules, and program objects
- Improved long name support:
  - Long names do not get converted into prelinker generated names
  - Long names appear in the binder maps, enabling full cross-referencing
  - Variables do not disappear after prelink
  - Fewer steps in the process of producing your executable program

The prelinker provided with OS/390 Language Environment combines the object modules comprising an OS/390 C/C++ application and produces a single object module. You can link-edit the object module into a load module (which is stored in a PDS), or bind it into a load module or a program object stored in a PDS, or a PDSE or HFS file.

---

## OS/390 UNIX System Services (OS/390 UNIX)

OS/390 UNIX provides capabilities under OS/390 to make it easier to implement or port applications in an open, distributed environment. OS/390 UNIX Services are available to OS/390 C/C++ application programs through the C/C++ language bindings available with OS/390 Language Environment.

Together, the OS/390 UNIX Services, OS/390 Language Environment, and OS/390 C/C++ compilers provide an application programming interface that supports industry standards.

OS/390 UNIX provides support for both existing OS/390 applications and new OS/390 UNIX applications:

- C programming language support as defined by ISO/ANSI C
- C++ programming language support
- C language bindings as defined in the IEEE 1003.1 and 1003.2 standards; subsets of the draft 1003.1a and 1003.4a standards; X/Open CAE Specification: System Interfaces and Headers, Issue 4, Version 2, which provides standard interfaces for better source code portability with other conforming systems; and X/Open CAE Specification, Network Services, Issue 4, which defines the X/Open UNIX descriptions of sockets and X/Open Transport Interface (XTI)
- OS/390 UNIX Extensions that provide OS/390-specific support beyond the defined standards
- The OS/390 UNIX Shell and Utilities feature, which provides:
  - A shell, based on the Korn Shell and compatible with the Bourne Shell
  - Tools and utilities that conform to the *X/Open Single UNIX Specification*, also known as *X/Open Portability Guide (XPG) Version 4, Issue 2*, and provide OS/390 support. The following utilities are included:

<b>ar</b>	Creates and maintains library archives
-----------	--

- |                 |  |
|-----------------|--|
| <b>BPXBATCH</b> | Allows you to submit batch jobs that run shell commands, scripts, or OS/390 C/C++ executable files in HFS files from a shell session   |
| <b>c89</b>      | Compiles, assembles, and binds OS/390 UNIX C applications  |
| <b>gencat</b>   | Merges the message text source files Messagefile (usually *.msg) into a formatted message Catalogfile (usually *.cat)  |
| <b>lex</b>      | Automatically writes large parts of a lexical analyzer based on a description that is supplied by the programmer   |
| <b>make</b>     | Helps you manage projects containing a set of interdependent files, such as a program with many OS/390 C/C++ source and object files, keeping all such files up to date with one another |
| <b>yacc</b>     | Allows you to write compilers and other programs that parse input according to strict grammar rules  |
- Support for other utilities such as:
- |                  |   |
|------------------|---|
| <b>c++</b>       | Compiles, assembles, and binds OS/390 UNIX C++ applications                                       |
| <b>mkcatdefs</b> | Preprocesses a message source file for input to the gencat utility                                |
| <b>runcat</b>    | Invokes mkcatdefs and pipes the message catalog source data (the output from mkcatdefs) to gencat |
| <b>dspcat</b>    | Displays all or part of a message catalog   |
| <b>dspmsg</b>    | Displays a selected message from a message catalog  |
- The OS/390 UNIX Debugger feature, which provides the dbx interactive symbolic debugger for OS/390 UNIX applications
  - OS/390 UNIX, which provides access to a hierarchical file system (HFS), with support for the POSIX.1 and XPG4 standards
  - OS/390 C/C++ I/O routines, which support using HFS files, standard OS/390 data sets, or a mixture of both
  - Application threads (with support for a subset of POSIX.4a)
  - Support for OS/390 C/C++ DLLs

OS/390 UNIX offers program portability across multivendor operating systems, with support for POSIX.1, POSIX.1a (draft 6), POSIX.2, POSIX.4a (draft 6), and XPG4.2.

To application developers who have worked with other UNIX environments, the OS/390 UNIX Shell and Utilities are a familiar environment for C/C++ application development. If you are familiar with existing MVS development environments, you may find that the OS/390 UNIX environment can enhance your productivity. Refer to the *OS/390 UNIX System Services User's Guide* for more information on the Shell and Utilities.

---

## OS/390 C/C++ Applications with OS/390 UNIX C/C++ Functions

Most OS/390 UNIX C functions are available at all times. However, to use some OS/390 UNIX C functions, you must run an OS/390 C/C++ program on a system where the OS/390 UNIX kernel is available and active. In some situations, you must also specify the POSIX(0N) runtime option. This is required for the POSIX.4a

threading functions, and the system and signal handling functions where the behavior is different between POSIX/XPG4 and ANSI. Refer to the *OS/390 C/C++ Run-Time Library Reference* for more information about requirements for each function.

You can invoke an OS/390 C/C++ program that uses OS/390 UNIX C functions using the following methods:

- Directly from the OS/390 UNIX Shell.
- From another program, or from the OS/390 UNIX Shell, using one of the `exec` family of functions, or the `BPXBATCH` utility from TSO or MVS batch.
- Using the `POSIX system()` call.
- Directly through TSO or MVS batch without the use of the intermediate `BPXBATCH` utility. In some cases, you may require the `POSIX(ON)` runtime option.

---

## Input and Output

The C/C++ runtime library that supports the OS/390 C/C++ compiler supports different input and output (I/O) interfaces, file types, and access methods. The C++ I/O Stream Class Library provides additional support.

### I/O Interfaces

The C/C++ runtime library supports the following I/O interfaces:

#### C Stream I/O

This is the default and the ANSI-defined I/O method. This method processes all input and output by character.

#### Record I/O

The library can also process your input and output by record. A record is a set of data that is treated as a unit. It can also process VSAM data sets by record. Record I/O is an OS/390 C/C++ extension to the ANSI standard.

#### TCP/IP Sockets I/O

OS/390 UNIX provides support for an enhanced version of an industry-accepted protocol for client/server communication that is known as *sockets*. A set of C language functions provides support for OS/390 UNIX sockets. OS/390 UNIX sockets correspond closely to the sockets that are used by UNIX applications that use the Berkeley Software Distribution (BSD) 4.3 standard (also known as OE sockets). The slightly different interface of the X/Open CAE Specification, Networking Services, Issue 4, is supplied as an additional choice. This interface is known as X/Open Sockets.

The OS/390 UNIX socket application program interface (API) provides support for both UNIX domain sockets and Internet domain sockets. UNIX domain sockets, or *local sockets*, allow interprocess communication within OS/390 independent of TCP/IP. Local sockets behave like traditional UNIX sockets and allow processes to communicate with one another on a single system. With Internet sockets, application programs can communicate with others in the network using TCP/IP.

In addition, the C++ I/O Stream Library supports formatted I/O in C++. You can code sophisticated I/O statements easily and clearly, and define input and output for your own data types. This helps improve the maintainability of programs that use input and output.

## File Types

In addition to conventional files, such as sequential files and partitioned data sets, the C/C++ runtime library supports the following file types:

### Virtual Storage Access Method (VSAM) Data Sets

OS/390 C/C++ has native support for three types of VSAM data organization:

- Key-sequenced data sets (KSDS). Use KSDS to access a record through a key within the record. A key is one or more consecutive characters that are taken from a data record that identifies the record.
- Entry-sequenced data sets (ESDS). Use ESDS to access data in the order it was created (or in the reverse order).
- Relative-record data sets (RRDS). Use RRDS for data in which each item has a particular number (for example, a telephone system with a record associated with each number).

For more information on how to perform I/O operations on these VSAM file types, see the *OS/390 C/C++ Programming Guide*.

### Hierarchical File System Files

When you are running under MVS, TSO (batch and interactive), or IMS environments, OS/390 C/C++ recognizes a Hierarchical File System (HFS) file. The name specified on the `fopen()` or `freopen()` call has to conform to certain rules (described in the *OS/390 C/C++ Programming Guide*). You can create regular HFS files, special character HFS files, or FIFO HFS files. You can also create links or directories.

### Memory Files

Memory files are temporary files that reside in memory. For improved performance, you can direct input and output to memory files rather than to devices. Since memory files reside in main storage and only exist while the program is executing, you primarily use them as work files. You can access memory files across load modules through calls to `non-POSIX system()` and `C fetch()`; they exist for the life of the root program. Standard streams can be redirected to memory files on a non-POSIX `system()` call using command line redirection.

### Hiperspace\* Expanded Storage

Large memory files can be placed in Hiperspace expanded storage to free up some of your home address space for other uses. Hiperspace expanded storage or high performance space is a range of up to 2 gigabytes of contiguous virtual storage space. A program can use this storage as a buffer (1 gigabyte =  $2^{30}$  bytes).

## Additional I/O Features

IBM OS/390 C/C++ provides additional I/O support through the following features:

- User error handling for serious I/O failures (SIGIOERR)

- Improved sequential data access performance through enablement of the DFSMS/MVS support for 31-bit sequential data buffers and sequential data striping on extended format data sets
- Full support of PDS/Es on OS/390 — including support for multiple members opened for write
- Overlapped I/O support under OS/390 (NCP, BUFNO)
- Multibyte character I/O functions
- Fixed-point (packed) decimal data type support in formatted I/O functions
- Support for multiple volume data sets that span more than one volume of DASD or tape
- Support for Generation Data Group I/O

---

## The System Programming C Facility

The System Programming C (SP C) facility allows you to build applications that require no dynamic loading of OS/390 Language Environment libraries. It also allows you to tailor your application to better utilize the low-level services available on your operating system. SP C offers a number of advantages:

- You can develop applications that you can execute in a customized environment rather than with OS/390 Language Environment services. Note that if you do not use OS/390 Language Environment services, only some built-in functions and a limited set of C/C++ runtime library functions are available to you.
- You can substitute the OS/390 C language in place of assembler language when writing system exit routines, by using the interfaces that are provided by SP C.
- SP C lets you develop applications featuring a user-controlled environment, in which an OS/390 C environment is created once and used repeatedly for C function execution from other languages.
- You can utilize co-routines, by using a two-stack model to write application service routines. In this model, the application calls on the service routine to perform services independently of the user. The application is then suspended when control is returned to the user application.

---

## Interaction with Other IBM Products

When you use OS/390 C/C++, you can write programs that utilize the power of other IBM products and subsystems:

- Cross System Product (CSP)

Cross System Product/Application Development (CSP/AD) is an application generator that provides ways to interactively define, test, and generate application programs to improve productivity in application development. Cross System Product/Application Execution (CSP/AE) takes the generated program and executes it in a production environment.

**Note:** You cannot compile CSP applications with the OS/390 C++ compiler. However, your OS/390 C++ program can use interlanguage calls (ILC) to call OS/390 C programs that access CSP.

- Customer Information Control System (CICS)

You can use the CICS/ESA Command-Level Interface to write C/C++ application programs. The CICS Command-Level Interface provides data, job, and task management facilities that are normally provided by the operating system.



**Note:** Code preprocessed with CICS/ESA versions prior to V4 R1 is not supported for OS/390 C++ applications. OS/390 C++ code preprocessed on CICS/ESA V4 R1 cannot run under CICS/ESA V3 R3.

- **DATABASE 2 (DB2)**

DB2 programs manage data that is stored in relational data bases. The IBM DATABASE 2 licensed program runs on OS/390.

You can access the data by using a structured set of queries that are written in Structured Query Language (SQL). The DB2 program uses SQL statements that are embedded in the program. The SQL translator (DB2 preprocessor) translates the embedded SQL into host language statements that perform the requested functions. The OS/390 C/C++ compilers compile the output of the SQL translator. The DB2 program processes a request, and processing returns to the application.

- **Data Window Services (DWS)**

The Data Window Services (DWS) part of the Callable Services Library allows your OS/390 C or OS/390 C++ program to manipulate temporary data objects that are known as TEMPSPACE and VSAM linear data sets.

- **Information Management System (IMS)**

The Information Management System/Enterprise Systems Architecture (IMS/ESA) product provides support for hierarchical databases.

- **Interactive System Productivity Facility (ISPF)**

OS/390 C/C++ provides access to the Interactive System Productivity Facility (ISPF) Dialog Management Services. A dialog is the interaction between a person and a computer. The dialog interface contains display, variable, message, and dialog services as well as other facilities that are used to write interactive applications.

- **Graphical Data Display Manager (GDDM)**

GDDM provides a comprehensive set of functions to display and print applications most effectively:

- A windowing system that the user can tailor to display selected information
- Support for presentation and keyboard interaction
- Comprehensive graphics support
- Fonts — including support for double-byte character set (DBCS)
- Business image support
- Saving and restoring graphics pictures
- Support for many types of display terminals, printers, and plotters

- **Query Management Facility (QMF)**

OS/390 C supports the Query Management Facility (QMF), a query and report writing facility, which allows you to write applications through a callable interface. You can create applications to perform a variety of tasks, such as data entry, query building, administration aids, and report analysis.

---

## Additional Features of OS/390 C/C++

Feature	Description
Multibyte Character Support	OS/390 C/C++ supports multibyte characters for those national languages such as Japanese whose characters cannot be represented by a single byte.

---

Feature	Description
Wide Character Support	Multibyte characters can be normalized by OS/390 C library functions and encoded in units of one length. These normalized characters are called wide characters. Conversions between multibyte and wide characters can be performed by string conversion functions such as <code>wcstombs()</code> , <code>mbstowcs()</code> , <code>wcsrtombs()</code> , and <code>mbstrtowcs()</code> , as well as the family of wide-character I/O functions. Wide-character data can be represented by the <code>wchar_t</code> data type.
Extended Precision Floating-Point Numbers	<p>OS/390 C/C++ provides three S/370 floating-point number data types: single precision (32 bits), declared as <code>float</code>; double precision (64 bits), declared as <code>double</code>; and extended precision (128 bits), declared as <code>long double</code>.</p> <p>Extended precision floating-point numbers give greater accuracy to mathematical calculations.</p> <p>As of Release 6, OS/390 C/C++ also supports IEEE 754 floating-point representation. By default, <code>float</code>, <code>double</code>, and <code>long double</code> values are represented in IBM S/390 floating point format. However, the IEEE 754 floating-point representation is used if you specify the <code>FL0AT(IEEE754)</code> compile option. For details on this support, see the description of the <code>FL0AT</code> option in the <i>OS/390 C/C++ User's Guide</i>.</p>
Command Line Redirection	You can redirect the standard streams <code>stdin</code> , <code>stderr</code> , and <code>stdout</code> from the command line or when calling programs using the <code>system()</code> function.
National Language Support	OS/390 C/C++ provides message text in either American English or Japanese. You can dynamically switch between the two languages.
Locale Definition Support	OS/390 C/C++ provides a locale definition utility that supports the creation of separate files of internationalization data, or locales. Locales can be used at run time to customize the behavior of an application to national language, culture, and coded character set (code page) requirements. Locale-sensitive library functions, such as <code>isdigit()</code> , use this information.
Coded Character Set (Code page) Support	The OS/390 C/C++ compiler can compile C/C++ source written in different EBCDIC code pages. In addition, the <code>iconv</code> utility converts data or source from one code page to another.
Selected Built-in Library Functions	Selected library functions, such as string and character functions, are built into the compiler to improve performance execution. Built-in functions are compiled into the executable, and no calls to the library are generated.
Multitasking Facility (MTF)	Multitasking is a mode of operation where your program performs two or more tasks at the same time. OS/390 C provides a set of library functions that perform multitasking. These functions are known as the Multitasking Facility (MTF). MTF uses the multitasking capabilities of OS/390 to allow a single OS/390 C application program to use more than one processor of a multiprocessing system simultaneously.
Packed Structures and Unions	OS/390 C provides support for packed structures and unions. Structures and unions may be packed to reduce the storage requirements of a OS/390 C program.
Fixed-point (Packed) Decimal Data	<p>OS/390 C supports fixed-point (packed) decimal as a native data type for use in business applications. The packed data type is similar to the COBOL data type <code>COMP-3</code> or the PL/I data type <code>FIXED DEC</code>, with up to 31 digits of precision.</p> <p>The Application Support Class Library provides the Binary Coded Decimal Class for C++ programs.</p>
Long Name Support	For portability, external names can be mixed case and up to 1024 characters in length. For C++, the limit applies to the mangled version of the name.
System Calls	You can call commands or executable modules using the <code>system()</code> function under OS/390, OS/390 UNIX, and TSO. You can also use the <code>system()</code> function to call EXECs on OS/390 and TSO, or Shell scripts using OS/390 UNIX.
Exploitation of ESA	Support for OS/390, IMS/ESA, Hiperspace expanded storage, and CICS/ESA allows you to exploit the features of the ESA.



Feature	Description
Exploitation of hardware	<p>Use the ARCHITECTURE compiler option to select the minimum level of machine architecture on which your program will run. ARCH(3) enables support for IEEE 754 Binary Floating-Point instructions. ARCH(2) instructs the compiler to generate faster instruction sequences available only on newer machines.</p> <p>Use the TUNE compiler option to optimize your application for a selected machine architecture. Tune(3) optimizes your application for the new G5 processor. TUNE(2) optimizes your application for other architectures. For information on which machines and architectures support the above options, refer to the ARCHITECTURE and TUNE compiler information in the <i>OS/390 C/C++ User's Guide</i>.</p>



---

## Part 2. The C and C++ Languages

This part of the Language Reference describes the language elements that are common to C and C++. It also describes the C++ constructs that support object-oriented programming.

### **Chapter 3. Introduction to C and C++**

Provides a brief overview of the features of C and C++, including a description of the C++ constructs that support object-oriented programming.

### **Chapter 4. Lexical Elements of C and C++**

Describes the basic elements of C and C++.

### **Chapter 5. Declarations**

Describes the declarations and declarators. It also describes program linkage, storage classes, fundamental data types, and initialization of the fundamental data types.

### **Chapter 6. Expressions and Operators**

Describes the expressions and standard C and C++ operators used in computation.

### **Chapter 7. Implicit Type Conversions**

Describes the standard conversions performed on the fundamental data types.

### **Chapter 8. Functions**

Describes the form and use of functions, including function declarations and definitions.

### **Chapter 9. Statements**

Describes the statements used to control the execution sequence of programs.

### **Chapter 10. Preprocessor Directives**

Discusses preprocessor directives and OS/390 C/C++ pragmas.



---

## Chapter 3. Introduction to C and C++

This chapter describes the C and C++ programming languages that are implemented by OS/390 C/C++ and shows you how to structure C and C++ source programs. It also briefly summarizes the differences between C and C++, and discusses the principles of object-oriented programming. Specifically, it discusses the following topics:

- “Overview of the C Language”
- “C Source Programs” on page 32
- “C Source Files” on page 33
- “Program Execution” on page 35
- “Scope in C” on page 35
- “Program Linkage” on page 37
- “Storage Duration” on page 39
- “Name Spaces” on page 39
- “Command-Line Arguments” on page 40
- “Command-Line Arguments” on page 40
- “Overview of the C++ Language” on page 42
- “C++ Support for Object-Oriented Programming” on page 42
- “C++ Programs” on page 44
- “Scope in C++” on page 46
- “Simple C++ Input and Output” on page 47
- “Linkage Specifications — Linking to non-C++ Programs” on page 50

---

### Overview of the C Language

C is a programming language that is designed for a wide variety of programming tasks. You can use it for system-level code, text processing, graphics, and many other application areas.

The C language described here is consistent with the Systems Application Architecture Common Programming Interface (also known as the SAA C Level 2 interface). It is also consistent 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) and is technically equivalent to the ANSI\*\* C standard.

C supports several data types, including character, packed decimal, integer, floating-point, and pointer – 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 that is added through its library. This division enables C to be both flexible and efficient. An additional benefit is that the language is consistent across different systems.

The C library contains functions for input and output, mathematics, exception handling, string and character manipulation, dynamic memory management, as

## Overview of the C Language

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.

The *OS/390 C/C++ Run-Time Library Reference* describes all of the C library functions.

---

## C Source Programs

A *C source program* is a collection of one or more directives, declarations, and statements that is contained in one or more source files. The resulting collection of files constitutes a *compilation unit*.

<i>Statements</i>	Specify the action the program performs.
<i>Directives</i>	Instruct the preprocessor to act on the text of the program. Pragma directives affect compiler behavior.
<i>Declarations</i>	Establish names and define characteristics such as scope, data type, and linkage.
<i>Definitions</i>	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 function name, its parameters, and the data type of the value it returns.

A program must contain one, and only one, function called `main()`. The `main()` function is the first function that a program calls when you run the program.

**Note:** This is not the case for C++ programs. If a C++ program instantiates an object in file scope, OS/390 C/C++ executes the constructor for that object first.

By convention, `main()` is the starting point for running a program. It typically calls other functions. A program usually stops running at:

- The end of the `main()` function
- A return statement in the `main()` function
- An `exit` function call.

## CBC3RAAA

This is the source code of 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
                           such as printf used below          */

#include <math.h>           /* Standard math library header that
                           contains macros and function declarations
                           such as 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;
}

```

This source program defines `main()` and declares a reference to the functions `cos` and `printf`. The program defines the global variables `x` and `y`, initializes them, and declares two local variables `z` and `w`.

---

## C Source Files

A *C source file* is a text file that contains all or part of a C source program. It can include any of the functions that the program needs. To create an executable module or program object, you compile the separate source files individually and then link or bind them as one program. With the `#include` directive, you can combine source files into larger source files. The resulting collection of files that are seen by the compiler in a single compilation is known as a compilation unit. A compilation unit does not necessarily constitute the entire program.

A source file contains any combination of directives, declarations, and definitions. You can split items such as function definitions and large data structures between source files, but you cannot split them between compiled files. Before you compile the source file, the preprocessor alters the source file in a predictable way. The preprocessor directives determine what changes the preprocessor makes to the source text. As a result of the preprocessing stage, OS/390 C/C++ completes the preprocessor directives and expands macros. It may create a new source file that contains C statements, processed directives, declarations, and definitions.

## C Source Files

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, if necessary. You can also organize constants and macros into separate files and include them into source files as required.

The following example is a C program in two source files. The `main()` and `max()` functions are in separate files. The program starts by running the `main()` function.

### CBC3RAAB - Source File 1

```
/******
 * 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;
}
```

### CBC3RMAX - Source file 2

```
/******
 * 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. They direct the preprocessor to replace the identifiers `ONE`, `TWO`, and `THREE` with the digits 1, 2, and 3. The directives do not apply to the second source file.

The second source file contains the function definition for `max()`, which the function `main()` calls four times. After you compile the source files, you can bind and run them as a single program.



---

## Program Execution

Every program must have a function called `main()` and usually contains other functions.

The `main()` function is the starting point for running a program. OS/390 C/C++ executes the statements within the `main()` function sequentially. There may be calls to other functions. A program usually stops running at the end of the `main()` function, although it can stop at other points in the program.

You can make your program modular by creating separate functions to perform a specific task or set of tasks. The `main()` function calls these functions to perform the tasks. When your program makes a function call, it executes statements sequentially. It starts with the first statement in the function until it encounters a statement that alters the flow of control. The function returns control to the calling function at the return statement or at the end of the function.

You can declare any function to have parameters. When you call functions, they receive values for their parameters from the arguments that you pass in the calling functions. You can declare parameters for the `main()` function so you can pass values to `main()` from the command line. The command line processor that starts the program can pass such values as described in “The `main()` Function” on page 184.

---

## Scope in C

An identifier becomes *visible* with its declaration. The region where an identifier is visible is the identifier's *scope*. The four kinds of scope are:

- Block
- Function
- File
- Function prototype.

The location of the identifier determines where the identifier is declared. See “Identifiers” on page 56 for more information on identifiers.

### Block Scope

The identifier's declaration is located inside a statement block. A block starts with an opening brace (`{`) and ends with a matching closing brace (`}`). An identifier with block scope is visible from the point where you declare it to the closing brace that ends the block. You can also refer to block scope as *local scope*.

You can nest *block visibility*. A block that is nested inside a block can contain declarations that reuses names declared in the outer block. The new declaration applies to the inner block. OS/390 C/C++ restores the original declaration when program control returns to the outer block. A name from the outer block is visible inside inner blocks that do not redefine the name.

### Function Scope

The only type of identifier with *function scope* is a label name. You implicitly declare a label by its appearance in the program text. A label is visible throughout the function that declares it. A `goto` statement transfers control to the label that is specified on 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 of any block or parameter list. It is visible from the point in the program where you declare it to the end of the source file. If source files are included by `#include` preprocessor directives, those files are considered to be part of the source. The identifier will be visible to all included files that appear after the declaration of the identifier. You can declare the identifier 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 you declare it to the closing parenthesis of the prototype declaration.

### Example of Scope in C

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

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 Information

- “C Source Files” on page 33
- “static Storage Class Specifier” on page 82
- “Chapter 8. Functions” on page 173
- “Labels” on page 197
- “Block” on page 198
- “goto” on page 208
- “Scope in C++” on page 46

---

## Program Linkage

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 you declare it.

A file scope identifier has one of the following kinds of *linkage*:

<i>Internal</i>	Identical identifiers within a single source file refer to the same data object or function.
<i>External</i>	Identical identifiers in separately compiled files refer to the same data object or function.
<i>No linkage</i>	Each identical identifier refers to a unique object.

**Note:** Program linkage is not the same as a *function calling convention*, which you can refer to as linkage. While it relates to program linkage, a calling

## Program Linkage

convention concerns itself with C++ linkage specifications and the use of certain keywords. This section only discusses program linkage.

Use linkage specifications to link to non-C++ declarations. In C, the `#pragma linkage` directive specifies non-C declarations.

See “Linkage Specifications — Linking to non-C++ Programs” on page 50 for more information.

## Internal Linkage

The following kinds of identifiers 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++ inline functions.
- C++ identifiers declared at file scope with the specifier `const` and not explicitly declared `extern`. In C, `const` objects have external linkage by default.

You can define a variable that has `static` storage class within a block or outside a function. If the definition occurs within a block, the variable has internal linkage and is only visible within the block after you can see its declaration. If the definition occurs outside a function, the variable has internal linkage. It is available from the point where it is defined to the end of the current source file.

A class is local to its compilation unit if it has no static members or no inline member functions, and if it has not been used in the declaration of an object, function, or class.

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

The following kinds of identifiers have external linkage:

- Identifiers with file or block scope that have the keyword `extern` in their declarations, and the previously visible declaration is not static.  
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 later declared with the keyword `extern` has internal linkage.
- Function identifiers declared without storage-class specifiers.
- Object identifiers that have file scope declarations without a storage-class specified. OS/390 C/C++ allocates storage for such object identifiers.
- Static class members and no inline member functions.

You can define identifiers that are declared with the keyword `extern` in other compilation units.

## No Linkage

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`

---

## Storage Duration

*Storage duration* determines how long storage for an object exists. An object has either *static* storage duration or *automatic* storage duration, but this depends on its declaration.

*Static storage*

OS/390 allocates this storage at initialization and it remains available until the program ends. Objects have static storage duration if they:

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

*Automatic storage*

OS/390 C/C++ allocates and removes this storage according to the scope of the identifier. Objects have automatic storage duration if they are either one of the following:

- Parameters in a function definition
- Declared at block scope and do not have any storage class specifier
- Declared at block scope, and contain 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. OS/390 C/C++ creates heap objects at run time and allocates storage for them by calling a function such as `malloc()`.

---

## 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. You can use the same identifier 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.

You can redefine identifiers in the same name space but within enclosed program blocks as described in “Scope in C” on page 35.

## Name Spaces

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, unions, and classes must be unique within a single structure, union, or class 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.

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);
}
```

OS/390 C/C++ interprets each occurrence of `student` 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, OS/390 C/C++ passes control to the null statement label. In other contexts, the identifier `student` refers to the structure variable.

## Related Information

- “Scope in C” on page 35
- “Identifiers” on page 56
- “Type Specifiers” on page 85
- “Chapter 6. Expressions and Operators” on page 133

---

## Command-Line Arguments

The maximum allowable length of a command-line argument for OS/390 Language Environment is 64K.

OS/390 C/C++ treats arguments that you enter on the command line differently in different environments. The following lists how `argv` and `argc` are handled.

## Under OS/390 Batch

<code>argc</code>	Returns the number of strings in the argument line
<code>argv[0]</code>	Returns the program name in uppercase
<code>argv[1 to n]</code>	Returns the arguments as you enter them

## Under IMS

<code>argc</code>	Returns 1
<code>argv[0]</code>	Is a null pointer

## Under CICS

<code>argc</code>	Returns 1
<code>argv[0]</code>	Returns the transaction ID

## Under TSO Command

<code>argc</code>	Returns the number of strings in the argument line
<code>argv[0]</code>	Returns the program name in uppercase
<code>argv[1 to n]</code>	Returns the arguments exactly as you enter them

## Under TSO Call

Without the ASIS option:

<code>argc</code>	Returns the number of strings in the argument line
<code>argv</code>	Returns the program name and arguments in lowercase

With the ASIS option:

<code>argc</code>	Returns the number of strings in the argument line
<code>argv[0]</code>	Returns the program name in uppercase
<code>argv[1 to n]</code>	Arguments entered in uppercase are returned in lowercase. Arguments entered in mixed or lowercase are returned as entered.

## Under OS/390 UNIX Shell

<code>argc</code>	Returns the number of strings in the argument line
<code>argv[0]</code>	Returns the program name as you enter it
<code>argv[1 to n]</code>	Returns the arguments exactly as you enter them

The only delimiter for the arguments that are passed to `main()` is white space. OS/390 C/C++ uses commas passed to `main()` by JCL as arguments and not as delimiters.

The following example appends the comma to the 'one' when passed to `main()`.

```
//FUNC EXEC PCGO,GPGM='FUNC',
//      PARM.GO=('one',
//              'two')
```

## Command-Line Arguments

For more information on restrictions of the command-line arguments, refer to the *OS/390 C/C++ User's Guide*.

## Related Information

- “Calling Functions and Passing Arguments” on page 185
- “Parameter Declaration List Syntax” on page 181
- “Type Specifiers” on page 85
- “Identifiers” on page 56
- “Block” on page 198

---

## Overview of the C++ Language

C++ is an object-oriented language based on the C programming language. It can be viewed as a superset of C. Almost all of the features and constructs available in C are also available in C++. However, C++ is more than just an extension of C. Its additional features support the programming style known as *object-oriented programming*. Several features that are already available in C, such as input and output may be implemented differently in C++. In C++ you may use the conventional C input and output routines or you may use object oriented input and output by using the I/O Stream class library.

C++ was developed by Bjarne Stroustrup of AT&T Bell Laboratories. It was originally based on the definition of the C language stated in *The C Programming Language* by Brian W. Kernighan and Dennis M. Ritchie. This C language definition is commonly called *K&R C*. Since then, the International Standards Organization C language definition (referred to here as ANSI/ISO C) has been approved. It specifies many features that K&R left unspecified. Some features of ANSI/ISO C have been incorporated into the current definition of C++, and some parts of the ANSI/ISO C definition have been motivated by C++.

While there is currently no C++ standard comparable to the ANSI/ISO C definition, an ISO committee is working on such a definition. The OS/390 C++ compiler implementation is based on the definition of the language contained in the Draft Proposal International Standard for Information Systems-Programming Language C++ (X3J16/92-00091). The OS/390 C++ compiler also conforms to a subset of the C++ ANSI/ISO (Draft) Standard (X3J16/93-0062).

---

## C++ Support for Object-Oriented Programming

Object-oriented programming is based on the concepts of *data abstraction*, *inheritance*, and *polymorphism*. Unlike procedural programming, it concentrates on the data objects that are involved in a problem and how they are manipulated, not on how something is accomplished. Based on the foundation of data abstraction, object-oriented programming allows you to reuse existing code more efficiently and increase your productivity.

## Data Abstraction

*Data abstraction* provides the foundation for object-oriented programming. In addition to providing fundamental data types, object-oriented programming languages allow you to define your own data types, called *user-defined* or *abstract*



data types. In the C programming language, related data items can be organized into structures. These structures can then be manipulated as units of data. In addition to providing this type of data structure, object-oriented programming languages allow you to implement a set of operations that can be applied to the data elements. The data elements and the set of operations applicable to the data elements together form the abstract data type.

To support data abstraction, a programming language must provide a construct that can be used to encapsulate the data elements and operations that make up an abstract data type. In C++, this construct is called a *class*. An instance of a class is called an *object*. Classes are composed of data elements called *data members* and *member functions* that define the operations that can be carried out on the object. Classes also contain typedefs, enums, and other classes.

## Encapsulation

Another key feature of object-oriented programming is *encapsulation*. Encapsulation means a class can hide the details of:

- The representation of its data members
- The implementation of the operations that can be performed on these data members

Application programs manipulate objects of a class using a clearly defined interface. As long as this interface does not change, you can change the implementation of a class without having to change the application programs that use the class. Encapsulation provides the following advantages:

- Users of a class do not have to deal with unnecessary implementation details.
- Programs are easier to debug and maintain.
- Permitted alterations are clearly specified.

In C++, encapsulation is accomplished by specifying the level of access for each member of a class. Both the data members and member functions of a class can be declared public, protected, or private depending on the kind of access required.

**Note:** C++ encapsulation is *not* a security mechanism. It is possible to circumvent the class access controls that make encapsulation possible. The language is not designed to prevent such misuse.

## Inheritance

*Inheritance* lets you reuse existing code and data structures in new applications. In C++, inheritance is implemented through class derivation. You can extend a library of existing classes by adding data elements and operations to existing classes to form *derived* classes. A derived class has all the members of its parent or *base* class, as well as extensions that can provide additional features. When you create a new derived class, you only have to write the code for the additional features. The existing features of the base class are already available.

A base class can have more than one class derived from it. In addition, a derived class can serve as a base class for other derived classes in a hierarchy. Typically, a derived class is more specialized than its base class.

A derived class can inherit data members and member functions from more than one base class. Inheritance from more than one base class is called *multiple inheritance*.

### Dynamic Binding and Polymorphism

Another key concept that allows you to write generic programs is *dynamic* or *late binding*. Dynamic binding allows a member function call to be resolved at run time, according to the run-time type of an object reference. This permits each user-defined class in an inheritance hierarchy to have a different implementation of a particular function. Application programs can then apply that function to an object without needing to know the specifics of the class to which the object belongs.

In C++, dynamic binding hides the differences between members of a group of classes in an inheritance hierarchy from the application program. At run time, the system determines the specific class of the object and invokes the appropriate function implementation for that class.

Dynamic binding is distinguished from *static* or *compile-time* binding, which involves compile-time member function resolution according to the static type of an object reference.

### Other Features of C++

C++ provides several other powerful extensions to the C programming language. Among these are:

- Constructors and destructors, which are used to initialize and destroy class objects
- Overloaded functions and operators, which let you extend the operations a function or operator can perform on different data types
- Inline functions, which can make programs more efficient
- References, which allow a function to modify its arguments in the calling function
- Template functions and classes, which allow the definition of generic classes and functions
- Object-Oriented Exception handling, which provides transfer of control and recovery from errors and other exceptional circumstances

---

## C++ Programs

C++ programs contain many of the same programming statements and constructs as C programs:

- C++ has many of the same fundamental types (built-in) data types as C, as well as some types that are not built-in to C. For example, packed decimal is supported in C but not C++.
- Like ANSI/ISO C, C++ allows you to declare new names for existing (perhaps complex) types by using the typedef construct. These new type names are not new types.
- In general, the scope and storage class rules for C also apply in C++.
- C and C++ have the same set of arithmetic and logical operators.

A C++ name can identify any of the following:

- An object
- A function
- A set of functions

- An enumerator
- A type
- A class member
- A template
- A value
- A label

A declaration introduces a name into a program and can define an area of storage associated with that name.

An expression can be evaluated and is composed of operations and operands. An expression ending with a ; (semicolon) is called a statement. A statement is the smallest independent computational unit. Functions are composed of groups of one or more statements.

A C++ program is composed of one or more functions. These functions can all reside in a single file or can be placed in different files that are linked to each other. In C++, a program must have one and only one non-member function called `main()`.

The following is a simple C++ program containing declarations, expressions, statements, and two functions:

## CBC3X02D

```

/**
** A simple C++ program containing declarations,
** expressions, statements, and two functions:
**/

#include <math.h>                // contains definition of fabs()
const double multiplier=2.2;    // variable initialization
const double common_ratio=3.1; // variable initialization
double geo_series(double a, double r) // function definition
{
    if (r == 1.0)                // if statement
        return -1.0;            // return statement
    else return -2.0;
};
int main()                      // program execution begins here
{
    double sum;                 // variable declaration
    sum = geo_series(multiplier, common_ratio); // function call
    // ..
    return 0;
}

```

---

## Scope in C++

The area of the code where an identifier is visible is referred to as the *scope* of the identifier. The four kinds of scope are:

- Local
- Function
- File
- Class

The scope of a name is determined by the location of the name's declaration.

A type name first declared in a function return type has file scope. In the following example, Y has file scope:

```
struct Y { int a; int b } foo(int a) { . }
```

A type name first declared in a function argument list has local scope. In the following example, X has local scope:

```
int foo(struct X { int a; int b; } x, int y) {
    .
}
```

A function name that is first declared as a friend of a class is in the first nonclass scope that encloses the class.

If the friend function is a member of another class, it has the scope of that class. The scope of a class name first declared as a friend of a class is the first nonclass enclosing scope. See "Friend Scope" on page 308 for more information.

## Local Scope

A name has *local scope* if it is declared in a block. A name with local scope can be used in that block and in blocks enclosed within that block, but the name must be declared before it is used. When the block is exited, the names declared in the block are no longer available.

Formal argument names for a function have the scope of the outermost block of that function.

If a local variable is a class object with a destructor, the destructor is called when control passes out of the block in which the class object was constructed.

When one block is nested inside another, the variable names from the outer block are usually visible in the nested block. However, if an outer block name is redefined in a nested block, the new declaration is in effect in the inner block. The original declaration is restored when program control returns to the outer block. This is called *block visibility*. See “C++ Scope Resolution Operator (::)” on page 137 for information on scope resolution.

## Function Scope

The only type of identifier with *function scope* is a label name. A label is implicitly declared by its appearance in the program text and is visible throughout the function that declares it.

## File Scope

A name has *file scope* if its declaration appears outside of all blocks and classes. A name with file scope is visible from the point where it is declared to the end of the source file. The name is also made accessible for the initialization of global variables. If a name is declared extern, it is also visible, at link time, in all object files being linked. Global names are names declared with file scope.

## Class Scope

The name of a class member has *class scope* and can only be used in the following cases:

- In a member function of that class
- In a member function of a class derived from that class
- After the . (dot) operator applied to an instance of that class
- After the . (dot) operator applied to an instance of a class derived from that class
- After the -> operator applied to a pointer to an instance of that class
- After the -> (arrow) operator applied to a pointer to an instance of a class derived from that class
- After the :: (scope resolution) operator applied to the name of a class
- After the :: (scope resolution) operator applied to a class derived from that class.

For more information on class scope, see “Scope of Class Names” on page 286.

---

## Simple C++ Input and Output

Like C, the C++ language has no built-in input and output facilities. Instead, input and output facilities for C++ are provided by the I/O Stream Library. For compatibility with C, C++ also supports the standard I/O functions of C. The I/O Stream Library supports a set of I/O operations, written in the C++ language, for the built-in types. You can extend these facilities to provide input and output functions for user-defined data types.

## Input and Output

For a complete description of the I/O Stream Library, see the *OS/390 C/C++ IBM Open Class Library Reference*.

There are four predefined I/O stream objects that you can use to perform standard I/O:

- `cout`
- `cin`
- `cerr`
- `clog`

You can use these in conjunction with the overloaded shift operators, `<<` (insertion or *output*) and `>>` (extraction or *input*). To use streams and operators, you must include the header file `iostream.h`. The following example prints `Hello World!` to standard output:

### CBC3X02F

```
/**
 ** Hello World
 **/

#include <iostream.h>
void main()
{
    cout << "Hello World!" << endl;
}
```

The manipulator `endl` acts as a newline character, causing any output following it to be directed to the next line. Because it also causes any buffered output to be flushed, `endl` is preferred over `\n` to end lines.

## Output (`cout`, `cerr`, and `clog`)

The `cout` stream is associated with standard output. You can use the output operator in conjunction with `cout` to direct a value to standard output. The following example prints out three strings in a row and produces the same result as the previous example, printing `Hello World!` to standard output.

### CBC3X02G

```
/**
 ** Another Hello World, illustrating concatenation with cout
 **/

#include <iostream.h>
void main()
{
    cout << "Hello "
         << "World"
         << "!"
         << endl;
}
```

Output operators are defined to accept arguments of any of the fundamental data types, as well as pointers, references, and array types. You can also overload the output operator to define output for your own classes.

The `cerr` and `clog` streams direct output to standard error. `cerr` provides unbuffered output, while `clog` provides buffered output. The following example checks for a division by zero condition. If one occurs, a message is sent to standard error.

### CBC3X02H

```
/**
 ** Check for a division by zero condition.
 ** If one occurs, a message is sent to standard error.
 **/

#include <iostream.h>

main()
{
    double val1, val2;
    cout << "Divide Two Values" << endl;
    cout << "Enter two numeric values: " << endl;
    cin >> val1 >> val2;
    if (val2 == 0 )
    {
        cerr << "The second value must be non-zero" << endl;
        return;
    }
    cout << "The answer is " << val1 / val2 << endl;
}
```

## Input (cin)

The `cin` class object is associated with standard input. You can use the input operator in conjunction with `cin` to read a value from standard input. By default, white space (including blanks, tabs, and new lines) is disregarded by the input operator. For example:

### CBC3X02I

```
/**
 ** This example illustrates the cin operator
 **/

#include <iostream.h>
main()
{
    double val1, val2;
    cout << "Enter two numeric values:" << endl;
    cin >> val1 >> val2;
    cout << "The first value entered is " << val1
        << " and the second value is "
        << val2 << "." << endl;
}
```

If the values 1.2 and 3.4 are entered through standard input, the above program prints the following to standard output:

```
Enter two numeric values:
1.2
3.4
The first value entered is 1.2 and the second value is 3.4.
```

Any white space entered between the two numeric values is disregarded by the input operator.

## Input and Output

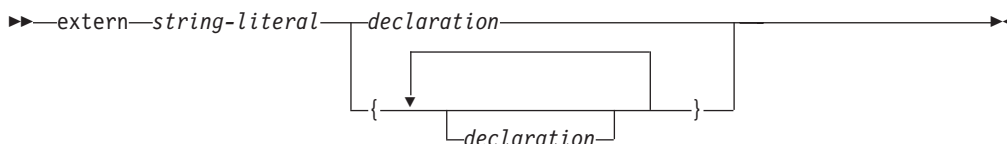
The input operator is defined to accept arguments of any of the fundamental data types, as well as pointers, references and array types. You can also overload the input operator to define input for your own class types.

---

## Linkage Specifications — Linking to non-C++ Programs

You can link C++ object modules to object modules produced using other source languages such as C and Fortran by using a linkage specification.

The syntax is:



The *string-literal* is used to specify the linkage associated with a particular function. For example:

### CBC3X02J

```
/**
 ** This example illustrates linkage specifications
 **/

extern "C" int printf(const char*,...);
void main()
{
    printf("hello\n");
}
```

Here the *string-literal*, "C", tells the C++ compiler that the routine `printf(const char*,...)` is a C library function. Note that string literals used in linkage specifications are not case-sensitive.

Some valid values for *string-literal* are:

"C++" Default  
"C" C type linkage

For more information on string literals, see "String Literals" on page 65. For linkage specification information, see the *OS/390 C/C++ Programming Guide*.

If the value of *string-literal* is not recognized, C type linkage is used.



---

## Chapter 4. Lexical Elements of C and C++

This chapter describes the following lexical elements of C and C++:

- “Tokens”
- “Source Program Character Set”
- “Comments” on page 54
- “Identifiers” on page 56
- “Constants” on page 60

---

### Tokens

During preprocessing and compilation, OS/390 C/C++ treats source code as a sequence of tokens. There are five different types of tokens:

- Identifiers
- Keywords
- Literals
- Operators
- Other separators

You should separate adjacent identifiers, keywords, and literals with white space. You should separate other tokens by white space to make the source code more readable. White space includes blanks, horizontal and vertical tabs, new lines, form feeds, and comments.

---

### Source Program Character Set

The following lists the basic character set that must be available at both compile and run time:

- The uppercase and lowercase letters of the English alphabet

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

- The decimal digits 0 through 9

0 1 2 3 4 5 6 7 8 9

- The following graphic characters:

! " # % & ' ( ) \* + , - . / :  
; < = > ? [ \ ] \_ { } ~

- The caret (^) character in ASCII (bitwise exclusive OR symbol), which may be represented by the equivalent not (~) character on EBCDIC systems
- The split vertical bar (|) character in ASCII, which you may represent by the vertical bar (|) character on EBCDIC systems
- The space character
- The control characters that represent new-line, horizontal tab, vertical tab, and form feed, and end of string (NULL character).

OS/390 C/C++ uses the number sign (#) character for preprocessing only, and treats the \_ (underscore) character as a normal letter.

## Character Set

The execution character set also includes control characters that represent alert, backspace, carriage return, and new-line.

In a source file, a record contains one line of source text; the end of a record indicates the end of a source line.

The encoding of the following characters from the basic character set may vary between the source-code generation environment and the runtime environment:

`! # ' [ ] \ { } ~ ^ |`

The OS/390 C/C++ compiler normalizes the encoding of source files indicated by the `#pragma filetag` directive and the `LOCALE` compile time option to the encoding defined by code page 1047.

The compiler uses the character set that is specified for the `LOCALE` option for any output. This includes:

- Listings that contain identifier names and source code
- String literals and character constants that are emitted in the object code
- Messages generated by the compiler

However, this does not include the source-code annotation in the pseudo-assembly listings.

Depending on the EBCDIC encoding that your installation uses, you can express the `^` and `|` characters as `~` and `|` respectively. This book refers to the `^` and `|` symbols as the *caret* and *vertical bar*, respectively. If you do not specify the `NOLocale` compile-time option, OS/390 C/C++ does not perform normalization. It assumes that the character set encoding is the IBM-1047 code page. In this case, it recognizes both the broken and unbroken vertical bars as the vertical bar. The caret and logical not sign are recognized as the caret. For a detailed description of the `#pragma filetag` directive and the `LOCALE` option, refer to the description of internationalization, locales, and character sets in the *OS/390 C/C++ Programming Guide*.

The compiler recognizes and supports the additional characters (the extended character set) which you can meaningfully use in string literals and character constants. The support for extended characters includes the multibyte character sets.

OS/390 systems represent multibyte characters by using Shiftout `<S0>` and Shiftin `<SI>` pairs. Strings are of the form:

`<S0> x y z <SI>`

Or they can be mixed:

`<S0> x <SI> y z`  
`x <S0> y <SI> z`

In the above, two bytes represent each character between the `<S0>` and `<SI>` pairs. OS/390 C/C++ restricts multibyte characters to character constants, string constants, and comments.

Refer to the *OS/390 C/C++ Run-Time Library Reference* for a discussion on strings that are passed to library routines, and to "Character Constants" on page 64 of this book for information on character constants. If you specify a lowercase `a` as part of an identifier name, you cannot substitute an uppercase `A` in its place. You must use the lowercase letter.

## Trigraph Sequences

Some characters from the C character set are not available in all environments. You can enter these characters into a C source program by using a sequence of three characters that are called a *trigraph*. The trigraph sequences are:

??=	#	number sign
??(	[	left bracket
??)	]	right bracket
??<	{	left brace
??>	}	right brace
??/	\	backslash
??'	^	caret
??!		vertical bar
??-	~	tilde

The preprocessor replaces trigraph sequences with the corresponding single-character representation by using the code page that is indicated by the `LOCALE` option. If you do not specify the `LOCALE` option, the preprocessor uses code page 1047.

At compile time, the compiler translates the trigraphs found in string literals and character constants into the appropriate characters they represent. These characters are in the coded character set you select by using the `LOCALE` compiler option.

**Note:** The OS/390 C/C++ compiler will compile source files that were edited using different encoding of character sets. However, they might not compile cleanly. OS/390 C/C++ does not compile source files that you edit with the following:

- A character set that does not support all the characters that are specified above, even if the compiler can access those characters by a trigraph.
- A character set for which no one-to-one mapping exists between it and the character set above.

**Note:** The exclamation mark (!) is a variant character. Its recognition depends on whether or not the `LOCALE` option is active. For more information on variant characters, refer to the *OS/390 C/C++ Programming Guide*.

### Example

```
some_array??(i??) = n;
```

Represents:

```
some_array[i] = n;
```

## Digraph Sequences

You can represent unavailable characters in an C++ source program by using a combination of two keystrokes that are called a *digraph sequence*. The preprocessor reads digraphs as tokens during the preprocessor phase.

**Note:** OS/390 C/C++ supports digraphs for C++ only.

## Character Set

The digraph sequences are:

%:	#	number sign
<:	[	left bracket
:>	]	right bracket
<%	{	left brace
%>	}	right brace
%%:	##	preprocessor macro concatenation operator

You can create digraphs by using macro concatenation. OS/390 C/C++ does not replace digraphs in string literals or in character literals. For example:

```
char *s = "<%%>"; // stays "<%%>"

switch (c)
{
    case '<%': { /* ... */ } // stays '<%'
    case '%>': { /* ... */ } // stays '%>'
}
```

The NODIGRAPH option disables processing of digraphs. The NODIGRAPH option is on by default.

The DIGRAPH option is described in the *OS/390 C/C++ User's Guide*.

## Additional Keywords

If you use the digraph option, you can represent unavailable characters in a C++ source program by using the following keywords:

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

These keywords are only reserved in C++ programs that are compiled with the DIGRAPH option. *OS/390 C/C++ User's Guide* describes the DIGRAPH option.

---

## Comments

Comments begin with the `/*` characters. They end with the `*/` characters, and can span more than one line. You can put comments anywhere the language allows white space.

The preprocessor replaces comments during preprocessing by a single space character.

Multibyte characters can also be included within a comment.

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

In the following program, line 6 is a comment:

```

1  #include <stdio.h>
2
3  int main(void)
4  {
5      printf("This program has a comment.\n");
6      /* printf("This is a comment line and will not print.\n"); */
7      return 0;
8  }
```

Because the comment on line 6 is equivalent to a space, the output of this program is:

This program has a comment.

Because the comment delimiters are inside a string literal, line 5 in the following program is not a comment.

```

1  #include <stdio.h>
2
3  int main(void)
4  {
5      printf("This program does not have \
6      /* NOT A COMMENT */ a comment.\n");
7      return 0;
8  }
```

The output of the program is:

This program does not have `/* NOT A COMMENT */` a comment.

You cannot nest comments. Each comment ends at the first occurrence of `*/`.

The following example highlights the comments:

```

1  /* A program with nested comments. */
2
3  #include <stdio.h>
4
5  int main(void)
6  {
7      test_function();
8      return 0;
9  }
10
11 int test_function(void)
12 {
13     int number;
14     char letter;
15     /*
16     number = 55;
17     letter = 'A';
18     /* number = 44; */
19     */
20     return 999;
21 }
```



## Special Characters in Identifiers

The first character in an identifier must be a letter or the underscore (`_`) character. The compiler reserves identifiers beginning with an underscore, however, for identifiers at file scope.

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

Avoid creating identifiers that begin with an underscore for function names and variable names.

At the extended and compatible language levels, C++ identifiers can contain the `$` character. At the ANSI language level, identifiers can begin with the underscore but not with a `$` (dollar sign).

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.

## Case Sensitivity in Identifiers

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

**Note:** If you do not use the OS/390 C compiler long name support, you may receive an error message if you use either `STOCKONHOLD` and `stockonhold` as external identifiers. For more information on long name support, see “longname” on page 261. For more information on the binder and the prelinker, see the *OS/390 C/C++ User's Guide*. Also see “OS/390 C/C++ External Name Mapping” on page 58 and “OS/390 Long Name Support” on page 59.

## Significant Characters in Identifiers

In general, OS/390 C/C++ truncates external and internal identifiers after 1024 characters. However, the C compiler truncates external identifiers after 8 characters if the `NOLONGNAME` compile-time option is in effect. Also, the C++ compiler truncates external identifiers that do not have C++ linkage after 8 characters if the `NOLONGNAME` compile-time option is in effect.

## Keywords

*Keywords* are identifiers that are reserved by the language for special use. Although you can use them for preprocessor macro names, it is poor programming style. Only the exact spelling of keywords is reserved. For example, `auto` is reserved, but `AUTO` is not. The following table lists the keywords common to both the C and C++ languages. The ANSI/ISO C language definition includes these keywords:

## Identifiers

*Table 3. Keywords Common to C and C++*

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

The C++ language also reserves the following keywords:

*Table 4. C++ Keywords*

asm	_Export	private	throw
__cdecl	friend	protected	try
catch	inline	public	virtual
class	new	template	wchar_t
delete	operator	this	

Future versions of the C++ compiler may reserve the following keywords, so you should avoid using them in your applications:

*Table 5. C++ Keywords (Future)*

bool	false	reinterpret_cast	typeid
const_cast	mutable	static_cast	typename
dynamic_cast	namespace	true	using
explicit			

The C compiler reserves the `__Packed` keyword.

## OS/390 C/C++ External Name Mapping

OS/390 C/C++ maps the names of variables or functions that have external linkages to names that are used in the object module. When you compile an OS/390 C/C++ program, refer to the following guidance for using names of variables or functions with external linkage:

- Do not use names of the library functions for user-defined functions.
- Some functions in the C library and C runtime environment begin with two underscores (`_ _`). Do not use an underscore as the first letter of an identifier.
- The compiler maps each underscore to an at sign (`@`) for external names without C++ linkage, except when you compile a program with the `LONGNAME` compile-time option. In that case, the underscore remains as an underscore.
- IBM-provided functions have names that begin with `IBM`, `CEE`, and `PLI`. Avoid using these names as the OS/390 C/C++ compiler changes these names to prevent conflicts between runtime functions and user-defined names. It changes all static or extern variable names that begin with `IBM`, `CEE`, and `PLI` in your source program to `IB$`, `CE$`, and `PL$`, respectively, in the object module. If you are using interlanguage calls, avoid using these prefixes. The compiler of the calling or called language may or may not change these prefixes in the same manner as the OS/390 C/C++ compiler does. All of this is completely integrated into the OS/390 C/C++ compiler, Debug Tool, and LE/370.

To call an external program or access an external variable that begins with `IBM`, `CEE`, and `PLI`, use the `#pragma map` preprocessor directive. The following is an example of `#pragma map` that forces an external name to be `IBMENTRY`.

```
#pragma map(ibmentry,"IBMENTRY")
```



For more information on the `#pragma map` directive, see “map” on page 262.

## OS/390 Long Name Support

If you do not specify the `LONGNAME` option when you compile your code with the C compiler, the compiler maps an underscore to an at sign. It also truncates external names to 8 characters and changes them to uppercase. The C++ compiler makes the same changes to external identifiers that do not have C++ linkage if you do not specify the `LONGNAME` option.

For example, consider if you compile the following C program and do not specify the `LONGNAME` option:

```
int test_name[4] = { 4, 8, 9, 10 };
int test_namesum;

int main(void) {
    int i;
    test_namesum = 0;

    for (i = 0; i < 4; i++)
        test_namesum += test_name[i];
    printf("sum is %d\n", test_namesum);
}
```

In the above example, the C compiler displays the following message:

```
ERROR CBC3244 ./sum.c:2 External name TEST_NAM cannot be redefined.
```

The compiler changes the external names `test_namesum` and `test_name` to uppercase and truncates them to 8 characters. If you specify the `CHECKOUT` compile-time option, the compiler will generate two informational messages to this effect. Because the truncated names are now the same, the compiler produces an error message and terminates the compilation.

If you compile the previous program with the `LONGNAME` compile-time option, the compiler does not produce any warning or error messages. However, if you specify the `LONGNAME` option, you must bind your program with the binder to produce a program object in a PDSE. Otherwise you must use the prelinker.

The `LONGNAME` compile-time option supports mixed case, external names of up to 1024 characters for OS/390 C/C++ programs.

Object modules that are produced by compiling with `LONGNAME` have external names that are mixed case and up to 1024 characters long. Object modules that are produced by compiling with `NOLONGNAME` have uppercase external names that are limited to a length of 8 characters.

To use external C names that are longer than 8 characters or external C++ names without C++ linkage that are longer than 8 characters, you can, in your source code:

- Use the `#pragma map` directive to map long external names in the source code to 8 or less characters in the object module.
 

```
#pragma map(verylongname, "sname")
```
- Use the long name support that is provided by the compile-time option `LONGNAME`. To use the long name support, you must do the following:
  - Use the `LONGNAME` compile-time option when compiling your program.

## Identifiers

- Use the binder to produce a program object in a PDSE, or use the prelinker. For more information on the binder and on the prelinker, see the *OS/390 C/C++ User's Guide*.

---

## Constants

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

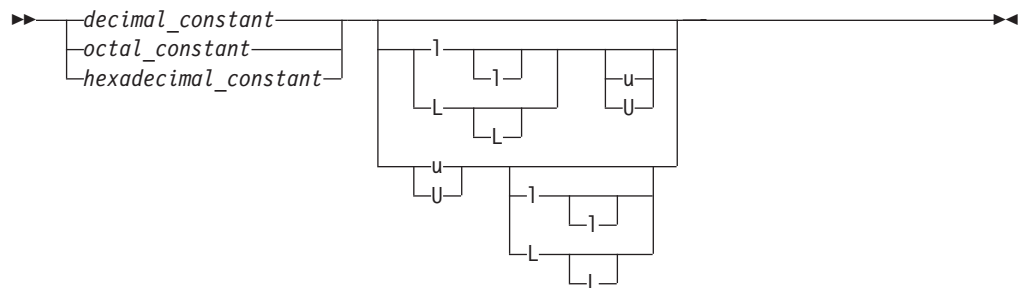
C/C++ contains the following types of constants (also called *literals*):

- Integer
- Floating-Point
- Fixed-Point Decimal Constants (C Only)
- Character
- String
- Enumeration

“Enumerations” on page 90 discusses enumeration constants, which belong to the lexical class of identifiers. For more information on data types, see “Type Specifiers” on page 85.

## Integer Constants

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



Note that the suffixes in the above syntax diagram are not case-sensitive; that is, `l` and `L` are the same to the compiler.

An integer constant without a suffix cannot have a value greater than `ULONG_MAX`. An integer constant with a suffix that contains `LL` cannot have a value greater than `ULLONG_MAX`. In these cases, the compiler will issue an *out of range* error message. For information on the `ULONG_MAX` and the `ULLONG_MAX` macros, see the *OS/390 C/C++ Run-Time Library Reference*.

## Data Types for Integer Constants

OS/390 C/C++ determines the data type of an integer constant by the form, value, and suffix of the constant. The following lists the integer constants and shows the possible data types for each constant. The compiler uses the smallest data type that can represent the constant value to store the constant.

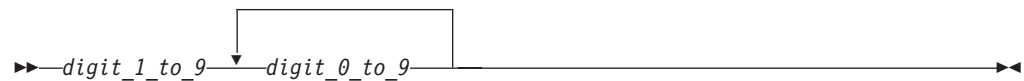
Table 6. Data Types for Integer Constants

Constant	Data Type
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 u or U	unsigned int, unsigned long int
suffixed by l or L	long int, unsigned long int
suffixed by both u or U, and l or L	unsigned long int
suffixed by ll or LL	long long int, unsigned long long int
suffixed by both u or U, and ll or LL	unsigned long long int

A plus (+) or minus (-) symbol can precede an integer constant. OS/390 C/C++ treats it as a unary operator rather than as part of the constant value.

## Decimal Constants

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



OS/390 C/C++ interprets integer constants that begin with the digit 0 as an octal constant, rather than as a decimal constant.

The following are examples of decimal constants:

$$\begin{array}{r} 485976 \\ -433132211 \\ +20 \\ 5 \end{array}$$

## Hexadecimal Constants

A *hexadecimal constant* begins with the 0 digit that is followed by either an x or X. This is 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:

## Constants

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

### Octal Constants

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



The following are examples of octal constants:

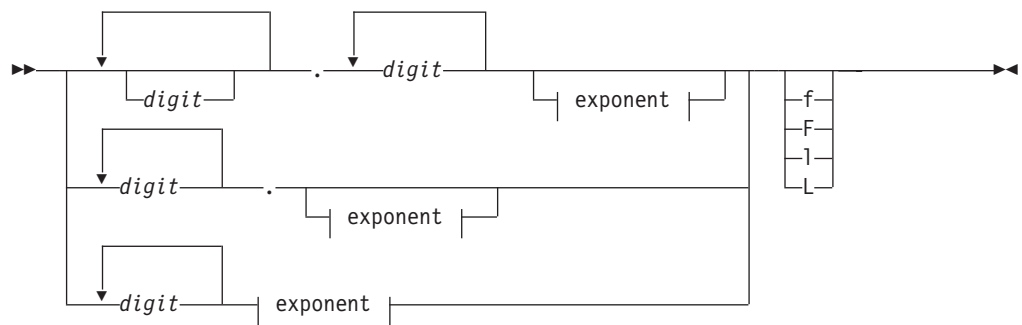
0  
0125  
034673  
03245

### Floating-Point Constants

A *floating-point constant* consists of following parts:

- 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.



#### Exponent:



The representation of a floating-point number on a system is unspecified. If a floating-point constant is too large or too small, the result is undefined by the language.

The suffix `f` or `F` indicates a type of float, and the suffix `l` or `L` indicates a type of long double. If you do not specify a suffix, the floating-point constant has a type double.

A plus (+) or minus (-) symbol can precede a floating-point constant. The compiler treats it as a unary operator rather than as part of the constant value.

The following are examples of floating-point constants:

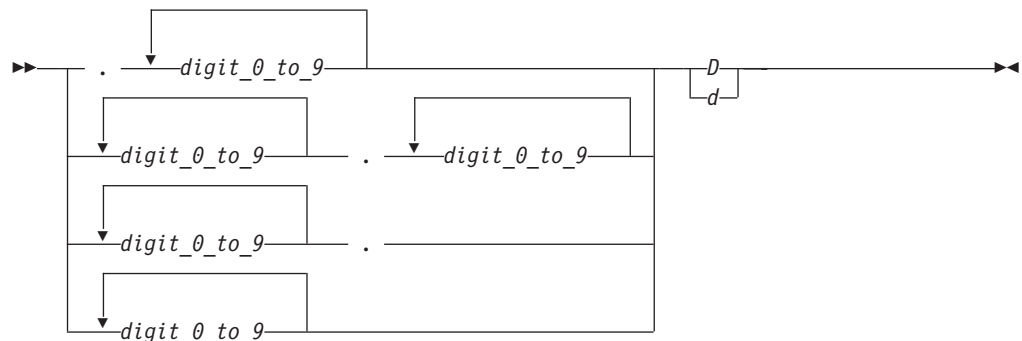
Floating-Point Constant	Value
5.3876e4	53,876
4e-11	0.000000000004
1e+5	100000
7.321E-3	0.007321
3.2E+4	32000
0.5e-6	0.0000005
0.45	0.45
6.e10	60000000000

## Fixed-Point Decimal Constants (C Only)

*Fixed-point decimal constants* are an IBM extension to ANSI/ISO C. This type is available when you specify the `LANGlvl(EXTENDED)` compile-time option.

A fixed-point decimal constant has a numeric part and a suffix that specifies its type. The numeric part can include a digit sequence that represents the whole-number part, followed by a decimal point (`.`), followed by a digit sequence that represents the fraction part. Either the integral part or the fractional part, or both must be present.

A fixed-point constant has the form:



## Constants

A fixed-point constant has two attributes:

Number of digits (size)

Number of decimal places (precision).

The suffix D or d indicates a fixed-point constant.

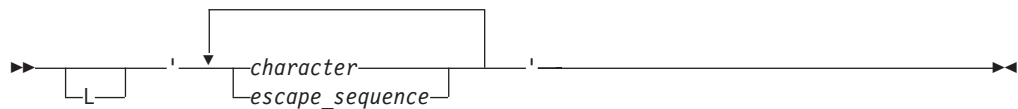
The following are examples of fixed-point decimal constants:

Fixed-Point Constant	(size, precision)
1234567890123456D	(16, 0)
12345678.12345678D	(16, 8)
12345678.d	( 8, 0)
.1234567890d	(10, 10)
12345.99d	( 7, 2)
000123.990d	( 9, 3)
0.00D	( 3, 2)

For more information on fixed-point decimal data types, see the *OS/390 C/C++ Programming Guide*.

## Character Constants

A *character constant* contains a sequence of characters or escape sequences that are 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 that contains a single character is the numeric representation of the character in the character set that is used at compile 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. If the character constant contains more than one character, the last 4 bytes represent the character constant. In C++, a character constant can contain only one character.

In C, a character constant has type `int`. In C++, a character constant has type `char`.

A wide character constant has type `wchar_t`, and is used to represent multibyte characters. Multibyte characters represent characters that use more than one byte for their encoding. Each multibyte character requires up to 4 bytes for its encoding.

You can represent the double quotation mark symbol by itself. You must, however, use the backslash symbol that is followed by a single quotation mark symbol (`\'`) as an escape sequence to represent the single quotation mark symbol.

You can represent the new-line character by the `\n` new-line escape sequence. You can represent the backslash character by the `\\` backslash escape sequence.

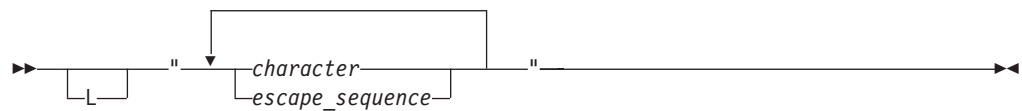
The following are examples of character constants:

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

## String Literals

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

The maximum size of a string literal on OS/390 C/C++ is 32,765 bytes.



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

OS/390 C/C++ appends a null (`'\0'`) character 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.

The compiler retains multiple spaces that are contained within a string constant.

To continue a string on the next line, you can use two or more consecutive strings. The compiler concatenates adjacent string literals 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" */
```

Another way to continue a string is to use the line continuation sequence (`\` symbol that is 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 \ /* continuation symbol */
invisible to the compiler.";      /* compilation error. */
```

Characters in concatenated strings remain distinct. For example, the string `"\xab"` occupies 2 bytes of storage. The first byte contains the value `X'ab'`, and the second byte contains the value `X'00'` which is the trailing null character. The string `"\xa\xb"` occupies 3 bytes of storage that contains the values `X'0a'`, `X'0b'`, and `X'00'`.

## Constants

Following any concatenation, OS/390 C/C++ appends a `'\0'` of type `char` at the end of each string. C++ library functions find the end of a string by scanning for this value. For a wide-character string literal, OS/390 C/C++ appends a `'\0'` of type `wchar_t`. 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 string constant has type *array of wchar\_t* and static storage duration.

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

Use the `#pragma strings` directive to change the default storage for string literals. “strings” on page 273 describes the `#pragma strings` directive.

OS/390 C/C++ stores string literals in static storage which can be modified like any other storage location. C/C++ has the concept of *readonly* and *writable* strings. This deals with how C/C++ stores multiple occurrences of strings, rather than whether or not you can modify the strings.

When a string literal appears more than once in the program source, how that string is stored depends on whether strings are *readonly* or *writable*. If strings are *readonly*, then OS/390 C/C++ allocates only one location for that string. All occurrences will refer to that one location. If strings are *writable*, then each occurrence of the string will have a separate, distinct storage location.

By default, the C compiler will consider strings to be *writable*. Note that for *readonly* `#pragma strings`, the compiler will put literal strings in an area of storage that is potentially read only. For *writable* `#pragma strings`, it will put them in an area of storage that is always modifiable.

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:

### **CBC3X02K**

```
/**
 ** This example illustrates escape sequences in string literals
 **/

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

This program produces the following output:



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

## Escape Sequences

You can represent any member of the execution character set by an *escape sequence*. You can use escape sequences to put unprintable 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. You can use it 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
\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

The value of an escape sequence represents the code point of the code page you use at run time. OS/390 C/C++ translates escape sequences during preprocessing. For example, on a system that uses the ASCII character codes, the value of the escape sequence \x56 is the letter V. On a system that uses EBCDIC character codes, the value of the escape sequence \xE5 is the letter V.

Use escape sequences only in character constants or in string literals. OS/390 C/C++ issues a message only if it does not recognize an escape sequence.

## Constants

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. For example:

```
cout << "The escape sequence \\n." << endl;
```

This statement results in the following output:

```
The escape sequence \n.
```

The following program prints the character 'a' four times to standard output, and then prints a new line:

### **CBC3X02L**

```
/** CBC3X02L
** This example illustrates escape sequences
**/

#include <iostream.h>
void main()
{
    char a,b,c,d,e;
    a='a';
    b=129;    // EBCDIC integer value
    c='\201'; // EBCDIC octal value
    d='\x81'; // EBCDIC hexadecimal value
    e='\n';
    cout << a << b << c << d << e;
}
```

---

## Chapter 5. Declarations

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, OS/390 C/C++ does not allocate storage. This chapter discusses the following topics on declarations:

- “Declarations Overview”
- “Block Scope Data Declarations” on page 70
- “File Scope Data Declarations” on page 71
- “Objects” on page 72
- “Storage Class Specifiers” on page 73
- “typedef” on page 84
- “Type Specifiers” on page 85
- “Declarators” on page 119
- “Initializers” on page 127
- “C/C++ Data Mapping” on page 129
- “C++ Function Specifiers” on page 129
- “C++ References” on page 129

---

### Declarations Overview

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. For a complete description of scope, see “Scope in C” on page 35.
- Linkage, which describes the association between two identical identifiers. See “Program Linkage” on page 37 for more information.
- Storage duration, which describes when the system allocates and frees storage for a data object. See “Storage Duration” on page 39 for more information.
- Type, which describes the kind of data the object is to represent.

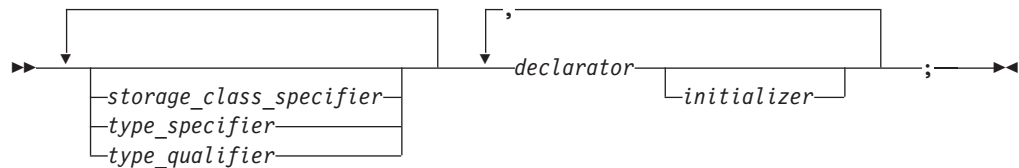
The lexical order of elements when you declare a data object is as follows:

- Storage duration and linkage specification, that are described in “Storage Class Specifiers” on page 73
- Type specification, described in “Type Specifiers” on page 85
- Declarators, which introduce identifiers and make use of type qualifiers and storage qualifiers, described in “Declarators” on page 119
- Initializers, which initialize storage with initial values, described in “Initializers” on page 127.

“Chapter 8. Functions” on page 173 describes function declarations.

All data declarations have the form:

## Declarations Overview



### C++ Notes:

1. One of the fundamental differences between C++ and C is the placement of variable declarations. Although you can declare variables in the same way, in C++, you can put variable declarations anywhere in the program. In C, declarations must come before any statements in a block.

In the following C++ example, the variable `d` is declared in the middle of the `main()` function:

```
#include <iostream.h>
void main()
{
    int a, b;
    cout << "Please enter two integers" << endl;
    cin >> a >> b;
    int d = a + b;
    cout << "Here is the sum of your two integers:" << d << endl;
}
```

2. A given function, object, or type can have only one definition. It can have more than one declaration as long as all of the declarations match. If you never call a function and never take its address, then you do not have to define it. If you declare an object, but never use it, or use it only as the operand of `sizeof`, you do not have to define it. You can declare a given class or enumerator more than once.

The following table shows examples of declarations and definitions. The identifiers that are 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 that are declared in the second column allocate storage; they are both declarations and definitions.

Table 7. Examples of Declarations and Definitions

Declarations	Declarations and Definitions
<code>extern double pi;</code>	<code>double pi = 3.14159265;</code>
<code>float square(float x);</code>	<code>float square(float x) { return x*x; }</code>
<code>struct payroll;</code>	<code>struct payroll {     char *name;     float salary; } employee;</code>

## Block Scope Data Declarations

You can only put a *block scope data declaration* 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`
- `register`
- `static`
- `extern`

If you do not specify a storage class specifier in a block-scope data declaration, OS/390 C/C++ uses the default storage class specifier `auto`. 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 that is 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 73 for details on the different storage classes.

## Storage

The duration and type of storage vary 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 Information

- “Declarators” on page 119
- “Storage Class Specifiers” on page 73
- “auto Storage Class Specifier” on page 73
- “extern Storage Class Specifier” on page 75
- “register Storage Class Specifier” on page 81
- “static Storage Class Specifier” on page 82
- “Initializers” on page 127
- “Type Specifiers” on page 85

---

## 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 OS/390 C/C++ to allocate storage 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.

## File Scope Data Declarations

The only storage class specifiers you can put in a file scope data declaration are `static` and `extern`. All file scope variables that are defined with `static` storage class have internal linkage; other file scope variables have external linkage. If you specify the storage class, you can omit the type specifier. If you omit the type specifier, all variables that are 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, a constant expression must describe the initializer. Otherwise, it must reduce to the address of a previously declared variable at file scope, possibly added to 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. OS/390 C/C++ allocates storage at run time which it frees when the program stops running.

## Related Information

- “`extern` Storage Class Specifier” on page 75
- “`static` Storage Class Specifier” on page 82
- “Initializers” on page 127
- “Type Specifiers” on page 85

---

## Objects

An *object* is a region of storage that contains a value or group of values. You can access each value by using its identifier or by using a complex expression that refers to the object. In addition, each object has a unique *data type*. OS/390 C/C++ establishes both the identifier and data type of an object in the object *declaration*.

The data type of an object determines the initial storage allocation for that object and the interpretation of the values during subsequent access. You can also use it in any type-checking operations.

C++ has built-in, or *standard*, data types, and user-defined data types. Standard data types include signed and unsigned integers, floating-point numbers, and characters. User-defined types include enumerations, structures, unions, and classes.

In C++ code, you reference objects by variables or references. A variable also represents the location in storage that contains the value of an object.

You commonly refer to an instance of a class type as a *class object*. The individual data members of an instantiated class object are also called objects. The set of all member objects comprises a class object.

## Storage Class Specifiers

The storage class specifier you use within the declaration determines whether:

- The object has internal, external, or no linkage.
- The object is 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 is referenced throughout a program or only within the function, block, or source file where you have defined the variable.
- The storage duration for the object is static or automatic. For static, OS/390 C/C++ maintains storage throughout the program run time. For automatic, OS/390 C/C++ maintains storage 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 duration. 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`.

This section describes the following storage class specifiers:

- `auto`
- `extern`
- `register`
- `static`

### auto Storage Class Specifier

The `auto` storage class specifier lets you define a variable with automatic storage. OS/390 C/C++ restricts its use and storage to the current or contained block. The storage class keyword `auto` is optional in a data declaration. You cannot use it in a parameter declaration. You must declare a variable that has the `auto` storage class specifier within a block. You cannot use it for file scope declarations.

Automatic variables require storage only while the function in which they are declared is active. Consequently, defining variables with the `auto` storage class can decrease the amount of memory that is 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. A change to an `auto` variable in one function never affects another function (unless you pass it as an argument).

#### 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

## Storage Class Specifiers

expression that represents the initial value can be any valid C or C++ expression. For structure and union members, the initial value must be a valid constant expression if you use an initializer list. Each time an object's definition (auto or register) is encountered during program execution, its initialization, if any, is done.

**Note:** If you use the goto statement to jump into the middle of a block, automatic variables defined before the label that is jumped to 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 that are defined in that block is made available. When the block is exited, the objects are no longer available for use.

If you define an auto object within a function that you invoke recursively, OS/390 C/C++ allocates memory for the object at each invocation of the block.

### Examples of auto Storage Class

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 that is created by the second definition is available. During the rest of the program, only the auto\_var that is created by the first definition is available.

#### CBC3RAAF:

```
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.

#### CBC3RAAG:

```

/*****
** 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 you run the program, interaction with the program could produce:

**Output**           Enter letters:

**Input**            zyfab

**Output**           The sorted string is: abfyz

#### Related Information

- “Block Scope Data Declarations” on page 70
- “register Storage Class Specifier” on page 81
- “Address (&)” on page 144
- “Function Declarator” on page 180

## extern Storage Class Specifier

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

## Storage Class Specifiers

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.

You can distinguish an extern declaration from an extern definition by the presence of the keyword `extern` and the absence of an initial value. If the keyword `extern` is absent or if there is an initial value, the declaration is also a definition; otherwise, it is just a declaration. An extern definition can appear only at file scope.

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 describes the variable that is externally defined.

If a declaration for an identifier already exists at file scope, any extern declaration of the same identifier that is 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. It is an error to include a declaration for the same function with the storage class specifier `static` before the declaration with no storage class specifier because of the incompatible declarations. Including the extern storage class specifier on the original declaration is valid, and the function has internal linkage.

In OS/390 C++, you can declare functions with the following:

Linkage	By specifying
<b>C</b>	<code>extern "C"</code>
<b>C++</b>	<code>extern "C++"</code>
<b>OS</b>	<code>extern "OS"</code>
<b>PLI</b>	<code>extern "PLI"</code>
<b>builtin</b>	<code>extern "builtin"</code>
<b>COBOL</b>	<code>extern "COBOL"</code>
<b>FORTRAN</b>	<code>extern "FORTRAN"</code>

There are some limitations to using `extern` to specify non-C++ linkage for a function. While the C++ language supports overloading, other languages do not. The implications of this are:

- You cannot overload a function that has non-C++ linkage:

```
extern "FORTRAN">{int func(int);}
extern "FORTRAN">{int func(int,int);} // not allowed-compiler
// will issue an error message
```
- You cannot declare a function with a linkage specification if you have already used the same function name in a declaration without a linkage specification:

```
int func(int);
extern "FORTRAN">{int func(int,int);} // not allowed-compiler
// will issue an error message
```

- You can overload a function as long as it has C++ (default) linkage. Therefore, OS/390 C/C++ allows the following series of statements:

```
extern "FORTRAN">{int func(int,int);}
int func(int);           // function with C++ linkage
int func(int,int);       // overloaded function with C++ linkage
```

- You cannot redefine a function that has a linkage specification:

```
extern func(int);
extern "FORTRAN">{int func(int,int);} // not allowed-compiler
// will issue an error message
```

For more information, see "Using Linkage Specifications in C++" in the *OS/390 C/C++ Programming Guide*, or refer to *OS/390 Language Environment Writing Interlanguage Applications*.

The following fragments illustrate the use of `extern "C"`:

```
extern "C" int cf();           //declare function cf to have C linkage

extern "C" int (*c_fp)();     //declare a pointer to a function,
                             // called c_fp, which has C linkage

extern "C" {
    typedef void(*cfp_T)(); //create a type pointer to function with C
                             // linkage
    void cfn();              //create a function with C linkage
    void (*cfp)();           //create a pointer to a function, with C
                             // linkage
}
```

Linkage compatibility affects all C library functions that accept a user function pointer as a parameter. Use the `extern "C"` linkage specification to ensure that the declared linkages are the same. An example of these library functions is `qsort()`; refer to the *OS/390 C/C++ Run-Time Library Reference* for more information.

The following example fragment uses `extern "C"` with `qsort()`.

```
#include <stdlib.h>

// function to compare table elements
extern "C" int TableCmp(const void *, const void *); // C linkage
extern void * GenTable();                          // C++ linkage

void main() {
    void *table;

    table = GenTable();           // generate table
    qsort(table, 100, 15, TableCmp); // sort table, using TableCmp
                                     // and C library routine qsort();
}
```

**C++ Note:** In C++, an `extern` declaration cannot appear in class scope.

## 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 do either of the following:

- Appear as part of the definition and the initial value must be described by a constant expression.
- Reduce to the address of a previously declared object with static storage duration. You can modify this object by adding or subtracting an integral constant expression.

## Storage Class Specifiers

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

extern objects have static storage duration. OS/390 C/C++ allocates memory for extern objects before the main function begins running. When the program finishes running, OS/390 C/C++ frees the storage.

### Controlling External Static

Certain program variables with the extern storage class may be constant and never be updated. If this is the case, it is not necessary to have a copy of these variables made for every user of the program. In addition, there may be a need to share constant program variables between C and another language.

### Examples of extern Storage Class

The following program fragment shows how to force an external program variable to be part of a program that includes executable code and constant data. It uses the `#pragma variable(varname, NORENT)` directive:

```
#pragma variable(rates, NORENT)
extern float rates[5] = { 3.2, 83.3, 13.4, 3.6, 5.0 };

extern float totals[5];

int main(void) {
    :
}
}
```

In this example, you compile the source file with the RENT option. The executable code includes the variable `rates` as you specify the `#pragma variable(rates, NORENT)`. The writable static includes the variable `totals`. Each user has a personal copy of the array `totals`, and all users of the program share the array `rates`. This sharing may yield a performance and storage benefit.

The `#pragma variable(varname, NORENT)` does not apply to, and has no effect on, program variables with the static storage class. OS/390 C/C++ always includes program variables with the static storage class with the writable static. An informational message appears if you write to a nonreentrant variable when you specify the C CHECKOUT compile-time option.

When you specify `#pragma variable(varname, NORENT)` for a variable, ensure that your program never writes to this variable. Program exceptions or unpredictable program behavior may result should this be the case. In addition, you must include `#pragma variable(varname, NORENT)` in every source file where you reference or define the variable.

For more information on the RENT and NORENT compile-time options, refer to the *OS/390 C/C++ User's Guide*.

The following program shows the linkage of extern objects and functions. It declares the extern object `total` 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 example defines extern function `tally` 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.

#### CBC3RAH1 (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
6
7  #include <stdio.h>
8
9  int main(void)
10 {
11     void tally(void);      /* begin main          */
12     extern float total;    /* declaration of function tally */
13                             /* first declaration of total   */
14
15     printf("Enter the purchase amount: \n");
16     tally();
17     printf("\nWith tax, the total is: %.2f\n", total);
18
19     return(0);
20 }
21
22                                     /* end main          */

```

#### CBC3RAH2 (File 2):

```

1  /*****
2  ** This file defines the function tally                      **
3  *****/
4
5  #include <stdio.h>
6  #define tax_rate 0.05
7
8  void tally(void)
9  {
10     float tax;
11     extern float total; /* second declaration of total */
12
13     scanf("%f", &total);
14     tax = tax_rate * total;
15     total += tax;
16 }
17
18                                     /* end tally */

```

#### CBC3RAH3 (File 3):

```

1  float total;

```

When you run this program and interaction with it, it could produce the following:

**Output**          Enter the purchase amount:

**Input**            99.95

**Output**          With tax, the total is: 104.95

The following program shows extern variables that are used by two functions. Both functions main and sort can access and change the values of the extern variables string and length. Consequently, main does not have to pass parameters to sort.

## Storage Class Specifiers

### CBC3RAAI:

```
/******  
** 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 you run this program, interacting with it could produce the following:

<b>Output</b>	Enter letters:
<b>Input</b>	zyfab
<b>Output</b>	The sorted string is: abfyz

The following code fragment shows a static variable `var1`, which gets defined at file scope and then declared with the storage class specifier `extern`. The second declaration refers to the first definition of `var1`, and so it has internal linkage.

```
static int var1;  
:  
  
extern int var1;
```

### Related Information

- “File Scope Data Declarations” on page 71
- “Constant Expressions” on page 138
- “Function Definitions” on page 178
- “Function Declarator” on page 180

## register Storage Class Specifier

The register storage class specifier marks heavily used objects (such as loop control variables). It indicates that the compiler should try to minimize access time to the object by placing its value in a machine register, if possible. Because of the limited size and number of registers available on OS/390 systems, few variables can actually be put in registers. The object is treated as having the storage class specifier `auto`.

OS/390 C/C++ requires the register storage class specifier in a block-scope data definition. It also requires it in a parameter declaration that describes an object that has the register storage class. You must define an object that has the register storage class specifier within a block. Or, you must declare it as a parameter to a function.

### 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 that represents the initial value can be any valid C or C++ expression. For structure and union members, the initial value must be a valid constant expression if you use an initializer list. The program then sets the object to that initial value each time it enters the program block that contains the object's definition.

### Storage

Objects with the register storage class specifier have automatic storage duration. Each time a block is entered, storage for register objects that are defined in that block is 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 you invoke recursively, OS/390 C/C++ allocates the memory for the variable at each invocation of the block.

### Restrictions

You cannot use the register storage class specifier in data scope declarations.

**C++ Notes:** In C programs, you cannot apply the address (&) operator to register variables. However, C++ lets you take the address of an object with the register storage class. For example:

```
register i;
int* b = &i;    // valid in C++, but not in C
```

### Related Information

- “Block Scope Data Declarations” on page 70
- “auto Storage Class Specifier” on page 73
- “Address (&)” on page 144
- “Parameter Declaration List Syntax” on page 181

### static Storage Class Specifier

The static storage class specifier lets you define objects with static storage duration and internal linkage, or to define functions with internal linkage.

You can define an object that has the static storage class specifier 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

Objects with the static storage class specifier have static storage duration. OS/390 C/C++ allocates the storage for a static variable when the program begins running. When the program finishes running, it frees the memory.

#### Usage

You can use static variables when you need an object that retains its value from one execution of a block to the next execution of that block. Using the static storage class specifier keeps the system from reinitializing the object each time the block where the object is defined runs.

If a local static variable is a class object with constructors and destructors, OS/390 C++ constructs the object when control passes through its definition for the first time. If a constructor creates a local class object, OS/390 C++ calls its destructor immediately before, or as part of, the calls of the `atexit()` function.

#### Restrictions

You cannot declare a static function at block scope.

#### Examples of Static Storage Class

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:



**CBC3RAJ1 (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);
}

```

**CBC3RAJ2 (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 function `test()` defines the static variable `stat_var`. This variable retains its storage throughout the program, even though `test()` is the only function that can refer to `stat_var`.

**CBC3RAAK:**

```

/*****
** 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;
}

```

## Storage Class Specifiers

```
    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 Information


- “Block Scope Data Declarations” on page 70
- “File Scope Data Declarations” on page 71
- “Function Definitions” on page 178
- “Function Declarator” on page 180

---

## typedef

A typedef declaration lets you define your own identifiers which you can use in place of type specifiers such as `int`, `float`, and `double`. A typedef declaration does not reserve storage. 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.

The syntax of a typedef declaration is:

►►—typedef—*type\_specifier*—*identifier*—;—►►

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.

**C++ Note:** A C++ class defined in a typedef without being named is given a dummy name and the typedef name for linkage. Such a class cannot have constructors or destructors. For example:

```
typedef class {
    Trees();
} Trees;
```

Here the function `Trees()` is an ordinary member function of a class whose type name is unspecified. In the above example, `Trees` is an alias for the unnamed class, not the class type name itself. Consequently, `Trees()` cannot be a constructor for that class.

## Examples of typedef Declarations

The following statements declare `LENGTH` as a synonym for `int` and 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 class type (structure, union, or C++ class). For example:

```
typedef struct {  
    int scruples;  
    int drams;  
    int grains;  
} WEIGHT;
```

You can then use the structure `WEIGHT` in the following declarations:

```
WEIGHT chicken, cow, horse, whale;
```

### Related Information

- “Characters” on page 86
- “Floating-Point Variables” on page 87
- “Integer Variables” on page 89
- “Enumerations” on page 90
- “Pointers” on page 94
- “void Type” on page 99
- “Arrays” on page 100
- “Structures” on page 106
- “Unions” on page 113
- “Chapter 11. C++ Classes” on page 281
- “Constructors and Destructors Overview” on page 325

---

## Type Specifiers

Type specifiers indicate the type of the object or function you are declaring. The fundamental data types are:

- Characters
- Floating-Point Numbers
- Integers
- Enumerations
- Void

From these types, you can derive:

- Pointers
- Arrays
- Structures
- Unions
- Functions

The integral types are `char`, `wchar_t` (C++ *only*), and `int` of all sizes. Floating-point numbers can have types `float`, `double`, or `long double`. You can collectively refer to integral and floating-point types as *arithmetic* types. In C++ *only*, you can also derive the following:

- References

## Type Specifiers

- Classes
- Pointers to Members

In C++, enumerations are not an integral type, but they can be subject to *integral promotion*, as described in “Integral Promotions” on page 167.

You can give names to both fundamental and derived types by using the `typedef` specifier.

## Characters

There are three character data types: `char`, `signed char`, and `unsigned char`. These three data types are not compatible. If you specify `LANG_LVL(ANSI)`, the C compiler recognizes `char`, `unsigned char`, and `signed char` as distinct types. They are always distinct types in C++.

The character data types provide enough storage to hold any member of the character set your program uses at run time. The amount of storage that is allocated for a `char` is implementation-dependent. The OS/390 C/C++ compiler represents a character by 8 bits, as defined in 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`, described in “chars” on page 247.

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 `signed char` or `unsigned char`. When a `char` (signed or unsigned) is widened to an `int`, its value is preserved.

To declare a data object that has a character type, use a `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.

Use the `char` specifier in variable definitions to define such variables as follows: arrays of characters, pointers to characters, and arrays of pointers to characters. Use `signed char` or `unsigned char` to declare numeric variables that occupy a single byte.

**C++ Note:** For the purposes of distinguishing overloaded functions, a C++ `char` is a distinct type from `signed char` and `unsigned char`.

## Examples of Character Data Types

The following example defines the identifier `end_of_string` as a constant object of type `char`. It has 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 Information

- "Character Constants" on page 64
- "Pointers" on page 94
- "Arrays" on page 100
- "Assignment Expressions" on page 162

## Floating-Point Variables

There are three types of floating-point variables: `float`, `double`, and `long double`.

The amount of storage that is allocated for a `float`, a `double`, or a `long double` 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.

To declare a data object that has 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. 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.

Note that OS/390 C/C++ supports IEEE binary floating-point variables as well as IBM S/390 hexadecimal floating-point variables. For details, see "Floating-Point" on page 413, or the section on the `FLOAT` option in the *OS/390 C/C++ User's Guide*.

### Examples of Floating-Point Data Types

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 Information

- “Floating-Point Constants” on page 62
- “Assignment Expressions” on page 162

## Fixed-Point Decimal Data Types (C Only)

Use the type specifier `decimal(n,p)` to declare fixed-point decimal variables and to initialize them with fixed-point decimal constants. For this type specifier, `decimal` is a macro that is defined in `<decimal.h>`. Remember to include `<decimal.h>` if you use fixed-point decimals in your program.

Fixed-point decimal types are classified as arithmetic types. The `decimal(n,p)` type specifier designates a decimal number with *n* digits, and *p* decimal places. *n* is the total number of digits for the integral and decimal parts combined. *p* is the number of digits for the decimal part only. For example, `decimal(5,2)` represents a number, such as, 123.45 where *n*=5 and *p*=2. The value for *p* is optional. If you leave it out, the default value is 0.

In the type specifier, *n* and *p* have a range of allowed values according to the following rules:

```
p <= n
1 <= n <= DEC_DIG
0 <= p <= DEC_PRECISION
```

**Note:** `<decimal.h>` defines `DEC_DIG` (the maximum number of digits *n*) and `DEC_PRECISION` (the maximum precision *p*). Currently, it uses a maximum of 31 digits for both limits.

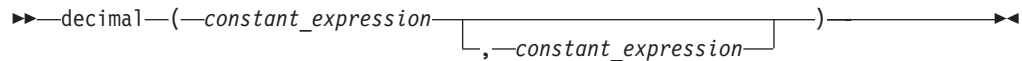
The following examples show how to declare a variable as a fixed-point decimal data type:

```
decimal(10,2) x;
decimal(5,0) y;
decimal(5) z;
decimal(18,10) *ptr;
decimal(8,2) arr[100];
```

In the previous example:

- *x* can have values between -99999999.99D and +99999999.99D.
- *y* and *z* can have values between -99999D and +99999D.
- *ptr* is a pointer to type decimal(18,10).
- *arr* is an array of 100 elements, where each element is of type decimal(8,2).

The fixed-point decimal type specifier has the form:



OS/390 C/C++ evaluates the first *constant\_expression* as a positive integral constant expression. The second *constant\_expression* is optional. If you leave it out, the default value is 0. The type specifiers, `decimal(n,θ)` and `decimal(n)` are type-compatible.

## Integer Variables

Integer variables fall into the following categories:

- short int or short or signed short int or signed short
- signed int or int
- long int or long or signed long int or signed long
- long long int or long long or signed long long int or signed long long
- unsigned short int or unsigned short
- unsigned or unsigned int
- unsigned long int or unsigned long
- unsigned long long int or unsigned long long

The default integer type for a bit field is unsigned. The amount of storage that is allocated for integer data is implementation-dependent.

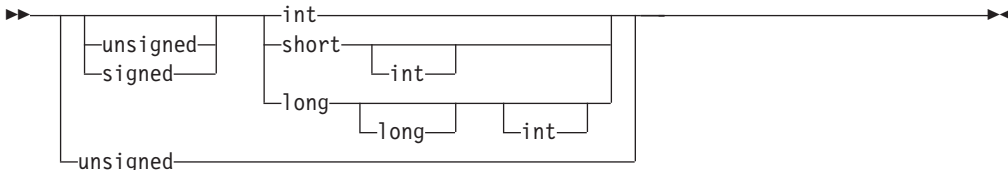
OS/390 C/C++ provides three sizes of integer data types. Objects that are of type short have a length of 2 bytes of storage. Objects that are of type long have a length of 4 bytes of storage. Objects that are of type long long have a length of 8 bytes of storage. An int data type represents the most efficient data storage size on the system (the word-size of the machine) and receives 4 bytes of storage.

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, `uint` reserves the same storage as `int`. Because a signed type reserves a sign bit, an unsigned type can hold a larger positive integer than the equivalent signed type.

To declare a data object that has an integer data type, use an `int` type specifier.

The `int` specifier has the form:

## Type Specifiers



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 you can assign as an integer. The storage class of a variable determines how you can initialize the variable.

**C++ Note:** When the arguments in overloaded functions and overloaded operators are integer types, two integer types that both come from the same group are not treated as distinct types. For example, you cannot overload an `int` argument against a signed `int` argument. “Chapter 13. C++ Overloading” on page 311 describes overloading and argument matching.

## Examples of Integer Data Types

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 Information

- “Integer Constants” on page 60
- “Decimal Constants” on page 61
- “Octal Constants” on page 62
- “Hexadecimal Constants” on page 61

## Enumerations

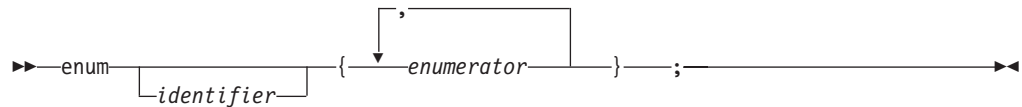
An *enumeration* data type represents a set of values that you declare. You can define an enumeration data type and all variables that have that enumeration type in one statement. You can also declare an enumeration type separately from the definition of variables of that type. You refer to the identifier that is associated with the data type (not an object) as an *enumeration tag*.



**C++ Note:** In C, an enumeration has an implementation-defined integral type. This restriction does not apply to C++. In C++, an enumeration has a distinct type that does not have to be integral.

## Declaring an Enumeration Data Type

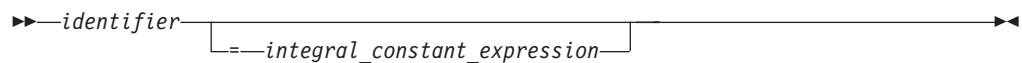
An enumeration type declaration contains the `enum` keyword that is followed by an optional identifier (the enumeration tag) and a brace-enclosed list of enumerators. Commas separate each enumerator in the enumerator list.



The keyword `enum`, that is followed by the `identifier`, names the data type (like the tag on a struct data type). The list of enumerators provides the data type with a set of values.

**C++ Note:** In C, each enumerator represents an integer value. In C++, each enumerator represents a value that you can convert to an integral value.

An enumerator has the form:



To conserve space, you can store enumerations in spaces smaller than the storage required by an `int` data type.

## 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 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 you do not assign an explicit value, 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 that is represented by the previous identifier.

In C, enumeration constants have type `int`.

In C++, each enumeration constant has a value that can be promoted to a signed or unsigned integer value and a distinct type that does not have to be integral. Use an enumeration constant anywhere an integer constant is allowed, or for C++, anywhere a value of the enumeration type is allowed.

## Type Specifiers

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 }
```

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          */
```

## 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` that is followed by the name of the enumeration data type. You must declare the enumeration data type before you can define a variable that has that type.

The initializer for an enumeration variable contains the `=` symbol that is followed by an expression.

In C, the initializer expression must evaluate to an `int` value. In C++, the initializer must behave the same type as the associated enumeration type.

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

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

In C, the type specifier `enum grain` indicates that the value of `g_food` is a member of the enumerated data type `grain`. In C++, the value of `g_food` has the enumerated data type `grain`.

C++ also makes the `enum` keyword optional in an initialization expression like the one in the second line of the preceding example. For example, both of the following statements are valid C++ code:

```
enum grain g_food = barley;
      grain cob_food = corn;
```

## Defining an Enumeration Type and Enumeration Objects

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;
```

C++ also lets you put the storage class immediately before the declarator. For example:

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

Either of these examples 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`. Variable `rating` has the storage class specifier `register`, the data type `enum score`, and the initial value `good`.

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

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

The above example defines the variable `weekday`, which you can assign to any of the specified enumeration constants.

## Example Program Using Enumerations

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 does not correspond with a weekday, the program prints "C'est le mauvais jour."

**CBC3RAAN:**

```
/**
 ** 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;
```

## Type Specifiers

```
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 Information

- “Identifiers” on page 56
- “Enumeration Constants” on page 91
- “Constant Expressions” on page 138

## Pointers

A *pointer* type variable holds the address of a data object or a function. A pointer can refer to an object of any one data type except to a bit field or a reference. Additionally, in C, a pointer cannot point to an object with the register storage class.

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, or members of a C++ class.
- To access an array of characters as a string.
- To pass the address of a variable to a function. (In C++, you can also use a reference to do this.) By referencing a variable through its address, a function can change the contents of that variable. “Calling Functions and Passing Arguments” on page 185 describes passing arguments by reference.

## Declaring Pointers

The following example declares `pcoat` as a pointer to an object that has 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. In the following example, `pvolt` is a constant pointer to an object that has type `short`:

```
short * const pvolt;
```

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

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

The following example defines `psoup` as a `volatile` pointer to an object that has 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 example initializes pointer `amount` to point to `total`:

## Type Specifiers

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

The compiler converts an unsubscripted array name to a pointer to the first element in the array. By specifying the name of the array, you can assign the address of the first element of an array to a pointer. 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;
```

The above example is equivalent to the following:

```
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"
};
```

You can also initialize a pointer to `NULL` by 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.

## Restrictions on C Pointers

The OS/390 C compiler supports only the pointers that are obtained in one of the following ways:

- Directly from a `malloc/calloc/realloc` call
- As an address of a data type (that is, `&variable`)
- From constants
- Received as a parameter from another C function
- Directly from a call to an OS/390 Language Environment service that allocates storage, such as `CEEGETST`

Any bitwise manipulation of a pointer can result in undefined behavior.

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

Packed and nonpacked objects have different memory layouts. Consequently, a pointer to a packed structure or union is incompatible with a pointer to a corresponding nonpacked structure or union. As a result, comparisons and assignments between pointers to packed and nonpacked objects are not valid.

You can, however, perform these assignments and comparisons with type casts. In the following example, the cast operation lets you compare the two pointers, but you must be aware that `ps1` still points to a nonpacked object:

```
int main(void)
{
    _Packed struct ss *ps1;
    struct ss          *ps2;
    .
    .
    .
    ps1 = (_Packed struct ss *)ps2;
    .
    .
    .
}
```

## Using Pointers

You can use two operators when you are 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`. It defines the variable `p_to_x` 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 to which `p_to_x` points:

```
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;
```

## 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.

## Type Specifiers

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 to which the pointers refer.

You can compare two pointers with the following operators: `==`, `!=`, `<`, `>`, `<=`, `>=`. See “Chapter 6. Expressions and Operators” on page 133 for more information on these operators.

You define pointer comparisons only when the pointers point to elements of the same array. You can perform pointer comparisons that use the `==` and `!=` operators 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.

## Example Program Using Pointers

The following program contains pointer arrays:

### CBC3RAAQ:

```
/******  
** 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 found\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 (existing names) **
** char *strng is a pointer to character array entered (new name) **
*****/

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 Information

- “Declarators” on page 119
- “volatile and const Qualifiers” on page 120
- “Initializers” on page 127
- “Address (&)” on page 144
- “Indirection (\*)” on page 145

## void Type

The void data type always represents an empty set of values. The only object that you can declare 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 that takes no arguments is void.

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

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

## Type Specifiers

### Example of void Type

Line 7 of the following example declares the function `find_max()` 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.

#### CBC3RAAM:

```
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
13     find_max(numbers, (sizeof(numbers) / sizeof(numbers[0])));
14
15     return(0);
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 */
```

## Arrays

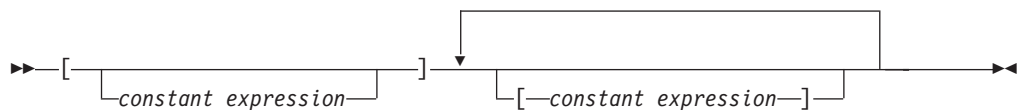
An *array* is an ordered group of data objects. Refer to each object as an *element*. All elements within an array have the same data type.

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

### Declaring Arrays

The array declarator contains an identifier that is followed by an optional *subscript declarator*. An identifier that is preceded by an `*` (asterisk) is an array of pointers.

A subscript declarator has the form:



The subscript declarator describes the number of dimensions in the array and the number of elements in each dimension. Each bracketed expression, or subscript, describes a different dimension and must be a constant expression. Note that the [ and ] characters can be represented by the trigraphs ??( and ??) respectively.

The following example defines a one-dimensional array that contains four elements that have 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];
```

OS/390 C/C++ stores multidimensional arrays in row-major order. When you are referring to elements in order of increasing storage location, the last subscript varies the fastest. For example, consider the following elements of array `roster`:

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

OS/390 C/C++ stores the elements of `roster` as:



You can leave the first, and only the first, set of subscript brackets empty in the following instances:

- 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, OS/390 C/C++ compares the initializer to the subscript declarator to determine the number of elements in the first dimension.

An unsubscripted array name (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 previously been declared. An unsubscripted array name with square brackets (for example, `region[]`) is allowed in the following contexts:

- In arrays that are declared at file scope
- In the argument list of a function declaration

## Type Specifiers

In function declarations and declarations with the `extern` specifier, the only dimension you can leave empty is the first one. You must specify the sizes of additional dimensions.

In extended modes, you can also use unsubscripted array names in the following contexts:

- In union members
- As the last member of a structure

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 two exceptions are when you use an array as an operand of the `sizeof` or the address (`&`) operator.

### Initializing Arrays

The initializer for an array contains the `=` symbol that is followed by a comma-separated list of constant expressions that are 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.

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

The following definition shows a completely initialized one-dimensional array:

```
static int number[3] = { 5, 7, 2 };
```

The array `number` contains the following values:

Element	Value
<code>number[0]</code>	5
<code>number[1]</code>	7
<code>number[2]</code>	2

The following definition shows a partially initialized one-dimensional array:

```
static int number1[3] = { 5, 7 };
```

The values of `number1` are:

Element	Value
<code>number1[0]</code>	5
<code>number1[1]</code>	7
<code>number1[2]</code>	0

Instead of an expression in the subscript declarator that defines the number of elements, the following one-dimensional array definition defines one element for each initializer specified:

```
static int item[ ] = { 1, 2, 3, 4, 5 };
```

The compiler gives `item` the five initialized elements:

Element	Value
<code>item[0]</code>	1
<code>item[1]</code>	2
<code>item[2]</code>	3
<code>item[3]</code>	4
<code>item[4]</code>	5

You can 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 that surround 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 you do not specify the array dimensions.

The following definitions show character array initializations:

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

These definitions create the following elements:

Element	Value	Element	Value	Element	Value
<code>name1[0]</code>	J	<code>name2[0]</code>	J	<code>name3[0]</code>	J
<code>name1[1]</code>	a	<code>name2[1]</code>	a	<code>name3[1]</code>	a
<code>name1[2]</code>	n	<code>name2[2]</code>	n	<code>name3[2]</code>	n
		<code>name2[3]</code>	<code>\0</code>	<code>name3[3]</code>	<code>\0</code>

Note that the following definition would result in the null character being lost:

```
static char name[3]="Jan";
```

In C, the compiler accepts `name[3]` with no warning or error messages. In C++, the compiler generates an error message that states the character array must be at least 4 characters in size to accept the string literal. To initialize this array in C++, use character-by-character initialization, for example:

```
static char name[3]={'J','a','n'};
```

You can initialize a multidimensional array by the following methods:

- 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

## Type Specifiers

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

The following definition explicitly initializes six elements in a 12-element array:

```
static int matrix[3][4] =
{
    { 1, 2 },
    { 3, 4 },
    { 5, 6 }
};
```

The initial values of `matrix` are:

Element	Value	Element	Value
<code>matrix[0][0]</code>	1	<code>matrix[1][2]</code>	0
<code>matrix[0][1]</code>	2	<code>matrix[1][3]</code>	0
<code>matrix[0][2]</code>	0	<code>matrix[2][0]</code>	5
<code>matrix[0][3]</code>	0	<code>matrix[2][1]</code>	6
<code>matrix[1][0]</code>	3	<code>matrix[2][2]</code>	0
<code>matrix[1][1]</code>	4	<code>matrix[2][3]</code>	0

You cannot have more initializers than the number of elements in the array.

### C++ Notes:

1. In C++, you can use a zero-sized array in a class definition, but it must be non-static.
2. In a class definition, the zero-sized array must be the last non-static data member. You can use members such as functions, static data members, and typedefs after the zero-sized array.
3. You cannot use a class that contains a zero-sized array as a base class.

## Example Programs Using Arrays

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

The first for statement prints the element values of `prices`. The second for statement adds five percent to the value of each element of `prices`. of total.

**CBC3RAAO:**

```

/**
 ** 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`.

## Type Specifiers

### CBC3RAAP:

```
/**
 ** 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 Information

- “Pointers” on page 94
- “Array Subscript [ ] (Array Element Specification)” on page 140
- “String Literals” on page 65
- “Declarators” on page 119
- “Initializers” on page 127
- “Chapter 7. Implicit Type Conversions” on page 167

## 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 data types. Each data object in a structure is 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;
```

To allocate storage for more than one address, group the components of each address by defining a structure data type and as many variables as you need to have the structure data type.

In the following example, lines 1 through 7 declare the structure tag `address`:

```
1 struct address {
2     int street_no;
3     char *street_name;
4     char *city;
5     char *prov;
6     char *postal_code;
7 };
8 struct address perm_address;
9 struct address temp_address;
10 struct address *p_perm_address = &perm_address;
```

The variables `perm_address` and `temp_address` are instances of the structure data type `address`. Both contain the members described in the declaration of `address`. The pointer `p_perm_address` points to a structure of `address` and is initialized to point to `perm_address`.

Refer to a member of a structure by specifying the structure variable name with the dot operator (`.`) or a pointer with the arrow operator (`->`) and the member name. For example, both of the following assign a pointer to the string "Ontario" to the pointer `prov` that is in the structure `perm_address`:

```
perm_address.prov = "Ontario";
p_perm_address -> prov = "Ontario";
```

All references to structures must be fully qualified. In the example, you cannot reference the fourth field by `prov` alone. You must reference this field by `perm_address.prov`.

Structures with identical members but different names are not compatible and cannot be assigned to each other. Structures are not intended to conserve storage. If you need direct control of byte mapping, use pointers. "Dot Operator (`.`)" on page 141 and "Arrow Operator (`->`)" on page 141 describe structure member references.

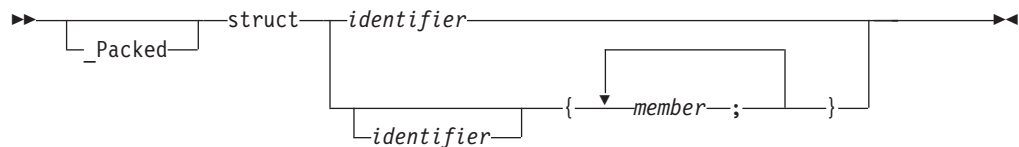
You cannot declare a structure with members of incomplete types. See "Incomplete Types" on page 119 for more information.

## Declaring a Structure

A structure type declaration describes the members that are part of the structure. It contains the `struct` keyword that is followed by an optional identifier (the structure tag), and a brace-enclosed list of members.

A structure declaration has the form:

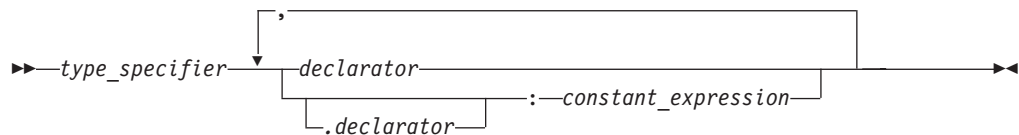
## Type Specifiers



The keyword `struct` followed by the `identifier` (tag) names the data type. If you do not provide a tag name to the data type, you must put all variable definitions that refer to it within the declaration of the data type.

The list of members provides the data type with a description of the values that you can store in the structure.

A structure member definition has the form:



If a `:` (colon) and a constant expression follow the member declarator, the member represents a *bit field*. A member that does not represent a bit field can be of any data type and can have the `volatile` or `const` qualifier. “Declaring and Using Bit Fields in Structures” on page 110 describes bit fields.

You can redefine identifiers that are used as structure or member names 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. You can, however, 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. You can, however, 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.

**C++ Note:** The keyword `struct` is optional in C++.

You can declare structures that have any storage class. Most compilers, however, treat structures that are 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. However, you need to initialize all members in the structure prior to the member of interest. For example, if you are interested in initializing the fifth

member of a structure, you must initialize the first four members, as well. You do not have to initialize the sixth and subsequent members. You cannot initialize unnamed bit fields.

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
<code>perm_address.street_no</code>	3
<code>perm_address.street_name</code>	address of string "Savona Dr."
<code>perm_address.city</code>	Address of string "Dundas"
<code>perm_address.prov</code>	Address of string "Ontario"
<code>perm_address.postal_code</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
<code>temp_address.street_no</code>	44
<code>temp_address.street_name</code>	address of string "Knyvet Ave."
<code>temp_address.city</code>	address of string "Hamilton"
<code>temp_address.prov</code>	address of string "Ontario"
<b><code>temp_address.postal_code</code></b>	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. See “Storage Class Specifiers” on page 73 for details on the initialization of different storage classes.

### 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 you can define a member or a structure variable 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 or a C++ class 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. The constant expression specifies how many bits the field reserves.

Bit fields with a length of 0 must be unnamed. You cannot reference or initialize unnamed bit fields. 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. A `_Packed` structure, a bit field of length 0, causes the next field to align on the next byte boundary.

The maximum bit-field length is implementation dependent.

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
- Have a reference to a bit field (**C++ only**)

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.

A bit field cannot have the `const` or `volatile` 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;
};
```

**C++ Note:** Unlike ANSI/ISO C, C++ bit fields can be any integral type or enumeration type. When you assign an out-of-range value to a bit field, OS/390 C/C++ preserves the low-order bit pattern and assigns the appropriate bits.

If a series of bit fields does not add up to the size of an `int`, padding can take place. OS/390 C/C++ determines the amount of padding by the alignment characteristics of the structure members. In some instances, bit fields can cross word boundaries.

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 that total 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 <code>int</code> boundary
<code>count</code>	The size of an <code>int</code>
<code>ac</code>	4 bits
(unnamed field)	4 bits
<code>clock</code>	1 bit
(padding — 23 bits)	To next <code>int</code> boundary (unnamed field)
<code>flag</code>	1 bit
(padding — 31 bits)	To next <code>int</code> boundary

## Type Specifiers

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 a value that is out of its range to a bit field, OS/390 C/C++ preserves the bit pattern and assigns the appropriate bits. The following expression sets the `toaster` field of the `kitchen` structure to 0 because it assigns only the least significant bit to the `toaster` field:

```
kitchen.toaster = 2;
```

## Declaring a Packed Structure

To qualify a C structure as packed, use `_Packed` qualifier on the structure declaration.

**C++ Note:** C++ does not support the `_Packed` qualifier. To change the alignment of C++ structures, use the `#pragma pack` directive (supported by both C and C++). Refer to “pack” on page 267 for information on this directive.

Packed and nonpacked structures cannot be assigned to each other, regardless of their type.

## Example Program Using Structures

The following program finds the sum of the integer numbers in a linked list:

### CBC3RAAS:

```
/**
** 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 example assigns the following values to the `record` type variables `name1`, `name2`, and `name3`:

Member Name	Value
<code>name1.number</code>	144
<code>name1.next_num</code>	The address of <code>name2</code>
<code>name2.number</code>	203
<code>name2.next_num</code>	The address of <code>name3</code>
<code>name3.number</code>	488
<code>name3.next_num</code>	NULL (Indicating the end of the linked list.)

The variable `recd_pointer` is a pointer to a structure of type `record`. OS/390 C/C++ initializes it 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 following statement advances the pointer to the next object in the list :

```
recd_pointer = recd_pointer->next_num;
```

### Related Information

- “Declarators” on page 119
- “Initializers” on page 127
- “Incomplete Types” on page 119
- “Dot Operator (.)” on page 141
- “Arrow Operator (->)” on page 141

## 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. OS/390 C/C++ overlays the members in storage.

The storage allocated for a union is the storage required for the largest member of the union (plus any padding that is required so that the union will end at a natural boundary of its strictest member).

### C++ Notes:

1. In C++, a union can have member functions, including constructors and destructors, but not virtual member functions. You cannot use a union as a base class nor derive it from a base class.

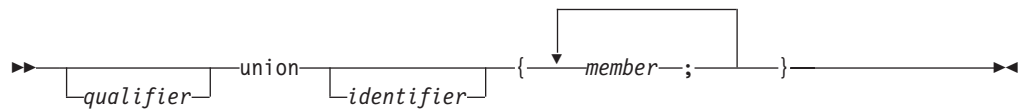
## Type Specifiers

2. A C++ union member cannot be a class object that has a constructor, destructor, or overloaded copy assignment operator. In C++, you cannot declare a member of a union with the keyword `static`.

### Declaring a Union

A union type declaration contains the union keyword followed by an identifier (optional) and a brace-enclosed list of members.

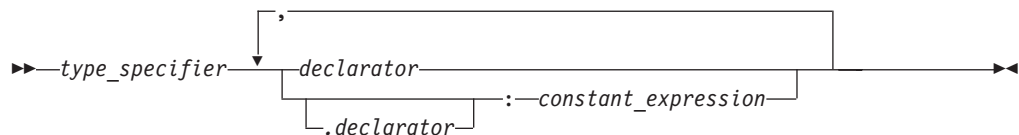
A union declaration has the form:



The *identifier* is a tag you give to the union that is specified by the member list. If you specify a tag, you can make any subsequent declaration of the union (in the same scope) by declaring the tag and omitting the member list. If you do not specify a tag, you must put all variable definitions that refer to that union within the statement that defines the data type.

The list of members provides the data type with a description of the objects that you can store in the union.

A union member definition has the form:



You can reference one of the possible union members the same way as you reference a member of a structure.

For example, the following code assigns `'\n'` to the first element in the character array `birthday`, a member of the union `people`:

```
union {  
    char birthday[9];  
    int age;  
    float weight;  
} people;  
  
people.birthday[0] = '\n';
```

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 that is 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.

**Type Specifier:** The type specifier contains the keyword union that is followed by the name of the union type. You must declare the union data type before you can define a union that has 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.

**Declarator:** The declarator is an identifier, possibly with the volatile or const qualifier.

**Initializer:** 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.

## Defining Packed Unions

To qualify a C union as packed, use `_Packed`.

**C++ Note:** C++ does not support the `_Packed` qualifier. To change the alignment of C++ unions, use the `#pragma pack` directive (which both C and C++ support). For more information on this directive, see “pack” on page 267.

Packed and nonpacked unions cannot be assigned to each other, regardless of their type.

The `#pragma pack` does not affect the memory layout of the union members. Each member starts at offset zero. The `#pragma pack` directive does affect the total alignment restriction of the whole union.

In the following example, each of the elements in the nonpacked `n_array` is of type union `uu`:

```
union uu {
    short a;
    struct {
        char x;
        char y;
        char z;
    } b;
};
```

## Type Specifiers

```
union uu          n_array[2];
/* _Packed union is not supported for C++ */
_Packed union uu  p_array[2];
```

Because it is not packed, each element in the nonpacked `n_array` has an alignment restriction of 2 bytes. (The largest alignment requirement among the union members is that of `short a`.) There is 1 byte of padding at the end of each element to enforce this requirement.

In the packed array, `p_array`, each element is of type `_Packed union uu`. Because every element aligned on the byte boundary, each element has a length of only 3 bytes, instead of the 4 bytes in the previous example.

The following equivalent C++ example uses the `#pragma pack` directive instead of the `_Packed` qualifier:

```
union uu {
    short a;
    struct {
        char x;
        char y;
        char z;
    } b;
};

union uu          n_array[2];
#pragma pack(pack)
union uu p_array[2];
#pragma pack(reset)
```

## Anonymous Unions in C

You can declare unions without declarators if they are members of another structure or union. Refer to unions without declarators as *anonymous unions*. C supports anonymous unions only when you use the `LANG_LVL(COMMONC)` compiler option.

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;
    }; /* no declarator */
} kurt;
```

You can make the following statements:

```
kurt.a = 5;
kurt.b = 36;
```

You can also declare an anonymous union:

1. By 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;
```

2. 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 the examples, you can access the members 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 you declare a union 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;
};
```

You cannot use the variables `b` and `c` from this union at file scope, and so the following statements generate errors:

```
b = 5;
c = 2.5;
```

## Anonymous Unions in C++

A C++ anonymous union is a union without a class name. A declarator cannot follow an anonymous union. An anonymous union is not a type; it defines an unnamed object and it cannot have member functions.

The member names of an anonymous union must be distinct from other names within the scope in which the union is declared. You can use member names directly in the union scope without any additional member access syntax.

For example, in the following code fragment, you can access the data members `i` and `cptr` directly because they are in the scope that contains the anonymous union. Because `i` and `cptr` are union members and have the same address, you should only use one of them at a time. The assignment to the member `cptr` will change the value of the member `i`.

## Type Specifiers

```
void f()
{
    union { int i; char* cptr ; };
    //      .
    //      .
    //      .
    i = 5;
    cptr = "string_in_union"; // overrides i
}
```

An anonymous union cannot have protected or private members. You must declare a global anonymous union with the keyword `static`.

## Examples of Unions

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 that is named `records`. Each element of `records` contains three members: the integer `id_num`, the integer `type_of_input`, and the union variable `input`. The variable `input` 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 Information

- “Declarators” on page 119
- “Initializers” on page 127
- “Structures” on page 106
- “Dot Operator (.)” on page 141

- “Arrow Operator (→)” on page 141

## 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 */
```

In the preceding example, `void` is an incomplete type that you cannot complete. You must complete structure or union and enumeration tags before using them to declare an object. You can, however, define a pointer to an incomplete structure or union.

### Related Information

- “void Type” on page 99
- “Arrays” on page 100
- “Structures” on page 106
- “Unions” on page 113

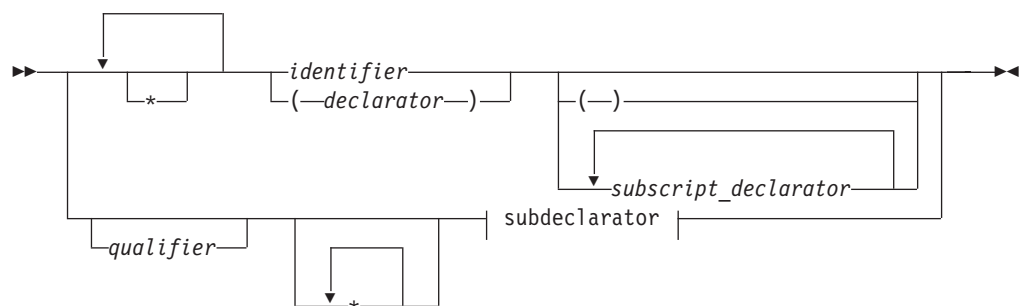
---

## Declarators

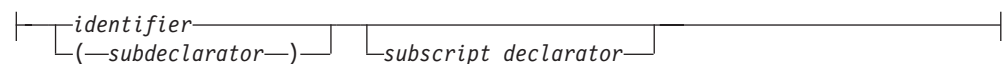
A *declarator* designates a data object or function. Declarators appear in all data definitions and declarations, and in some type definitions.

In a declarator, you can specify the type of an object to be an array, a pointer, or a reference. You can specify that the return type of a function is a pointer or a reference. You can also perform initialization in a declarator.

A declarator has the form:



### subdeclarator:



A qualifier is one of:

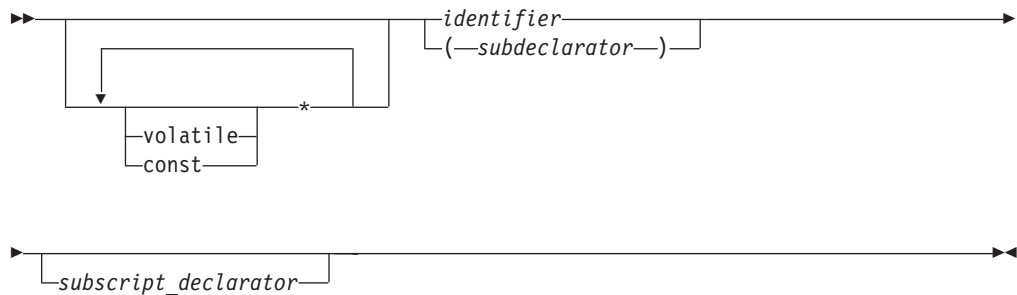
## Declarators

- `const`
- `volatile`

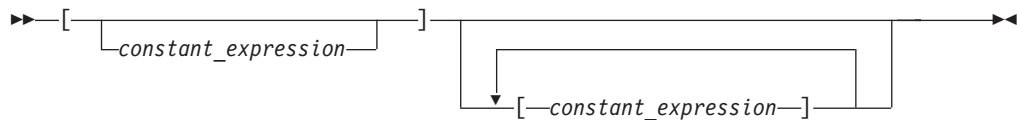
The OS/390 C compiler also implements the `_Packed` qualifier, and the OS/390 C++ compiler also implements the `_Export` qualifier.

In C, you cannot declare or define a `volatile` or `const` function. C++ class member functions can be qualified with `const` or `volatile`.

A declarator can contain a *subdeclarator*. A subdeclarator has the form:



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



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`.

You can define or declare a structure, union, or array. Use a declarator that contains an identifier which names the data object, and some combination of symbols and identifiers which describe the type of data that the object represents.

The following declaration uses `compute[5]` as the declarator:

```
extern long int compute[5];
```

## volatile and const Qualifiers

The `volatile` qualifier maintains consistency of memory access to data objects. It tells the compiler that the variable should always contain its current value even when optimized. This is necessary so the variable can be queried when an exception occurs. OS/390 C/C++ reads volatile objects from memory each time it needs their value, and writes back to memory each time they are changed.

The `volatile` qualifier is useful for data objects that have values that can change in ways unknown to your program (such as the system clock). Do not change or move portions of an expression that reference `volatile` objects.

The `const` qualifier explicitly declares a data object as a data item that you cannot change. OS/390 C/C++ sets its value at initialization. You cannot use `const` data objects in expressions that require a modifiable lvalue. For example, a `const` data object cannot appear on the left side of an assignment statement. (An lvalue is an expression whose address you can take; you can examine or change the object that the lvalue represents. For more information on lvalues, see “lvalues” on page 136.)

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 *x;      /* x is a pointer to a volatile int */
```

or

```
int volatile *x;      /* x is a pointer to a volatile int */
```

```
const int *y;         /* y is a pointer to a const int */
```

or

```
int const *y;         /* y 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 to which `y` points is a constant integer and you cannot change it. 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;
```

The above example provides the same storage as:

```
struct omega {
    int limit;
    char code;
} volatile group;
```

In both examples, 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, union, or class also apply to the members of the structure, union, or class.

## Declarators

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 can declare or define a `volatile` or `const` function only if it is a C++ member function. You can define or declare any function to return a pointer to a `volatile` or `const` function.

You can put more than one qualifier on a declaration, but you cannot specify the same qualifier more than once on a declaration.

## **`_Packed` Qualifier (C Only)**

OS/390 C/C++ stores data elements of structure and unions in memory on an address boundary specific for that data type. For example, a `double` value is stored in memory on a doubleword (8-byte) boundary. There may be gaps left in memory between structure and union elements to align elements on their natural boundaries. You can reduce the padding of bytes within a structure or union by *packing*.

The `_Packed` qualifier removes padding between members of structures and affects the alignment of unions whenever possible. However, the storage that is saved using packed structures and unions may come at the expense of run time performance. Most machines access data more efficiently if the data aligns on appropriate boundaries. With packed structures and unions, members are generally not aligned on natural boundaries. The result is that operations using the class-member access operators (`.` and `->`) are slower.

**Note:** OS/390 C/C++ aligns pointers on their natural boundaries, 4 bytes, even in packed structures and unions.

You can only use `_Packed` with structures or unions. If you use `_Packed` with other types, OS/390 C/C++ generates a warning message, and the qualifier has no effect on the declarator it qualifies. Packed and nonpacked structures and unions have different storage layouts.

You cannot perform comparisons between packed and nonpacked structures, or unions of the same type. Packed and nonpacked structures or unions cannot be assigned to each other, regardless of their type.

You cannot pass a packed union or packed structure as a function parameter if the function expects a nonpacked version. If the function expects a packed structure or a packed union, you cannot pass a nonpacked version as a function parameter.



If you specify the `_Packed` qualifier on a structure or union that contains a structure or union as a member, the qualifier is not passed on to the contained structure or union. See “Pragma Directives (`#pragma`)” on page 243 for more information on `#pragma pack`.

## `__cdecl` Keyword (C++ Only)

Use the `__cdecl` keyword to set linkage conventions for function calls in C++ applications. You can use the `__cdecl` linkage keyword at any language level. The `__cdecl` keyword instructs the compiler to read and write a parameter list by using C linkage conventions.

To set the `__cdecl` calling convention for a function, place the linkage keyword immediately before the function name or at the beginning of the declarator. For example:

```
void __cdecl f();
char (__cdecl *fp) (void);
```

OS/390 C/C++ allows the `__cdecl` keyword on member functions and nonmember functions. These functions can be static or non-static. It also allows the keyword on pointer-to-member function types and the typedef specifier.

**Note:** The compiler accepts both `_cdecl` and `__cdecl` (both single and double underscore).

Following is an example:

```
// C++ nonmember functions
void __cdecl f1();
static void __cdecl f2();

// pointer to member function type
char (__cdecl *A::mfp) (void);

// typedef
typedef void (*_cdecl void_fcn)(int);
// C++ member functions
class A {
public:
    void __cdecl func();
    static void __cdecl func1();
}

// Template member functions
template <class T> X {
public:
    void __cdecl func();
    static void __cdecl func1();
}

// Template functions
template <class T> T __cdecl foo(T i) {return i+1;}
template <class T> T static _cdecl foo2(T i) {return i+1;}
```

## Semantics of `__cdecl`

The `__cdecl` linkage keyword only affects parameter passing; it does not prevent function name mangling. Therefore, you can still overload functions with non-default linkage. Note that you only acquire linkage by explicitly using the `__cdecl` keyword. It overrides the linkage that it inherits from an extern “linkage” specification.

## Declarators

Following is an example:

```
void __cdecl foo(int);      // C linkage with name mangled
void __cdecl foo(char);    // overload foo() with char is OK

void foo(int(*)());        // overload on linkage of function
void foo(int (__cdecl *)()); // pointer parameter is OK

extern "C++" {
    void __cdecl foo(int);    // foo() has C linkage with name mangled
}

extern "C" {
    void __cdecl foo(int);    // foo() has C linkage with name mangled
}
```

Overrides of a virtual function must have the same linkage as the introducing function, otherwise an error diagnostic is issued. Following is an example:

```
class A {
public:
    virtual void __cdecl func();
};

class B : public A {
public:
    virtual void func();    // error 1731, Function linkage differs
                           // from the overridden function
};
```

If the function is redeclared, the linkage keyword must appear in the first declaration, otherwise OS/390 issues an error diagnostic. Following are two examples:

```
int c_cf();
int __cdecl c_cf();        // error 1251, the previous declaration
                           // did not have a linkage specification

int __cdecl c_cf();
int c_cf();                // OK, the linkage is inherited from
                           // first declaration
```

## Examples of \_\_cdecl Use

Prior to the Version 2 Release 4 OS/390 C/C++ compiler, the C++ function pointer could not pass in the C function parameter list as the compiler did not support \_\_cdecl linkage. The following examples illustrate how you can pass in the C parameter list by using the \_\_cdecl linkage:

### Example 1

```
/*-----*/
/* C++ source file */
/*-----*/
//
// C++ Application: passing a C++ function pointer to a C function
//
#include <stdio.h>

void __cdecl callcxx() {    // C++ function declares with
    printf(" I am a C++ function\n"); // C calling convention
}

void (__cdecl *p1)();        // declare a function pointer
                           // with __cdecl linkage

extern "C" {
    void CALLC(void (__cdecl *pp)()); // declare an extern C function
}
```

```

}                                     // accepting a __cdecl function
                                     // pointer

void main() {
    p1 = callcxx;                     // assign the function pointer
                                     // to a __cdecl function

    CALLC(p1);                        // call the C function with
                                     // the __cdecl function pointer
}

```

### Example 2

```

/*-----*/
/* C source file                               */
/*-----*/

/*                                           */
/* C Routine: receiving a function pointer with C linkage */
/*                                           */
#include <stdio.h>
extern void CALLC(void (*pp)()){
    printf(" I am a C function\n");
    (*pp)();                                // call the function passed in
}

```

## \_Export Keyword

Use the `_Export` keyword (in C++ applications only) with a function name or external variable to declare that it is to be exported (made available to other modules). For example:

```
int _Export anthony(float);
```

The above statement exports the function `anthony`, if you define the function within the compilation unit. You must define the function in the same compilation unit in which you use the `_Export` keyword.

OS/390 C/C++ allows `_Export` only at file scope. You cannot use it in a typedef. You cannot apply the `_Export` keyword to the return type of a function. For example, the following declaration causes an error :

```
int _Export * a(); // error
```

If the `_Export` keyword is repeated in a declaration, OS/390 C/C++ issues a warning when you specify the `info(gen)` option.

Since `_Export` is part of the declarator, it affects only the closest identifier. In the following declaration, `_Export` only modifies `a`:

```
int _Export a, b;
```

You can use `_Export` at any language level.

The `_Export` keyword is an alternative to the `#pragma export()` directive.

To export member functions, you may apply the `_Export` keyword to the function declaration, but the function definition must not be inlined. For example:

```

Class X {
public:
    ...
    void _Export Print();
    ...
}

```

## Declarators

```
};  
  
void X::Print() {  
    ...  
}
```

The above example will cause the function `X::Print()` to be exported.

**C++ Note:** It is not possible to export C++ inlined functions even with the `#pragma export()` directive.

If the you apply the `_Export` keyword to a class, then OS/390 C/C++ automatically exports any static members of that class. In the example below, both `X::Print()` and `X::GetNext()` will be exported.

```
Class _Export X {  
    public:  
        ...  
        void Print();  
        int GetNext();  
        ...  
};  
  
void X::Print() {  
    ...  
}  
int X::GetNext() {  
    ...  
}
```

You can apply the `_Export` keyword to SOM classes. The function `main()` cannot be exported. For a description of `#pragma export`, see “export” on page 253.

For more information on using DLLs and exporting functions, see the *OS/390 C/C++ Programming Guide*.

## Example Declarators

The following table describes some declarators:

*Table 8. Example Declarators*

Example	Description
<code>int owner</code>	owner is an int data object.
<code>int *node</code>	node is a pointer to an int data object.
<code>int names[126]</code>	names is an array of 126 int elements.
<code>int *action( )</code>	action is a function returning a pointer to an int.
<code>volatile int min</code>	min is an int that has the volatile qualifier.
<code>int * volatile volume</code>	volume is a volatile pointer to an int.
<code>volatile int * next</code>	next is a pointer to a volatile int.
<code>volatile int * sequence[5]</code>	sequence is an array of five pointers to volatile int objects.
<code>extern const volatile int op_system_clock</code>	op_system_clock is a constant and volatile integer with static storage duration and external linkage.
<code>_Packed struct struct_type s</code>	s is a packed structure of type struct_type.

### Related Information

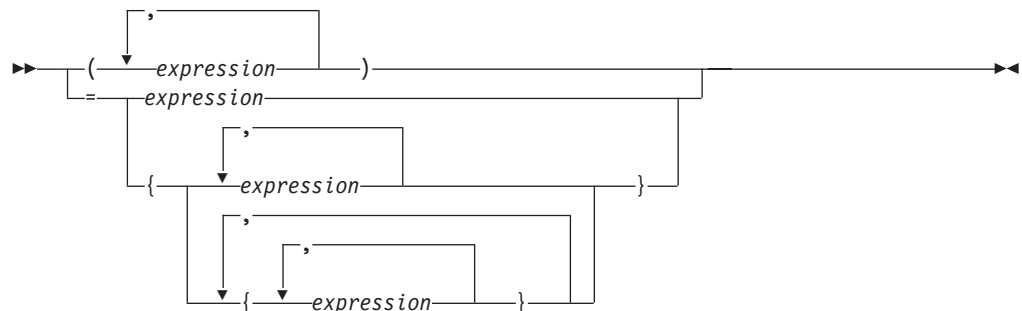
- “Enumerations” on page 90
- “Pointers” on page 94

- “Arrays” on page 100
- “Structures” on page 106
- “Unions” on page 113

## Initializers

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

An initializer has the form:



**C++ Note:** Only C++ allows the form *(expression)*.

The initializer consists of the = symbol that is followed by an initial *expression* or a braced list of initial expressions that are separated by commas. The number of initializers must not be more than the number of elements you will initialize. 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, structures, and aggregate classes, the set of initial expressions must be enclosed in brace brackets ({ }) unless the initializer is a string literal. Aggregate classes refer to classes with no constructors, base classes, virtual functions, or private or protected members.

If the initializer of a character string is a string literal, the brace brackets are optional. You must separate individual expressions by using commas. You can enclose groups of expressions in braces and separate them by using commas.

In an array, structure, or union that you have initialized using a brace-enclosed initializer list, OS/390 C/C++ implicitly initializes any members or subscripts that are not initialized to zero of the appropriate data type.

The section for the data type describes the initialization properties of each data type.

## Initializers

### C++ Notes:

1. You can use an initializer of the form (*expression*) to initialize fundamental types in C++. For example, the following two initializations are identical:  

```
int group = 3;  
int group(3);
```
2. You can also use the (*expression*) form to initialize C++ classes. See “Initialization by Constructor” on page 336 for more information on initializing classes.
3. You can initialize variables at file scope with nonconstant expressions. ANSI/ISO C does not allow this.
4. If your code jumps over declarations that contain initializations, the compiler generates an error. For example, the following code is not valid in C++:  

```
goto skiplabel;    // error - jumped over declaration  
int i = 3;         //    and initialization of i  
  
skiplabel: i = 4;
```
5. You can initialize classes in external, static, and automatic definitions. The initializer contains an equal sign (=) that is followed by a brace-enclosed, comma-separated, list of values. You do not need to initialize all members of a class.

The following example explicitly initializes the first eight elements of the array `grid`. 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 initial values of `grid` are:

Element	Value	Element	Value
grid[0][0]	0	grid[1][2]	1
grid[0][1]	0	grid[1][3]	1
grid[0][2]	0	grid[2][0]	0
grid[0][3]	1	grid[2][1]	0
grid[1][0]	0	grid[2][2]	0
grid[1][1]	0	grid[2][3]	0

## Related Information

- “Block Scope Data Declarations” on page 70
- “File Scope Data Declarations” on page 71
- “Arrays” on page 100
- “Characters” on page 86
- “Enumerations” on page 90
- “Floating-Point Variables” on page 87
- “Integer Variables” on page 89
- “Pointers” on page 94
- “Structures” on page 106
- “Unions” on page 113

---

## C/C++ Data Mapping

The System/390 architecture has the following boundaries in its memory mapping:

- Byte
- Halfword
- Fullword
- Doubleword

The code that is produced by the C/C++ compiler places data types on natural boundaries. Some examples are:

- Byte boundary for `char`
- Byte boundary for decimal(*n,p*) (C only)
- Halfword boundary for `short int`
- Fullword boundary for `int`
- Fullword boundary for `long int`
- Fullword boundary for pointers
- Fullword boundary for `float`
- Doubleword boundary for `double`
- Doubleword boundary for `long double`

For each external defined variable, the OS/390 C/C++ compiler defines a writeable static data instance of the same name. The compiler places other external variables, such as those in programs that you compiled with the NORENT compiler option, in separate CSECTs that are based on their names.

---

## C++ Function Specifiers

The function specifiers `inline` and `virtual` are used only in C++ function declarations, which are described in “Function Declarations” on page 174.

You can use the function specifier `inline` to suggest to the compiler that it incorporate the code of a function into your program code at the point of the call. For more information, see “C++ Inline Functions” on page 195.

You can only use the function specifier `virtual` in nonstatic member function declarations. For more information, see “Virtual Functions” on page 359.

---

## C++ References

A C++ *reference* is an alias or an alternative name for an object. All operations that are applied to a reference act on the object the reference refers to. The address of a reference is the address of the aliased object.

You can define a reference type by placing the `&` after the type specifier. You must initialize all references except function parameters when you define them.

Because you pass arguments of a function by value, a function call does not modify the actual values of the arguments. If a function needs to modify the actual value of an argument, you must pass the argument by *reference*. This is as opposed to being passed by *value*. You can pass arguments by reference by using either references or pointers. In C++, this is transparent. Unlike C, C++ does not force you to use pointers if you want to pass arguments by reference. For example:

## C++ References

```
int f(int&);
void main()
{
    extern int i;
    f(i);
}
```

You cannot tell from the function call `f(i)` that it is passing the argument by reference.

You cannot refer to `NULL`.

## Initializing References

The object that you use to initialize a reference must be of the same type as the reference. Otherwise, it must be of a type that is convertible to the reference type. If you initialize a reference to a constant by using an object that requires conversion, you create a temporary object. The following example creates a temporary object of type `float`:

```
int i;
const float& f = i; // reference to a constant float
```

Attempting to initialize a nonconstant reference with an object that requires a conversion is an error.

Once a reference has been initialized, it cannot be modified to refer to another object. For example:

```
int num1 = 10;
int num2 = 20;

int &RefOne = num1;           // valid
int &RefOne = num2;           // error, two definitions of RefOne
RefOne = num2;                // assign num2 to num1
int &RefTwo;                  // error, uninitialized reference
int &RefTwo = num2;           // valid
```

Note that the initialization of a reference is not the same as an assignment to a reference. Initialization operates on the actual reference by initializing the reference with the object it is an alias for. Assignment operates through the reference on the object to which it refers.

You can declare a reference without an initializer:

- When you use it as an argument declaration
- In the declaration of a return type for a function call
- In the declaration of class member within its class declaration
- When you explicitly use the `extern` specifier.

You cannot have references to any of the following:

- Other references
- Bit fields
- Arrays of references
- Pointers to references

## Related Information

- “Passing Arguments by Reference” on page 188
- “Pointers” on page 94



- “Declarators” on page 119
- “Initializers” on page 127
- “Temporary Objects” on page 333



---

## Chapter 6. Expressions and Operators

Expressions are sequences of operators, operands, and punctuators that specify a computation. OS/390 C/C++ evaluates expressions based on the operators that the expressions contain and the context in which they are used.

An expression can result in an lvalue, rvalue, or no value, and can produce side effects in each case.

**C++ Note:** You can define C++ operators to behave differently when they are applied to operands of class type. Refer to this as operator *overloading*. This chapter describes the behavior of operators that are not overloaded. The C language does not permit overloading.

This chapter discusses the following topics:

- “Operator Precedence and Associativity”
- “Operands” on page 135
- “Lvalues” on page 136
- “Primary Expressions” on page 136
- “Unary Expressions” on page 142
- “Binary Expressions” on page 152
- “Conditional Expressions” on page 160
- “Assignment Expressions” on page 162
- “Assignment Expressions” on page 162
- “Comma Expression (,)” on page 165

### Related Information

- “Chapter 5. Declarations” on page 69
- “Overloading Operators” on page 315

---

## Operator Precedence and Associativity

Two operator characteristics determine how operands group with operators: *precedence* and *associativity*. Precedence is the priority for grouping different types of operators with their operands. 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 above example does not specify the order of subexpression evaluation, you can explicitly force the grouping of operands with operators by using parentheses.

## Operator Precedence and Associativity

In the following expression, the \* and / operations are performed before + because of precedence. In addition, b is multiplied by c before it is divided by d because of associativity:

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

The following table lists the C and C++ language operators in order of precedence and shows the direction of associativity for each operator. In C++, the primary scope resolution operator (::) has the highest precedence, followed by the other primary operators. In C, because there is no scope resolution operator, the other 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 scope resolution	left to right	::
Primary	left to right	() [ ] . ->
Unary	right to left	++ -- + - ! ~ & * (type_name) sizeof new delete digitsof <sup>1</sup> precisionof <sup>1</sup>
C++ Pointer-to-Member	left to right	.* ->*
Multiplicative	left to right	* / %
Additive	left to right	+ -
Bitwise Shift	left to right	<< >>
Relational	left to right	< > <= >=
Equality	left to right	== !=
Bitwise Logical AND	left to right	&
Bitwise Exclusive OR	left to right	^ or ~
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	,

Do not specify the order of evaluation for function call arguments or for the operands of binary operators. Avoid writing ambiguous expressions such as:

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

In the example above, all C language implementations may not evaluate ++y and func1(y) in the same order. If y had the value of 1 before the first statement, you will not know whether or not the value of 1 or 2 is passed to func1(). In the second statement, if i had the value of 1, you will not know whether the first or second array element of x[ ] is passed as the second argument to func2().

---

1. C only

The example does not specify the order of grouping operands with operators in an expression that contains more than one instance of an operator with both associative and commutative properties. The operators that have the same associative and commutative properties are: `*`, `+`, `&`, `|` (or `|`), and `^` (or `^`). You can force the grouping of operands by grouping the expression in parentheses.

### Examples of Expressions and Precedence

The parentheses in the following expressions explicitly show how the compiler groups operands and operators. If parentheses do not appear in these expressions, the compiler groups the operands and operators as indicated by the parentheses.

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

The above example does not specify the order of grouping operands with operators that are both associative and commutative. Consequently, the compiler can group the operands and operators in the following expression:

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

It groups them in the following ways:

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

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

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

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

This expression can give different results that depend 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();
```

---

## Operands

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

### Integral

Character objects and constants, objects that have an enumeration type, and objects that have the type `short`, `int`, `long`, `long long`, `unsigned short`, `unsigned int`, `unsigned long`, or `unsigned long long`

## Operands

<b>Arithmetic</b>	Integral objects and objects that have the type float, double, and long double.
<b>Scalar</b>	Arithmetic objects and pointers to objects of any type. Also C++ references.
<b>Aggregate</b>	Arrays, structures, and unions. Also C++ classes.

Many operators cause conversions from one data type to another. “Chapter 7. Implicit Type Conversions” on page 167 discusses conversions.

---

## Ivalues

An *lvalue* is an expression whose address you can take. You can examine or change the object that the lvalue represents. A *modifiable lvalue* is an expression that represents an object that you can change. It is typically the left operand in an assignment expression. For example, array names and const objects are not modifiable lvalues, but static int objects are.

All assignment operators evaluate their right operand and assign that value to their 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.

### Examples of Ivalues

Expression	lvalue
x = 42;	x
*ptr = newvalue;	*ptr
a++	a

### Related Information

- “Dot Operator (.)” on page 141
- “Arrow Operator (->)” on page 141
- “Assignment Expressions” on page 162
- “Address (&)” on page 144

---

## Primary Expressions

A *primary expression* can be:

- An identifier
- A qualified class name
- A string literal
- A parenthesized expression
- A constant expression
- A function call
- An array element specification
- A structure or union member specification

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

## C++ Scope Resolution Operator (::)

The scope resolution operator (::) is used to qualify hidden names so that you can still use them. You can use the unary scope operator if an explicit declaration of the same name in a block or class hides a file scope name, for example:

```
int i = 10;
int f(int i)
{
    return i ? i : :: i; // return global i if local i is zero
}
```

You can use the *class scope operator* to qualify class names or class member names. You can use a hidden class member name by qualifying it with its class name and the class scope operator. Whenever you follow a class name by a :: operator, OS/390 C/C++ interprets the name as a class name.

In the following example, the declaration of the variable X hides the class type X. However, you can still use the static class member count by qualifying it with the class type X and the scope resolution operator.

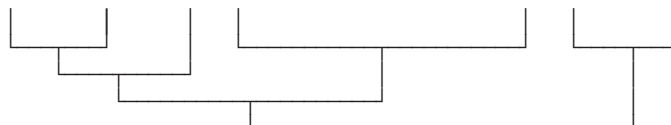
```
#include <iostream.h>
class X
{
public:
    static int count;
};
int X::count = 10;           // define static data member
void main ()
{
    int X = 0;               // hides class type X
    cout << X::count << endl; // use static member of class X
}
```

The scope resolution operator is also discussed in “Class Names” on page 283 and in “Scope of Class Names” on page 286.

## Parenthesized Expressions ( )

Use parentheses to explicitly force the order of expression evaluation. The following expression does not contain any parentheses that are used for grouping operands and operators. The parentheses that surround weight, zipcode 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:

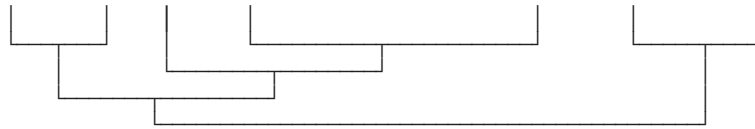
-discount \* item + handling(weight, zipcode) < .10 \* item



The following expression is similar to the previous expression, but it contains parentheses that change the grouping of the operands and operators:

## Primary Expressions

```
(-discount * (item + handling(weight, zipcode) ) ) < (.10 * item)
```



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:

```
x = f + (g + h);
```

## Constant Expressions

A *constant expression* is an expression with a value that may be determined during compilation. It cannot be changed at runtime, it can only be evaluated. You can compose a constant expression with the following:

- Integer constants
- Character constants
- Floating-point constants
- Enumeration constants
- Address constants
- Other constant expressions

Some constant expressions, such as string literals or address constants, are lvalues.

The C and C++ languages require integral constant expressions in the following places:

- In the subscript declarator, as the description of an 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 specified in the preprocessor `#if` statement.)
- In the initializer of a file scope data definition.

In all 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. You can initialize function-scope static and extern declarations.

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 the following:

- integer, character, enumeration, and float constants
- casts to any type
- `sizeof` expressions
- unary address expressions (static objects only)

OS/390 C/C++ does not allow functions, class objects, pointers, and references unless they occur in `sizeof` expressions. Comma operators and assignment operators cannot appear in constant expressions.



## Examples of Constant Expressions

The following examples show constants that are used in expressions.

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

## Function Calls ( )

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

For example:

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

OS/390 C/C++ evaluates the arguments, and initializes each formal parameter with the value of the corresponding argument. The semantics of argument passing are identical to those of assignments. 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 statement in the function definition determines the return value. 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 is copied; the object pointed to is not copied. (See “Pointers” on page 94.)

OS/390 C/C++ converts arguments that are arrays and functions to pointers before passing them as function arguments.

Arguments passed to nonprototyped C functions undergo conversions. OS/390 C/C++ converts short or char parameters to int, and float parameters to double. Use a cast expression for other conversions. (See “Cast Expressions” on page 145 for more information.)

An implicit declaration of `extern int func();` is assumed. Consequently, in C only, if a function definition has external linkage and a return type of int, you can make calls to the function before you explicitly declare it. This is *not* true in C++.

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

- In function prototype format and the parameters differ from the prototype
- OR
- Visible at the point where you call the function.

## Primary Expressions

For example, the declaration of `func` is a prototype. When you call function `func`, OS/390 C/C++ converts parameter `f` to a double, and parameter `c` to an int:

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

The order in which parameters are evaluated is not specified. Avoid such calls as:  
`method(sample1, batch.process--, batch.process);`

In this example, the compiler may evaluate `batch.process--` last, causing the last two 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`, not the values themselves.

### CBC3X06C

```
/**
 ** This example illustrates function calls
 **/

#include <stdio.h>

void func (int a, int b);
int main(void)
{
    int x = 5, y = 7;

    func(x, y);
    printf("In main, x = %d    y = %d\n", x, y);
}
return(0);

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
```

See “Chapter 8. Functions” on page 173 for detailed characteristics of functions.

## Array Subscript [ ] (Array Element Specification)

A primary expression followed by an expression in `[ ]` (square brackets) specifies an element of an array. You can refer to the expression within the square brackets as a *subscript*.

The primary expression must have a pointer type, and the subscript must have integral type. The result of an array subscript is an lvalue.

The first element of each array has the subscript 0. The expression `contract[35]` refers to the 36th element in the array `contract`.

In a multidimensional array, you can reference each element (in the order of increasing storage locations) by incrementing the rightmost subscript most frequently.

For example, the following statement gives the value 100 to each element in the array `code[4][3][6]`:

```
for (first = 0; first <= 3; ++first)
    for (second = 0; second <= 2; ++second)
        for (third = 0; third <= 5; ++third)
            code[first][second][third] = 100;
```

Consider the following expression:

```
*((exp1) + (exp2))
```

By definition, the above expression is identical to the following expression:

```
exp1[exp2]
```

The above expression is also identical to the following:

```
exp2[exp1]
```

“Arrays” on page 100 explains how to define and use an array.

## Dot Operator (.)

Use the `.` (dot) operator to access structure or C++ class members that use a structure object. Specify the member by using a primary expression, followed by a `.` (dot) operator, followed by a *name*. For example:

```
roster[num].name
roster[num].name[1]
```

The primary expression must be an object of type `class`, `struct`, or `union`. The *name* must be a member of that object.

The value of the expression is the value of the selected member. If the primary expression and the *name* are lvalues, the expression value is also an lvalue.

For more information on class members, see “Chapter 12. C++ Class Members and Friends” on page 291. See also “Unions” on page 113 and “Structures” on page 106.

## Arrow Operator (->)

Use the `->` (arrow) operator to access structure or C++ class members using a pointer. A primary expression, that is followed by an `->` (arrow) operator, that is followed by a *name*, designates a member of the object to which the pointer points. For example:

```
roster -> name
```

The primary expression must be a pointer to an object of type `class`, `struct`, or `union`. The *name* must be a member of that object.

## Primary Expressions

The value of the expression is the value of the selected member. If the name is an lvalue, the expression value is also an lvalue.

For more information on class members, see “Chapter 12. C++ Class Members and Friends” on page 291. See also “Unions” on page 113 and “Structures” on page 106.

---

## 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.

As indicated in the following descriptions, you can perform the usual arithmetic conversions on the operands of most unary expressions. See “Arithmetic Conversions” on page 170 for more information.

The following table summarizes the operators for unary expressions:

Increment (++)	Decrement (--)	Unary Plus (+)
Unary Minus (-)	Logical Negation (!)	Bitwise Negation (~)
Address (&)	Indirection (*)	Cast ( <i>type_name</i> )
sizeof	digitof	precisionof
new	delete	throw

### Increment (++)

The increment operator (++) adds 1 to the value of an operand. If the operand is a pointer, it 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.

You can put the ++ before or after the operand. If it appears before the operand, OS/390 C/C++ increments the operand, and uses the incremented value in the expression. If you put the ++ after the operand, OS/390 C/C++ uses the value of the operand in the expression *before* it increments the operand. For example:

```
play = ++play1 + play2++;
```

is equivalent to the following three expressions:

```
play1 = play1 + 1;  
play  = play1 + play2;  
play2 = play2 + 1;
```

**C++ Note:** C++ distinguishes between *prefix* and *postfix* forms of the increment operator: The result of a C++ postfix increment has the same type as the operand, except for possible integral promotion, but is not an lvalue. The result of a C++ prefix increment has the same type as the operand, except for possible integral promotion, and *is* an lvalue. The C language makes no such distinction. The result in C has the same type as the operand, except for possible integral promotion, but is not an lvalue.

You can perform the usual arithmetic conversions on the operand. See “Arithmetic Conversions” on page 170.

## Decrement (--)

The decrement operator (--) subtracts 1 from the value of an operand. If the operand is a pointer, it 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.

You can put the decrement operator before or after the operand. If it appears before the operand, OS/390 C/C++ decrements the operand, and uses the decremented value 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.

For example:

```
play = --play1 + play2--;
```

is equivalent to the following three expressions:

```
play1 = play1 - 1;
play = play1 + play2;
play2 = play2 - 1;
```

**C++ Note:** C++ distinguishes between *prefix* and *postfix* forms of the decrement operator. The result of a C++ postfix decrement has the same type as the operand, except for possible integral promotion, but is not an lvalue. The result of a C++ prefix decrement has the same type as the operand, except for possible integral promotion, and *is* an lvalue. The C language makes no such distinction. The result in C has the same type as the operand, except for possible integral promotion, but is not an lvalue.

OS/390 C/C++ performs the usual arithmetic conversions on the operand. See “Arithmetic Conversions” on page 170.

## Unary Plus (+)

The unary plus operator (+) maintains the value of the operand. The operand can have any arithmetic type. The result is not an lvalue.

The result has the same type as the operand, except for possible integral promotion.

**Note:** Any plus sign in front of a constant is not part of the constant.

## Unary Minus (-)

The unary minus operator (-) negates the value of the operand. The operand can have any arithmetic type. The result is not an lvalue.

For example, if `quality` has the value 100, `-quality` has the value -100.

The result has the same type as the operand, except for possible integral promotion.

**Note:** Any minus sign in front of a constant is not part of the constant.

## Unary Expressions

### Logical Negation (!)

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. It yields 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 `int` and is not an lvalue.

The following two expressions are equivalent:

```
!right;  
right == 0;
```

### Bitwise Negation (~)

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.

Suppose a short integer `x` represents the decimal value 5. The 16-bit binary representation of `x` is:

```
0000000000000101
```

The expression `~x` yields the following result (that is represented here as a 16-bit binary number):

```
1111111111111010
```

Note that you can represent the `~` character by the trigraph `??~`.

The 16-bit binary representation of `~0` is:

```
1111111111111111
```

### Address (&)

The address operator (&) yields a pointer to its operand. The operand must be an lvalue, a function designator, or a qualified name. It cannot be a bit field, nor can it have the storage class `register`.

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 `int`, the result is a pointer to an object that has type `int`.

If the operand is a qualified name and the member is not static, the result is a pointer to a member of class. It has the same type as the member. The result is not an lvalue.

Suppose you define `p_to_y` as a pointer to an `int`, and you define `y` as an `int`. The following expression assigns the address of the variable `y` to the pointer `p_to_y`:

```
p_to_y = &y;
```

Refer to “Pointers” on page 94 for related information.

**C++ Note:** You can use the `&` operator with overloaded functions only in an initialization or assignment where the left side uniquely determines

which version of the overloaded function is used. For more information, see “Overloading Functions” on page 311.

## Indirection (\*)

The indirection operator (\*) determines the value to which the pointer-type operand points.

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. OS/390 C/C++ converts arrays and functions to pointers.

The type of the operand determines the type of the result. For example, if the operand is a pointer to an int, the result has type int.

Do not apply the indirection operator to any pointer that contains an address that is not valid, such as NULL. The result is not defined.

Suppose you define `p_to_y` as a pointer to an int, and you define `y` as an int. Then, following the expressions cause the variable `y` to receive the value 3:

```
p_to_y = &y;
*p_to_y = 3;
```

See also “Pointers” on page 94.

## Cast Expressions

Use the cast operator for *explicit type conversions*. The cast operator converts the value of the operand to a specified data type and performs the necessary conversions to the operand for the type.

For C, the operand must be scalar, and the type must be either scalar or void. For C++, the operand can have class type. If the operand has class type, you can cast it to any type for which the class has a user-defined conversion function. “Conversion Functions” on page 335 describes user-defined conversion functions.

The result of a cast is not an lvalue unless the cast is to a reference type. When you cast to a reference type, OS/390 C/C++ does not perform user-defined conversions, and the result is an lvalue.

There are two types of casts that take one argument:

- *C-style* casts, with the format `(X)a`. Both C and C++ allow these casts.
- *function-style* casts with one argument, such as `X(a)`. Only C++ allows these casts.

Both types of casts convert the argument `a` to the type `X`. In C++, they can invoke a constructor, if the target type is a class, or they can invoke a conversion function, if the source type is a class. They can be ambiguous if both conditions hold.

A function-style cast with no arguments, such as `X()`, creates a temporary object of type `X`. If `X` is a class with constructors, the default constructor `X::X()` is called.

## Unary Expressions

A function-style cast with more than one argument, such as `X(a,b)`, creates a temporary object of type `X`. This object must be a class with a constructor that takes two arguments of types compatible with the types of `a` and `b`. The constructor is called with `a` and `b` as arguments.

- For more information on implicit conversions that use constructors, see “Conversion by Constructor” on page 335.
- You can also do explicit conversions using conversion functions. For more information, see “Conversion Functions” on page 335.
- “Standard Type Conversions” on page 167 describes implicit conversions using standard types.

## sizeof (Size of an Object)

The `sizeof` operator yields the size in *bytes* of the operand. You cannot use the `sizeof` operation on the following:

- A bit field
- A function
- An undefined structure or class
- An incomplete type (such as `void`)

The operand can be the parenthesized name of a type or an expression.

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 `b` is 5 from initialization to the end of program runtime:

```
#include <stdio.h>

int main(void){
    int b = 5;
    sizeof(b++);
    return(0);
}
```

The result is an integer constant.

The size of a `char` object is the size of a byte. For example, if a variable `x` has type `char`, the expression `sizeof(x)` always evaluates to 1.

The result of a `sizeof` operation has type `size_t`. This type is an unsigned integral type that the `<stddef.h>` header file defines.

The size of an object is determined on the basis of its definition. The `sizeof` operator does not perform any conversions. If the operand contains operators that perform conversions, the compiler does take these conversions into consideration. The compiler performs the usual arithmetic conversions due to the following expression. The result of the expression `x + 1` has type `int` (if `x` has type `char`, `short`, or `int` or any enumeration type). It is equivalent to `sizeof(int)`:

```
sizeof (x + 1);
```

Except in preprocessor directives, you can use a `sizeof` expression wherever you require an integral constant. A very common use for the `sizeof` operator is to determine the size of objects that are referred to during storage allocation, input, and output functions.



Another use of `sizeof` is in porting code across platforms. You should use the `sizeof` operator to determine the size that a data type represents, for example:

```
sizeof(int);
```

Using the `sizeof` operator with `decimal(n,p)` results in the total number of bytes that are occupied by the decimal type. OS/390 C/C++ implements decimal data types using the native packed decimal format. Each digit occupies half a byte. The sign occupies an additional half byte. The following example gives you a result of 6 bytes:

```
sizeof(decimal(10,2));
```

**C++ Notes:** The result of a `sizeof` expression depends on the type to which it is applied:

*An array*

The result is the total number of bytes in the array. For example, in an array with 10 elements, the size is equal to 10 times the size of a single element. The compiler does not convert the array to a pointer before evaluating the expression.

*A class*

The result is always nonzero. It is equal to the number of bytes in an object of that class including any padding required for placing class objects in an array.

*A reference*

The result is the size of the referenced object.

## digitsof and precisionof (C Only)

The `digitsof` and `precisionof` operators yield information about decimal types or an expressions of the decimal type. The `<decimal.h>` header file defines the `digitsof` and `precisionof` macros.

The `digitsof` operator gives the number of significant digits of an object, and `precisionof` gives the number of decimal digits. That is,

```
digitsof(decimal(n,p)) = n
precisionof(decimal(n,p)) = p
```

The results of the `digitsof` and `precisionof` operators are integer constants. See “Fixed-Point Decimal Constants (C Only)” on page 63 and “Fixed-Point Decimal Data Types (C Only)” on page 88 for more information about decimal types.

## C++ new Operator

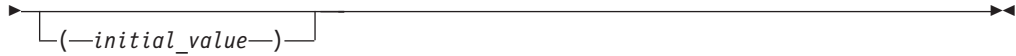
The `new` operator provides dynamic storage allocation. The syntax for an allocation expression that contains the `new` operator is:

```

>> [::] new [ (—argument_list—) ] [ (—type—) ]
    [::] new [ (—argument_list—) ] [ new_type ]

```

## Unary Expressions



If you prefix `new` with the scope resolution operator (`::`), your program uses the global operator `new()`. If you specify an *argument\_list*, your program uses the overloaded `new` operator that corresponds to that *argument\_list*. The *type* is an existing built-in or user-defined type. A *new\_type* is a type that you have not already defined. It can include type specifiers and declarators.

Use an allocation expression that contains the `new` operator to find storage in free store for the object you are creating. The *new expression* returns a pointer to the object created. You can use it to initialize the object. If the object is an array, it returns a pointer to the initial element.

You can use the routine `set_new_handler()` to change the default behavior of `new`. See “`set_new_handler()` — Set Behavior for `new` Failure” on page 150 for more information.

You cannot use the `new` operator to allocate function types, `void`, or incomplete class types because these are not object types. However, you can allocate pointers to functions with the `new` operator. You cannot create a reference with the `new` operator.

When the created object is an array, only the first dimension can be a general expression. All subsequent dimensions must be constant integral expressions. The first dimension can be a general expression even when you are using an existing *type*. You can create an array with zero bounds with the `new` operator. The following example returns a pointer to a unique object:

```
char * c = new char[0];
```

An object created with operator `new()` or operator `new[]()` exists until the program ends, or you call the operator `delete()` or operator `delete[]()`. These calls destroy the objects and deallocate the memory pointed to.

If you use parentheses within a *new\_type*, they should also surround the *new\_type* to prevent syntax errors. In the following example, OS/390 C++ allocates storage for an array of pointers to functions:

```
void f();
void g();
void main()
{
    void (**p)(), (**q)();
    // declare p and q as pointers to pointers to void functions
    p = new (void (*[3])());
    // p now points to an array of pointers to functions
    q = new void(*[3])(); // error
    // error - bound as 'q = (new void) (*[3])();'
    p[0] = f; // p[0] to point to function f
    q[2] = g; // q[2] to point to function g
    p[0](); // call f()
    q[2](); // call g()
}
```

However, the second `new` causes an erroneous binding of:

```
q = (new void) (*[3])()
```

The type of the created object cannot contain class declarations, enumeration declarations, or `const` or `volatile` types. It can contain pointers to `const` or `volatile` objects.

For example, you can use `const char*`, but not `char* const`.

You can supply additional arguments to `new` by using the *argument\_list*, also called the *placement syntax*. If you use placement arguments, a declaration of operator `new()` or operator `new[]()` with these arguments must exist. For example:

```
#include <stddef.h>
class X
{
public:
    void* operator new(size_t,int, int){ /* ... */ }
};
//      .
//      .
//      .
void main ()
{
    X* ptr = new(1,2) X;
}
```

For more information on the class member operator `new()` and operator `new[]()` function, see “Overloaded new and delete” on page 322 and “Free Store” on page 330. For more information on constructing and destructing class objects with `new` and `delete`, see “Constructors and Destructors Overview” on page 325.

## Member Functions and the `new()` and `new[]()` operators

When an object of a class type is created with the `new` operator, the member operator `new()` function (for objects that are not arrays) or the member operator `new[]()` function (for arrays of any number of dimensions) is implicitly called. The first argument is the amount of space requested.

The following rules determine the storage allocation function that OS/390 C++ uses:

1. If your own operator `new[]()` exists, the object is an array, and it does not use the `::` (scope resolution) operator, OS/390 C++ uses your operator `new[]()`.
2. If you have not defined an operator `new[]()` function, the global `::operator new[]()` function defined in `<new.h>` is used. The allocation expression of the form `::operator new[]()` ensures that the global new operator is called, rather than your class member operator.
3. If your own operator `new()` exists, and the object is not an array, and the `::` operator is not used, your operator `new()` is used.
4. If you have not defined an operator `new()` function, the global `::operator new()` function defined in `<new.h>` is used. The allocation expression of the form `::operator new()` ensures that the global new operator is called, rather than your class member operator.

When a nonclass object is created with the `new` operator, the global `::operator new()` is used.

The order of evaluation of a call to an operator `new()` is undefined in the evaluation of arguments to constructors. If operator `new()` returns 0, the arguments to a constructor may or may not have been evaluated.

### Initializing Objects Created with the new Operator

You can initialize objects that are created with the new operator in several ways. For nonclass objects, or for class objects without constructors, a *new initializer* expression can be provided in a new expression by specifying (*expression*) or (). For example:

```
double* pi = new double(3.1415926);
int* score = new int(89);
float* unknown = new float();
```

If a class has a constructor, you must provide the new initializer when you allocate any object of that class. The arguments of the new initializer must match the arguments of a class constructor, unless the class has a default constructor.

You cannot specify an initializer for arrays. You can initialize an array of class objects only if the class has a default constructor. OS/390 C++ calls the constructor to initialize each array element (class object).

Initialization using the new initializer is performed only if new successfully allocates storage.

For more information on the class member operator new() and operator new[] () function, see “Overloaded new and delete” on page 322 in Special Overloaded Operators, and “Free Store” on page 330. For more information on constructing and destructing class objects with new and delete, see “Constructors and Destructors Overview” on page 325.

### set\_new\_handler() — Set Behavior for new Failure

When the new operator creates a new object, it calls the operator new() or operator new[] () function to obtain the needed storage.

When new cannot allocate storage to create a new object, it calls a *new handler* function if one has been installed by a call to set\_new\_handler(). The set\_new\_handler() function is defined in <new.h>. Use it to call a new handler you have defined or the default new handler.

The set\_new\_handler() function has the prototype:

```
typedef void(*PNH)();
PNH set_new_handler(PNH);
```

set\_new\_handler() takes as an argument a pointer to a function (the new handler), which has no arguments and returns void. It returns a pointer to the previous new handler function.

If you do not specify your own set\_new\_handler() function, new returns the NULL pointer.

The following program fragment shows how you could use set\_new\_handler() to return a message if the new operator cannot allocate storage:

```
#include <iostream.h>
#include <new.h>
void no_storage()
{
    cerr << "Operator new failed: no storage is available.\n";
    exit(1);
}
```

```
main()
{
    set_new_handler(&no_storage);
    // Rest of program ...
}
```

If the program fails because new cannot allocate storage, the program exits with the message:

Operator new failed: no storage is available.

## C++ delete Operator

The delete operator destroys the object created with new by deallocating the memory associated with the object.

The delete operator has a void return type. It has the syntax:



The operand of `delete` must be a pointer returned by `new`, and cannot be a pointer to constant. If an attempt to create an object with `new` fails, the pointer returned by `new` will have a zero value. However, it can still be used with `delete`. Deleting a null pointer has no effect.

The `delete[]` operator frees storage allocated for array objects created with `new[]`. The `delete` operator frees storage allocated for individual objects created with `new`.

It has the syntax:



The result of deleting an array object with `delete` is undefined, as is deleting an individual object with `delete[]`. You do not need to specify the array dimensions with `delete[]`.

The results of attempting to access a deleted object are undefined because the deletion of an object can change its value.

If you have defined a destructor for a class, `delete` invokes that destructor. Whether a destructor exists or not, `delete` frees the storage pointed to by calling the function operator `delete()` of the class if one exists.

The global `::operator delete()` is used in the following cases:

- The class has no operator delete().
- The object is of a nonclass type.
- The ::delete expression deletes the object.

The global `::operator delete[]()` is used in the following cases:

- The class has no operator delete[] ()
- The object is of a nonclass type

## Unary Expressions

- The object is deleted with the `::delete[]` expression.

The default global operator `delete()` only frees storage allocated by the default global operator `new()`. The default global operator `delete[]()` only frees storage allocated for arrays by the default global operator `new[]()`.

For more information on the class member operator `new()` and operator `new[]()` functions, see “Overloaded new and delete” on page 322 in Special Overloaded Operators, and “Free Store” on page 330. For more information on constructing and destructing class objects with `new` and `delete`, see “Constructors and Destructors Overview” on page 325.

## C++ throw Expressions

A *throw* expression is used to throw exceptions to C++ exception handlers. It passes control out of the block enclosing the `throw` statement to the first C++ exception handler whose catch argument matches the throw expression. A throw expression is a unary expression of type `void`.

For more information on the throw expression, see Chapter 17. C++ Exception Handling.

---

## Binary Expressions

A *binary expression* contains two operands that are separated by one operator.

Not all binary operators have the same precedence. The table in the section “Operator Precedence and Associativity” on page 133 shows the order of precedence among operators. All binary operators have left-to-right associativity.

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.

As indicated in the following descriptions, OS/390 C++ performs the usual arithmetic conversions on the operands of most binary expressions. See “Arithmetic Conversions” on page 170 for more information.

The following table summarizes the operators for binary expressions:

Multiplication (*)	Division (/)	Remainder (%)
Addition (+)	Subtraction (-)	Bitwise Shifts (<< >>)
Relational (< > <= >=)	Equality (== !=)	Bitwise AND (&)
Bitwise Exclusive OR (^)	Bitwise Inclusive OR ( )	Logical AND (&&)
Logical OR (  )	Pointer-to-Member (*->*)	

## Multiplication (\*)

The multiplication operator `(*)` yields the product of its operands. The operands must have an arithmetic type. The result is not an lvalue. OS/390 C/C++ performs the usual arithmetic conversions on the operands. See “Arithmetic Conversions” on page 170.

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. Consider the following example:

```
sites * number * cost
```

The above expression can be interpreted in any of the following ways:

```
(sites * number) * cost  
sites * (number * cost)  
(cost * sites) * number
```

## Division (/)

The division operator (/) yields the quotient of its operands. The operands must have an arithmetic type. The result is not an lvalue.

If both operands are positive integers and the operation produces a remainder, OS/390 C/C++ ignores the remainder. For example, expression `7 / 4` yields the value 1 (rather than 1.75 or 2). On all IBM C and C++ compilers, if either operand is negative, the result is rounded towards zero.

The result is undefined if the second operand evaluates to 0.

OS/390 C/C++ performs the usual arithmetic conversions on the operands. See “Arithmetic Conversions” on page 170.

## Remainder (%)

The remainder operator (%) yields the remainder from the division of the left operand by the right operand. For example, the expression `5 % 3` yields 2. The result is not an lvalue.

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 `a/b` is representable:

```
( a / b ) * b + a % b;
```

The sign of the remainder is the same as the sign of the quotient.

The usual arithmetic conversions on the operands are performed. See “Arithmetic Conversions” on page 170.

## Addition (+)

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.

When both operands have an arithmetic type, OS/390 C/C++ performs the usual arithmetic conversions on the operands. The result has the type produced by the conversions on the operands and is not an lvalue.

You can add a pointer to an object in an array to a value that has integral type. The result is a pointer of the same type as the pointer operand. The result refers to

## Binary Expressions

another element in the array, offset from the original element by the amount that is specified by the integral value. If the resulting pointer points to storage that is outside the array, other than the first location outside the array, the result is undefined. The compiler does not check the boundary of the pointers. For example, after the addition, `ptr` points to the third element of the array:

```
int array[5];
int *ptr;
ptr = array + 2;
```

See “Pointer Conversions” on page 168 and “Pointer Arithmetic” on page 97 for more information about expressions that contain pointers.

## Subtraction (–)

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.

When both operands have an arithmetic type, OS/390 C/C++ performs the usual arithmetic conversions on the operands. The result has the type produced by the conversions on the operands and is not an lvalue.

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.

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.

See “Pointer Conversions” on page 168 and “Pointer Arithmetic” on page 97 for more information about expressions that contains pointers.

## Bitwise Left and Right Shift (<< >>)

The bitwise shift operators (<< >>) move the bit values of a binary object. The left operand specifies the value to shift. The right operand specifies the number of positions that the bits in the value are shifted. The result is not an lvalue. Both operands have the same precedence and are left-to-right associative.

*Table 9. Bitwise Shift Operators*

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 operands with integral type. Then it converts the right operand 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 16-bit format) of `left_op` is:

```
0000111110110011
```

The expression `left_op << 3` yields:

```
0111110110011000
```

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.

## Relational (< > <= >=)

The relational operators (< > <= >=) compare two operands and determine the validity of a relationship. If the relationship that is 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.

The following table describes the four relational operators:

*Table 10. Relational Operators*

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.

Both operands must have arithmetic types or be pointers to the same type. The result has type `int`.

If the operands have arithmetic types, OS/390 C/C++ performs the usual arithmetic conversions on the operands.

When the operands are pointers, the locations of the objects to which the pointer refer determine the result. If the pointers do not refer to objects in the same array, the result is not defined.

You can compare a pointer to a constant expression that evaluates to 0. You can also compare a pointer to a pointer of type `void*`. OS/390 C/C++ converts the pointer to a pointer of type `void*`.

## Binary Expressions

If two pointers refer to the same object, you can consider them to be equal. If two pointers refer to data members of the same union, they have the same address value.

If two pointers refer to elements of the same array, or to the first element beyond the last element of an array, the pointer to the element with the higher subscript value has the higher address value.

You can only compare members of the same object with relational operators.

Relational operators have left-to-right associativity. For example, consider the following expression:

```
a < b <= c
```

OS/390 C/C++ interprets the expression as follows:

```
(a < b) <= c
```

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*.

## 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 that is 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:

*Table 11. Equality Operators*

Operator	Usage
<code>==</code>	Indicates whether the value of the left operand is equal to the value of the right operand.
<code>!=</code>	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, OS/390 C/C++ performs the usual arithmetic conversions on the operands.

If the operands are pointers, the locations of the objects to which the pointers refer determines the result.

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:** Do not confuse the equality operator (==) with the assignment (=) operator.

For example:

```
if(x == 3)    Evaluates to 1 if x is equal to three. You should code
               equality tests like this with spaces between the operator and
               the operands to prevent unintentional assignments.

while
if(x = 3)     OS/390 C/C++ takes this to be true, because (x = 3)
               evaluates to a non-zero value (3). The expression also
               assigns the value 3 to x.
```

## Bitwise AND (&)

The bitwise AND operator (&) compares each bit of its first operand to the corresponding bit of the second operand. If both bits are 1's, OS/390 C/C++ sets the corresponding bit of the result to 1. Otherwise, it sets the corresponding result bit to 0.

Both operands must have an integral type. OS/390 C/C++ performs the usual arithmetic conversions on each operand. The result has the same type as the converted operands.

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.

The following example shows the values of a, b, and the result of a & b represented as 16-bit binary numbers:

```
bit pattern of a      0000000001011100
bit pattern of b      000000000101110
bit pattern of a & b  0000000000001100
```

**Note:** Do not confuse the bitwise AND (&) operator with the logical AND (&&) operator. For example,

```
1 & 4 evaluates to 0
while
1 && 4 evaluates to 1
```

## Bitwise Exclusive OR (^)

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, OS/390 C/C++ sets the corresponding bit of the result to 0. Otherwise, it sets the corresponding result bit to 1.

Both operands must have an integral type. OS/390 C/C++ performs the usual arithmetic conversions on each operand. The result has the same type as the converted operands and is not an lvalue.

## Binary Expressions

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. Note that you can represent the  $\wedge$  character by the trigraph, `??^`.

The following example shows the values of `a`, `b`, and the result of `a ^ b` represented as 16-bit binary numbers:

bit pattern of <code>a</code>	0000000001011100
bit pattern of <code>b</code>	000000000101110
bit pattern of <code>a ^ b</code>	0000000001110010

**Note:** The bitwise exclusive OR may appear as a  $\wedge$  on your screen. For more information on these symbols, refer to the *OS/390 C/C++ Programming Guide*.

## Bitwise Inclusive OR (`|`)

The bitwise inclusive OR operator (`|`) compares the values (in binary format) of each operand. It 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.

Both operands must have an integral type. OS/390 C/C++ performs the usual arithmetic conversions on each operand. The result has the same type as the converted operands and is not an lvalue.

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. Note that you can represent the  $\mid$  character by the trigraph, `??|`.

The following example shows the values of `a`, `b`, and the result of `a | b` represented as 16-bit binary numbers:

bit pattern of <code>a</code>	0000000001011100
bit pattern of <code>b</code>	000000000101110
bit pattern of <code>a   b</code>	000000000111110

**Note:**

- The bitwise OR may appear as a  $\mid$  on your screen. For more information on these symbols, refer to the *OS/390 C/C++ Programming Guide*.
- Do not confuse the bitwise OR (`|`) operator with the logical OR (`||`) operator. For example,

```
1 | 4 evaluates to 5
while
1 || 4 evaluates to 1
```

## Logical AND (`&&`)

The 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.

Both operands must have a scalar type. OS/390 C/C++ performs the usual arithmetic conversions on each operand. The result has type `int` and is not an lvalue.

Unlike the `&` (bitwise AND) operator, the `&&` operator guarantees left-to-right evaluation of the operands. If the left operand evaluates to 0, OS/390 C/C++ does not evaluate the right operand.

The following examples show how the expressions that contain the logical AND operator are evaluated:

Expression	Result
<code>1 &amp;&amp; 0</code>	0
<code>1 &amp;&amp; 4</code>	1
<code>0 &amp;&amp; 0</code>	0

The following example uses the logical AND operator to avoid division by zero:  
`(y != 0) && (x / y)`

The expression `x / y` is not evaluated when `y != 0` evaluates to 0.

**Note:** The logical AND (`&&`) should not be confused with the bitwise AND (`&`) operator. For example:

```

1 && 4 evaluates to 1
while
1 & 4 evaluates to 0

```

## Logical OR (||)

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.

Both operands must have a scalar type. The usual arithmetic conversions on each operand are performed. The result has type `int` and is not an lvalue.

Unlike the `|` (bitwise inclusive OR) operator, the `||` operator guarantees left-to-right evaluation of the operands. If the left operand has a nonzero value, OS/390 C/C++ does not evaluate the right operand.

The following examples show how OS/390 C/C++ evaluates expressions that contain the logical OR operator:

Expression	Result
<code>1    0</code>	1
<code>1    4</code>	1
<code>0    0</code>	0

The following example uses the logical OR operator to conditionally increment `y`:  
`++x || ++y;`

OS/390 C/C++ does not evaluate the expression `++y` when the expression `++x` evaluates to a nonzero quantity.

## Binary Expressions

**Note:** The logical OR may appear as a `||` on your screen. For more information on these symbols, refer to the *OS/390 C/C++ Programming Guide*.

**Note:** Do not confuse the logical OR (`||`) with the bitwise OR (`|`) operator. For example:

```
1 || 4 evaluates to 1
while
1 | 4 evaluates to 5
```

## C++ Pointer-to-Member Operators (`.*` `->*`)

There are two pointer-to-member operators: `.*` and `->*`.

Use the `.*` operator to dereference pointers to class members. The first operand must be a class type. If the type of the first operand is class type `T`, or is a class that has been derived from class type `T`, the second operand must be a pointer to a member of a class type `T`.

Use the `->*` operator to also dereference pointers to class members. The first operand must be a pointer to a class type. If the type of the first operand is a pointer to class type `T`, or is a pointer to a class derived from class type `T`, the second operand must be a pointer to a member of class type `T`.

The `.*` and `->*` operators bind the second operand to the first. This results in an object or function of the type specified by the second operand.

If the result of `.*` or `->*` is a function, you can only use the result as the operand for the `( )` (function call) operator. If the second operand is an lvalue, the result of `.*` or `->*` is an lvalue.

For more information on pointer-to-member operators, see “Pointers to Members” on page 297.

---

## Conditional Expressions

A *conditional expression* is a compound expression that contains a condition (*operand<sub>1</sub>*), an expression to be evaluated if the condition has a nonzero value (*operand<sub>2</sub>*), and an expression to be evaluated if the condition has the value 0 (*operand<sub>3</sub>*).

Conditional expressions have right-to-left associativity. OS/390 C/C++ evaluates the left operand first, and then evaluates only one of the remaining two operands.

The conditional expression contains one two-part operator. The `?` symbol follows the condition, and the `:` symbol appears between the two action expressions. OS/390 C/C++ treats all expressions that occur between the `?` and `:` 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.

## Conditional Expressions

The second and third operands can also be a pointer or a null pointer constant.

Two objects 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.

OS/390 C/C++ evaluates the first operand, and its value determines whether OS/390 C/C++ evaluates the second or third operand:

- If the value is not equal to 0, it evaluates the second operand.
- If the value is equal to 0, it evaluates the third operand.

The result is the value of the second or third operand.

If the second and third expressions evaluate to arithmetic types, OS/390 C/C++ performs the usual arithmetic conversions on the values. The following tables show how the types of the second and third operands determine the type of the result.

### Type of Conditional C Expressions

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

### Type of Conditional C++ Expressions

Type of One Operand	Type of Other Operand	Type of Result
Reference to type	Reference to type	Reference after usual reference conversions
Class T	Class T	Class T
Class T	Class X	Class type for which a conversion exists. If more than one possible conversion exists, the result is ambiguous.
throw expression	Other (type, pointer, reference)	Type of the expression that is not a throw expression

### Examples of Conditional Expressions

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;
```

## Conditional Expressions

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 that 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 OS/390 C/C++ interprets it as if it were parenthesized this way:

```
int i,j,k;
((i == 7) ? j ++ : k) = j;
```

That is, `k` is treated as the third operand, not the entire assignment expression `k = j`. The 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:

```
int i,j,k;
(i == 7) ? j ++ : (k = j);
```

---

## Assignment Expressions

An *assignment expression* stores a value in the object that is 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 has completed.

In C, the result of an assignment expression is not an lvalue. The result of an assignment expression *is* an lvalue in C++.

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

### Simple Assignment (=)

The simple assignment operator (`=`) stores the value of the right operand in the object that is 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`.



## Assignment Expressions

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 right operand is a pointer to a type, the left operand can be a pointer to a `const` of the same type. If the right operand is a pointer to a `const` type, the left operand must also be a pointer to a `const` type.

If the right operand is a pointer to a type, the left operand can be a pointer to a `volatile` of the same type. If the right operand is a pointer to a `volatile` type, the left operand must also be a pointer to a `volatile` type.

If the left operand is a pointer to a member, the right operand must be a pointer to a member or a constant expression that evaluates to zero. OS/390 C/C++ converts the right operand to the type of the left operand before assignment.

If the left operand is an object of reference type, the assignment is to the object that is denoted by the reference.

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. 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, OS/390 C/C++ remaps the layout of the right operand 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. See “Initializing Pointers” on page 95 for more information on assignments with pointers.

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 */
```

The following example assigns the value of `number` to the member `employee` of the structure `payroll`:

```
payroll.employee = number;
```

The following example assigns in order the value 0 (zero) to `strangeness`, the value of `strangeness` to `charm`, the value of `charm` to `beauty`, and the value of `beauty` to `truth`:

```
truth = beauty = charm = strangeness = 0;
```

## Assignment Expressions

**Note:** The assignment (=) operator should not be confused with the equality comparison (==) operator. For example:

`if(x == 3)`      Evaluates to 1 if x is equal to three  
while

`if(x = 3)`      Is true because (x = 3) evaluates to a non-zero value (3).  
The expression also assigns the value 3 to x.

## Compound Assignment

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.

The following table shows the operand types of compound assignment expressions:

Operator	Left Operand	Right Operand
<code>+=</code> or <code>-=</code>	Arithmetic	Arithmetic
<code>+=</code> or <code>-=</code>	Pointer	Integral type
<code>*=</code> , <code>/=</code> , and <code>%=</code>	Arithmetic	Arithmetic
<code>&lt;&lt;=</code> , <code>&gt;&gt;=</code> , <code>&amp;=</code> , <code>^=</code> , and <code> =</code>	Integral type	Integral type

Note that the expression:

`a *= b + c`

is equivalent to:

`a = a * (b + c)`

and *not*:

`a = a * b + c`

The following table lists the compound assignment operators and shows an expression that uses each operator:

Operator	Example	Equivalent Expression
<code>+=</code>	<code>index += 2</code>	<code>index = index + 2</code>
<code>-=</code>	<code>*(pointer++) -= 1</code>	<code>*pointer = *(pointer++) - 1</code>
<code>*=</code>	<code>bonus *= increase</code>	<code>bonus = bonus * increase</code>
<code>/=</code>	<code>time /= hours</code>	<code>time = time / hours</code>
<code>%=</code>	<code>allowance %= 1000</code>	<code>allowance = allowance % 1000</code>
<code>&lt;&lt;=</code>	<code>result &lt;&lt;= num</code>	<code>result = result &lt;&lt; num</code>
<code>&gt;&gt;=</code>	<code>form &gt;&gt;= 1</code>	<code>form = form &gt;&gt; 1</code>
<code>&amp;=</code>	<code>mask &amp;= 2</code>	<code>mask = mask &amp; 2</code>
<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) that OS/390 C/C++ evaluates twice, OS/390 C/C++ evaluates the left operand only once.

## Comma Expression (,)

A *comma expression* (,) contains two operands that are separated by a comma. Although the compiler evaluates both operands, the value of the right operand is the value of the expression. The compiler evaluates the left operand, possibly producing side effects, and discards the value. 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. OS/390 C/C++ fully evaluates the left operand 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 you require a side effect, but not the immediate result of the expression

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 following function has only three arguments: the value of a, the value 5, and the value of c.

```
f(a, (t = 3, t + 2), c);
```

The value of the second argument is the result of the comma expression in parentheses, which has the value 5:

```
t = 3, t + 2
```

## Comma Expression

The following table gives some examples of the uses of the comma operator:

Statement	Effects
<code>for (i=0; i&lt;2; ++i, f() );</code>	A for 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 if 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 if 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.

---

## Chapter 7. Implicit Type Conversions

There are two kinds of implicit type conversions: standard conversions and user-defined conversions. This chapter describes the standard type conversions that are listed below:

- “Integral Promotions”
- “Standard Type Conversions”
- “Arithmetic Conversions” on page 170

### Related Information

- “Chapter 6. Expressions and Operators” on page 133
- “Chapter 8. Functions” on page 173
- “Cast Expressions” on page 145
- “User-Defined Conversions” on page 334.

---

### Integral Promotions

You can use certain fundamental types wherever you can use an integer. You can convert the following fundamental types through integral promotion:

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

Except for `wchar_t`, if you cannot represent the value by an `int`, OS/390 C/C++ converts the value to an unsigned `int`. For `wchar_t`, if an `int` can represent all the values of the original type, OS/390 C/C++ converts the value to the type that can best represent all the values of the original type. For example, if a `long` can represent all the values, the value is converted to a `long`.

---

### Standard Type Conversions

Many C and C++ operators cause *implicit type conversions*, which change the type of a value. When you add values that have different data types, OS/390 C/C++ first converts both values to the same type. For example, when you add a `short int` value and an `int` value together, the compiler converts the `short int` value to the `int` type.

Implicit type conversions can occur when you:

- Prepare an operand for an arithmetic or logical operation
- Assign an lvalue that has a different type than the assigned value
- Provide a prototyped function that has a different type than the parameter
- Specify a value in the return statement of a function that has a different type from the defined return type for the function

## Standard Type Conversions

You can perform explicit type conversions by using the cast operator or the function style cast. For more information on explicit type conversions, see “Cast Expressions” on page 145.

## Signed-Integer Conversions

The compiler converts a signed integer to a shorter integer. It does this 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 generally takes place without loss of information. However, when you convert an `int`, a `long int`, or a `long long int` value to a `float`, some precision may be lost. When converting a `long long int` type to a `float`, OS/390 C/C++ rounds to the nearest representable number. When converting a signed integer to an unsigned integer, OS/390 C/C++ converts the signed integer to the size of the unsigned integer. It interprets the result as an unsigned value.

When converting a `long long int` type to packed decimal, the resulting size is `decimal(20,0)`.

## Unsigned-Integer Conversions

You can convert an unsigned integer to a shorter unsigned or signed integer by truncating the high-order bits. OS/390 C/C++ converts an unsigned integer to a longer unsigned or signed integer by zero-extending. Zero-extending pads the leftmost bits of the longer integer with binary zeros.

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

## Floating-Point Conversions

If you convert a `float` value to a `double`, it undergoes no change in value. If you convert a `double` to a `float` OS/390 C/C++ represents it 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 OS/390 C/C++ converts a floating-point value to an integer value, it discards the decimal fraction portion of the floating-point value in the conversion. If the result is too large for the given integer type, the result of the conversion is undefined.

## Pointer Conversions

OS/390 C/C++ performs pointer conversions when you use pointers. These conversions include pointer assignment, initialization, and comparison.

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

## Standard Type Conversions

You can convert any pointer to an object that is not a `const` or `volatile` object 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 generally convert an expression with type array of some type to a pointer to the initial element of the array. You cannot do this conversion when the expression is used as the operand of the `&` (address) operator or the `sizeof` operator.

Generally, you can convert an expression with a type of function returning `T` to a pointer to a function returning `T`. You cannot perform this conversion 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.

You can convert a pointer to a class `A` to a pointer to an accessible base class `B` of that class, as long as the conversion is not ambiguous. The conversion is ambiguous if the expression for the accessible base class can refer to more than one distinct class. The resulting value points to the base class subobject of the derived class object. A null pointer (pointer with a zero value) is converted into itself.

For more information on pointer conversions, see “Pointer Arithmetic” on page 97.

## Reference Conversions

A reference conversion can be performed wherever a reference initialization occurs, including reference initialization done in argument passing and function return values. You can convert a reference to a class to a reference to an accessible base class of that class, as long as the conversion is not ambiguous. The result of the conversion is a reference to the base class subobject of the derived class object.

You can perform reference conversion if OS/390 C/C++ allows the corresponding pointer conversion.

## Pointer-to-Member Conversions

Pointer-to-member conversion can occur when you initialize, assign, or compare pointers to members.

A constant expression that evaluates to zero converts to a distinct pointer to a member.

**Note:** A pointer to a member is not the same as a pointer to an object or a pointer to a function.

You can convert a pointer to a member of a base class to a pointer to a member of a derived class, if the following conditions are true:

- The conversion is not ambiguous. The conversion is ambiguous if multiple instances of the base class are in the derived class.
- You can convert a pointer to the derived class to a pointer to the base class. If this is the case, the base class is *accessible*. See “Derivation Access of Base Classes” on page 350 for more information.

## Standard Type Conversions

For more information, see “Pointers to Members” on page 297 and “C++ Pointer-to-Member Operators (. \* ->\*)” on page 160.

## Function Argument Conversions

If no function prototype declaration is visible when you call a function, the compiler can perform default argument promotions. These promotions consist of the following:

- Integral promotions.
- Arguments with type float that convert to type double.

## Other Conversions

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

You cannot convert between structure or union types.

When a packed decimal type is converted to a long long int type, OS/390 C/C++ discards the fractional part.

In C, when you define a value by using the enum type specifier, OS/390 C/C++ treats the value as an int. Conversions to and from an enum value proceed as for the int type.

In C++, you can convert from an enum to any integral type but not from an integral type to an enum.

There are no standard conversions between class types.

---

## Arithmetic Conversions

Most C++ operators perform type conversions to bring the operands of an expression to a common type. Or, they extend short values to the integer size used in OS/390 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 that are ordinarily used in arithmetic.

You can use arithmetic conversions to match the operands of arithmetic operators.



Arithmetic conversion proceeds in the following order:

Operand Type	Conversion
One operand has long double type	The other operand is converted to long double type.
One operand has double type	The other operand is converted to double.
One operand has float type	The other operand is converted to float.
One operand has unsigned long long int type	The other operand is converted to unsigned long long int.
One operand has long long int type	The other operand is converted to long long int.
One operand has unsigned long int type	The other operand is converted to unsigned long int.
One operand has unsigned int type and the other operand has long int type and the value of the unsigned int can be represented in a long int	The operand with unsigned int type is converted to long int.
One operand has unsigned int type and the other operand has long int type and the value of the unsigned int cannot be represented in a long int	Both operands are converted to unsigned long int
One operand has long int type	The other operand is converted to long int.
One operand has unsigned int type	The other operand is converted to unsigned int.
Both operands have int type	The result is type int.

**Note:** On OS/390 C/C++, an int type and a long type are the same length, so unsigned int cannot be represented by a signed long.

## Arithmetic Conversions

---

## Chapter 8. Functions

This chapter describes the structure and use of functions in C and C++. Specifically, it discusses the following topics:

- “Functions Overview”
- “C++ Enhancements to C Functions”
- “Function Declarations” on page 174
- “The main() Function” on page 184
- “Calling Functions and Passing Arguments” on page 185
- “Default Arguments in C++ Functions” on page 190
- “Function Return Values” on page 192
- “Pointers to Functions” on page 193
- “C++ Inline Functions” on page 195

### Related Information

- “Member Functions” on page 293
- “Inline Member Functions” on page 294
- “Chapter 13. C++ Overloading” on page 311
- “Chapter 14. Special C++ Member Functions” on page 325
- “Virtual Functions” on page 359

---

## Functions Overview

Functions specify the logical structure of a program and define how particular operations are to be implemented. A function *declaration* consists of a return type, a name, and an argument list. Use the declaration to declare the format and existence of a function prior to using the function. A function *definition* contains a function declaration and the body of the function. A function can only have one definition.

Both C++ and ANSI/ISO C use the style of declaration that is called *prototyping*. A *function prototype* refers to the return type, name, and argument list components of a function. The compiler uses the prototype to check argument types and to convert arguments. Prototypes can appear several times in a program, if the declarations are compatible. They allow the C compiler to check for mismatches between the parameters of a function call and those in the function declaration.

**C++ Note:** C++ functions *must* use prototypes. Usually, you place them in header files, while you place function definitions in source files. Only C allows functions that do not have prototypes.

---

## C++ Enhancements to C Functions

The C++ language provides many enhancements to C functions. These are:

- Reference arguments, described in “Passing Arguments by Reference” on page 188

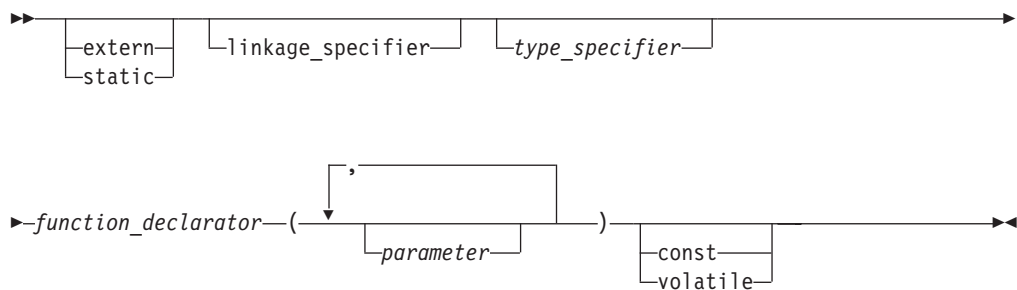
## C++ Enhancements to C Functions

- Default arguments, described in “Default Arguments in C++ Functions” on page 190
- Reference return types, described in “Using References as Return Types” on page 193
- Inline functions, described in “C++ Inline Functions” on page 195
- Member functions, introduced in “Member Functions” on page 293
- Overloaded functions, introduced in “Overloading Functions” on page 311
- Operator functions, introduced in “Overloading Operators” on page 315
- Constructor functions and destructor functions, introduced in “Constructors and Destructors Overview” on page 325
- Conversion functions, introduced in “Conversion Functions” on page 335
- Virtual functions, introduced in “Virtual Functions” on page 359
- Function templates, introduced in “Function Templates” on page 373

---

## Function Declarations

A function declaration establishes the name and the parameters of the function.



A C 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`. Implicit declarations are only valid in C.

To indicate that the function does not return a value, declare it with a return type of `void`.

**C++ Note:** Only C++ supports the use of the `const` and `volatile` specifiers.

## C Function Declarations

A function cannot be declared as returning a data object having a `volatile` or `const` type. It can, however, 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`, 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);
```

## C++ Function Declarations

In C++, you can specify the qualifiers `volatile` and `const` in member function declarations. You can also specify exception specifications in function declarations. You must declare all C++ functions before you can call them.

You cannot define types in return or argument types. For example, the following declarations are not valid in C++:

```
void print(struct X { int i; } x);
//error
enum count{one, two, three} counter(); //error
```

This example attempts to declare a function `print()` that takes an object `x` of class `X` as its argument. However, you cannot have the class definition within the argument list. In the attempt to declare `counter()`, the enumeration type definition cannot appear in the return type of the function declaration. The two function declarations and their corresponding type definitions can be rewritten as follows:

```
struct X { int i; };
void print(X x);
enum count {one, two, three};
count counter();
```

## Multiple Function Declarations

All function declarations for a particular function must have the same number and type of arguments, and must have the same return type and the same linkage keywords. These return and argument types are part of the function type, although the default arguments are not.

For the purposes of argument matching, consider ellipsis and linkage keywords as a part of the function type. You must use them consistently in all declarations of a function. If the only difference between the argument types in two declarations is in the use of typedef names or unspecified argument array bounds, the declarations are the same. A `const` or `volatile` specifier is also part of the function type, but can only be part of a declaration. Or it can be part of a nonstatic member function definition.

Declaring two functions that differ only in return type is not valid function overloading, and the compiler flags it as an error. For example:

```
void f();
int f(); // error, two definitions differ only in
// return type
int g()
{
    return f();
}
```

## Checking Function Calls

The compiler checks C++ function calls by comparing the number and type of the actual arguments used in the function call with the number and type of the formal arguments in the function declaration. It performs implicit type conversion when necessary.

## Function Declaration

### Argument Names in Function Declarations

You can supply argument names in a function declaration, but the compiler ignores them except in the following two situations:

1. If two argument names have the same name within a single declaration, which is an error.
2. If an argument name is the same as a name outside the function, the program hides the name outside the function. You cannot use the name in the argument declaration.

In the following example, the third argument `intersects` is meant to have enumeration type `subway_line`. The name of the first argument, however, hides this name. The declaration of the function `subway()` causes a compile-time error because `subway_line` is not a valid type name in the context of the argument declarations.

```
enum subway_line {yonge, university, spadina, bloor};
int subway(char * subway_line, int stations,
           subway_line intersects);
```

### Examples of Function Declarations

The following example defines the function `absolute()` with the return type `double`. Because this is a noninteger return type, the example declares `absolute` prior to the function call.

#### CBC3RAAV

```
/**
 ** This example shows how a function is declared and defined
 **/

#include <stdio.h>
double absolute(double);

int main(void)
{
    double f = -3.0;

    printf("absolute number = %lf\n", absolute(f));

    return (0);
}

double absolute(double number)
{
    if (number < 0.0)
        number = -number;

    return (number);
}
```

Specifying a return type of `void` on a function declaration indicates that the function does not return a value. The following example defines the function `absolute()` with the return type `void`. The function `main()`, declares `absolute()` with the return type `void`.

**CBC3RAAW**

```

/**
 ** This example uses a function with a void return type
 **/

#include <stdio.h>

int main(void)
{
    void absolute(float);
    float f = -8.7;

    absolute(f);

    return(0);
}

void absolute(float number)
{
    if (number < 0.0)
        number = -number;

    printf("absolute number = %f\n", number);
}

```

The following code fragments show several function declarations. The first fragment declares a function `f` that takes two integer arguments and has a return type of `void`:

```
void f(int, int);
```

The following code fragment declares a function `f1()`. `f1()` takes an integer argument, and returns a pointer to a function that takes an integer argument and returns an integer:

```
int (*f1(int))(int);
```

Alternatively, you can use a typedef for the complicated return type of function `f1()`:

```
typedef int pf1(int);
pf1* f1(int);
```

The following code fragment declares a pointer `p1` to a function that takes a pointer to a constant character and returns an integer:

```
int (*p1) (const char*);
```

The following example declares an external function `f2()`. `f2()` takes a constant integer as its first argument, and can have variable numbers and types of other arguments. It returns type `int`:

```
int extern f2(const int ...);
```

Function `f3()` takes an `int` argument with a default value. This is the value that is returned from function `f2()`, and that has a return type of `int`:

```
const int j = 5;
int f3( int x = f2(j) );
```

See “Default Arguments in C++ Functions” on page 190 for more information about default function arguments.

## Function Declaration

Function `f6()` is a constant class member function of class `X` with no arguments, and with an `int` return type:

```
class X
{
public:
    int f6() const;
};
```

See “const and volatile Member Functions” on page 293 for more information about constant member functions.

Function `f4()` takes no arguments, has return type `void`, and can throw class objects of types `X` and `Y`.

```
class X;
class Y;
//      .
//      .
//      .
void f4() throw(X,Y);
```

Function `f5()` takes no arguments, has return type `void`, and cannot throw an exception.

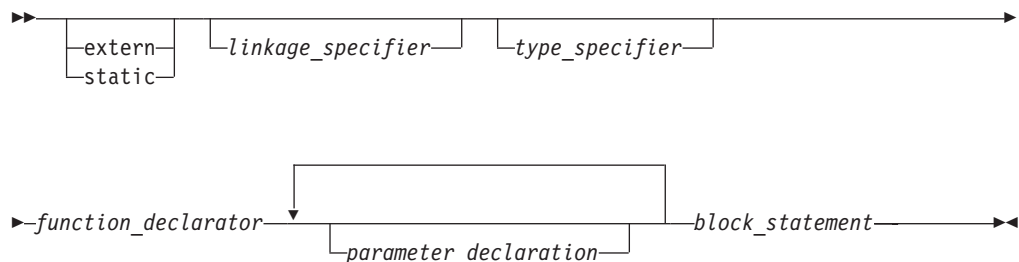
```
void f5() throw();
```

### Related Information:

- “Declarators” on page 119
- “extern Storage Class Specifier” on page 75

## Function Definitions

A *function definition* contains a function declaration and the body of a function. It specifies the function name, formal parameters, the return type, and storage class of the function.



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 you do not specify a storage class specifier, the function has external linkage.
- An optional *linkage specifier*, which determines the linkage of the function. If you do not specify a linkage specifier, the function has the default linkage.
- An optional *type specifier*, which determines the type of value that the function returns. If you do not provide a type specifier, the function has type `int`.



- A *function declarator* provides the function with a name. It can further describe the type of the value that the function returns. The declarator can also list any parameters that the function expects and their types. It encloses the parameters that the function expects 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 arguments that the function can receive. In nonprototype functions, parameters that are not declared have type `int`.

A function can call itself. In addition, other functions can call the function. Unless a function definition has the storage class specifier `static`, functions that appear in other files or modules can also call the function. You can directly invoke functions with a storage class specifier of `static` from within the same source file only.

Consider a function that has the storage class specifier `static`, or a return type other than `int`. In this case, the function definition or a declaration for the function must appear before a call to the function, and must be in the same file as the call.

If a C function definition has external linkage and a return type of `int`, you can make calls to the function before it is visible. This is because the compiler assumes an implicit declaration of `extern int func();`. This is *not* true for C++.

All declarations for a given function must be compatible; that is, the return type must be the same, and the parameters must 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 if you do not pass any parameters to it, use the keyword `void` as the type specifier.

A function can return a pointer or reference to a function, array, or to an object with a `volatile` or `const` type. In C, you cannot declare a function as a struct or union member. (*This restriction does not apply to C++.*)

A function cannot have a return type of function or array. In C, a function cannot return any type that has the `volatile` or `const` qualifier. (*This restriction does not apply to C++.*)

You cannot define an array of functions. You can, however, define an array of pointers to functions.

In the following example, `ary` is an array of two function pointers. The example type casts the values that are assigned to `ary` for compatibility:

## Function Definitions

### CBC3RAAT

```
/**
 ** This example uses an array of pointers to functions
 **/

#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);

    retnum=((int (*)(void))ary[0])();    /* calls func1 */
    printf("number returned = %i\n", retnum);
    ((void (*)(double))ary[1])(num);    /* calls func2 */

    return(0);
}

int func1(void)
{
    int number=3;
    return number;
}

void func2(double a)
{
    a=333.3333;
    printf("result of func2 = %f\n", a);
}
```

The following example is a complete definition of the function sum:

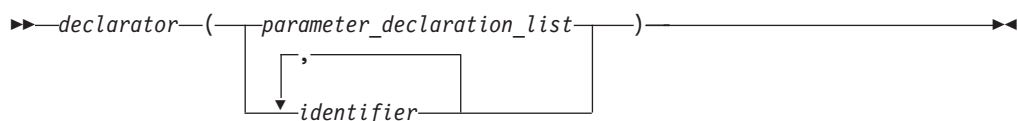
```
int sum(int x,int y)
{
    return(x + y);
}
```

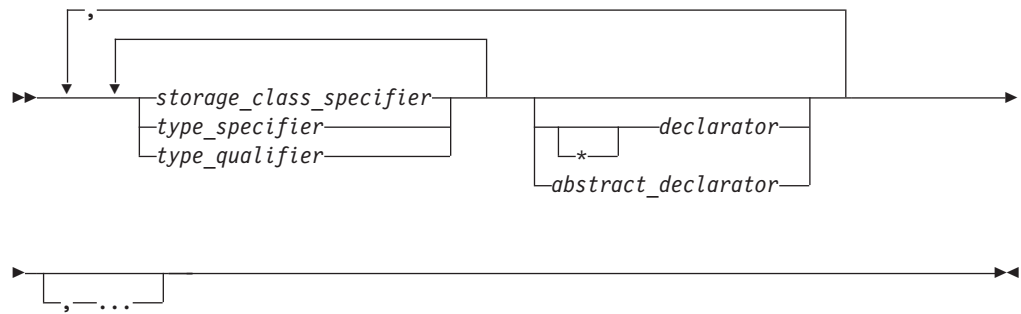
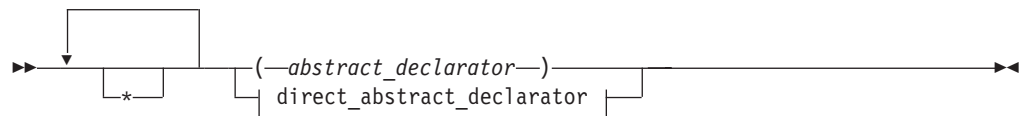
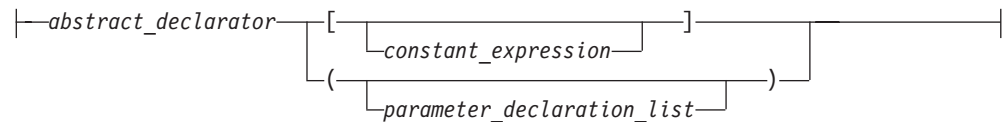
The function sum has external linkage and returns an object that has type int. It has two parameters of type int that are declared as x, and y. The function body contains a single statement that returns the sum of x and y.

### Function Declarator

A *function declarator* contains an identifier that names a function, and a contains a list of the function parameters. You should always use prototype function declarators because you can check the function parameters with them. C++ functions *must* have prototype function declarators.

#### Function Declarator Syntax:



**Parameter Declaration List Syntax:****Abstract Declarator Syntax:****direct\_abstract\_declarator:**

**Prototype Function Declarators:** You should declare each parameter 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:** You should declare each parameter in a *parameter declaration list* following the declarator. If you do not declare a parameter, it has type `int`.

The compiler widens `char` and `short` parameters to `int`, and widens `float` parameters to `double`. The compiler performs no type checking between the argument type and the parameter type for nonprototyped functions. As well, it does not check to ensure that the number of arguments matches the number of parameters.

You should declare each value that a function receives 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 that OS/390 C/C++ allows is the register storage class specifier. It allows any type specifier for a parameter. If you do not specify the register storage class specifier, the parameter will have the auto storage class

## Function Definitions

specifier. In C only, if you omit the type specifier and you are not using the prototype form to define the function, the parameter will have type `int`, as follows:

```
int func(i,j)
{
    /* i and j have type int */
}
```

In C only, 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 the compiler permits, an ellipsis that is preceded by a comma is equivalent to a simple ellipsis. The comma before the ellipsis is optional in C++ only.

```
int f(int,...);
```

For information on how to pass multiple arguments, refer to the sections describing `va_arg`, `va_end()`, and `va_end()` in *OS/390 C/C++ Run-Time Library Reference*.

The compiler promotes parameters as needed, but does not check the types of the variable arguments.

You can declare a function with no arguments in two ways:

```
int f(void);      // ANSI/ISO C Standard

int f();          // C++ enhancement
// Note: In ANSI/ISO C, this declaration means that
// f may take 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. You cannot use `void` as an argument type, although you can use types that are derived from `void` (such as pointers to `void`).

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);
}

```

## Examples of Function Declarators

The following example contains a function declarator `sort` with `table` and `length`. The example declares `table` as a pointer to `int`, and declares `length` as type `int`. Note that the compiler implicitly converts arrays as parameters to a pointer to the type.

### CBC3RAAU

```

/**
 ** This example illustrates function declarators.
 ** Note that arrays as parameters are implicitly
 ** converted to a pointer to the type.
 **/

#include <stdio.h>

void sort(int table[ ], int length);

int main(void)
{
    int table[ ]={1,5,8,4};
    int length=4;
    printf("length is %d\n",length);
    sort(table,length);
}

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 illustrates how you can use a typedef identifier 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`.

## Function Definitions

```
set_date(register struct date *date_ptr)
{
    date_ptr->mon = 12;
    date_ptr->day = 25;
    date_ptr->year = 87;
}
```

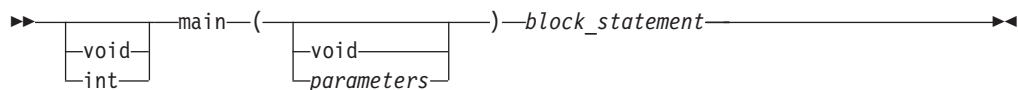
## Related Information

- “Block” on page 198
- “Function Definitions” on page 178
- “Function Declarations” on page 174

---

## The main() Function

When a program begins running, the system automatically calls the function `main`, which marks the entry point of the program. Every program must have one function named `main`. You cannot call any other function in the program `main`. A `main` function has the form:



By default, `main` has the storage class `extern` and a return type of `int`. You can also declare `main` to return `void`.

In C++, you cannot declare `main` as `inline` or `static`. You cannot call `main` from within a program or take the address of `main`.

## Arguments to main

You can declare the function `main` with or without parameters. Although you can give any name to these parameters, you can refer to them as `argc` and `argv`.

The first parameter, `argc` (argument count), has type `int`. It indicates how many arguments you entered on the command line when running the program.

The second parameter, `argv` (argument vector), has type array of pointers to char array objects. char array objects are null-terminated strings.

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. This array contains the program name or the invocation name of the program you are running. If the name cannot be determined, the first element in `argv` points to a null character.

The compiler counts this name as one of the arguments to the function `main`. For example, if you only enter the program name on the command line, `argc` has a value of 1, and `argv[0]` points to the program name.

Regardless of the number of arguments that are entered on the command line, `argv[argc]` always contains `NULL`.

## Example of Arguments to main

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]);
}
```

Consider invoking this program from a command line with the following:

```
backward string1 string2
```

This 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

**Note:** Be careful when entering mixed case characters on a command line because some environments are not case sensitive. Also, the exact format of the string pointed to by *argv[0]* is system dependent.

### Related Information

- “Calling Functions and Passing Arguments”
- “Parameter Declaration List Syntax” on page 181
- “Type Specifiers” on page 85
- “Identifiers” on page 56
- “Block” on page 198

---

## 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. When you call a function, OS/390 C/C++ uses the *actual* arguments to initialize the *formal* arguments.

The type of an actual argument is checked against the type of the corresponding formal argument in the function prototype. All standard and user-defined type conversions are applied, as necessary.

## Calling Functions and Passing Arguments

For example:

```
#include <iostream.h>
#include <math.h>
extern double root(double, double);    // declaration
double root(double value, double base) // definition
{
    double temp = exp(log(value)/base);
    return temp;
}
void main()
{
    int value = 144;
    int base = 2;
    // Call function root and print return value
    cout << "The root is: " << root(value,base) << endl;
}
```

The output is The root is: 12

In the above example, the function `root` is expecting arguments of type `double`. Consequently, the two `int` arguments, `value` and `base`, are implicitly converted to type `double` when you call the function.

The arguments to a function are evaluated before the function is called. When a function call passes an argument, 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 to which the address points. In a case like  $f(g(x))$ , a function is used as an argument. Consequently, OS/390 C/C++ evaluates the function  $g(x)$ , and uses the result of the evaluation as the argument for function  $f$ .

If you pass an array as an argument, OS/390 C/C++ uses a pointer to the array as the argument.

OS/390 C/C++ converts arguments that are passed to parameters in prototype declarations to the declared parameter type. For nonprototype function declarations, OS/390 C/C++ promotes the `char` and `short` arguments to `int`, and `float` to `double`.

You cannot pass a packed structure argument to a function that expects a nonpacked structure of the same type and vice versa. (The same applies to packed and nonpacked unions.)

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, OS/390 C/C++ may evaluate `x++` first, or it may evaluate `x` 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);
```



## Passing Arguments in C++

In C++, if you pass a nonstatic class member function as an argument, OS/390 C/C++ converts the argument to a pointer-to-the-member.

Consider a class that has a destructor or a copy constructor that does more than a bitwise copy. Passing a class object by value results in the construction of a temporary constructor that is actually passed by reference.

It is an error when a function argument is a class object and all of the following properties hold:

- The class needs a copy constructor
- The class does not have a user-defined copy constructor
- You cannot generate a copy constructor for that class

For more information on copy constructors, see “Constructors” on page 326.

## Examples of Calling Functions

The following statement calls the function `startup` and passes no parameters:

```
startup();
```

The following function call causes copies of `a` and `b` to be stored in a local area for the function `sum()`. The function `sum()` runs using the copies of `a` and `b`.

```
sum(a, b);
```

The following function call passes the value 2 and the value of the expression `a + b` to `sum()`:

```
sum(2, a + b);
```

The following statement calls the functions `printf()` and `sum()`. The function `printf()` receives a character string and the return value of `sum()`. The function `sum()` receives the values of `a` and `b`:

```
printf("sum = %d\n", sum(a,b));
```

The following program passes the value of `count` to the function `increment`. `increment` increases the value of the parameter `x` by 1.

### CBC3RAAX

```
/**
 ** This example shows how an argument is passed to a function
 **/

#include <stdio.h>

void increment(int);

int main(void)
{
    int count = 5;

    /* value of count is passed to the function */
    increment(count);
    printf("count = %d\n", count);

    return(0);
}
```

## Calling Functions and Passing Arguments

```
void increment(int x)
{
    ++x;
    printf("x = %d\n", x);
}
```

The output illustrates that the value of count in main remains unchanged:

```
x = 6
count = 5
```

In the following example, main passes the address of count to increment. This example has changed the function increment to handle the pointer. It declares the parameter x as a pointer. The contents to which x points are then incremented.

### CBC3RAAY

```
/**
 ** This example shows how an address is passed to a function
 **/

#include <stdio.h>

int main(void)
{
    void increment(int *x);
    int count = 5;

    /* address of count is passed to the function */
    increment(&count);
    printf("count = %d\n", count);

    return(0);
}

void increment(int *x)
{
    ++*x;
    printf("*x = %d\n", *x);
}
```

The output shows that the above example increases the variable count:

```
*x = 6
count = 6
```

## Passing Arguments by Reference

The term *pass-by-reference* describes a general method of passing arguments from a calling routine to a called routine. If you use a reference type as a formal argument, you can make a pass-by-reference call to a function. In a pass-by-reference call, you can modify the values of arguments in the calling function in the called function. In pass-by-value calls, you can only pass copies of the arguments to the function.

**C++ Note:** The term *reference* in the context of C++ refers to a specific way of declaring objects and functions.

You cannot pass ellipsis arguments as references.

When the actual argument cannot be referenced directly by the formal argument, the compiler creates a temporary variable that is referenced by the formal

## Calling Functions and Passing Arguments

argument. It is initialized using the value of the actual argument. In this case, the formal argument must be a const reference.

You can use reference arguments declared const to pass large objects efficiently to functions. You do not need to make a temporary copy of the object that is passed to the function. Because you declare the reference as const, the function cannot change the actual arguments, for example:

```
void printbig (const bigvar&); // Function prototype
```

When you call the function printbig, it cannot modify the object of type bigvar because a constant reference passes the object.

The following example shows how arguments are passed by reference. Note that OS/390 C/C++ initializes the reference formal arguments with the actual arguments, when you call the function.

### CBC3X06A

```
/**
 ** This example shows how arguments are passed by reference
 **/

#include <iostream.h>
void swapnum(int &i, int &j)
{
    int temp = i;
    i = j;
    j = temp;
}
// .
// .
// .
main()
{
    int    a = 10,    // a is 10
          b = 20;    // b is 20
    swapnum(a,b);    // now a is 20 and b is 10
    cout << "A is :" << a
          << "and B is :"
          << b << endl;
}
```

When the function swapnum() is called, the actual values of the variables a and b are exchanged because they are passed by reference. The output is:

A is : 20 and B is : 10

For the values of the actual arguments to be modified by the function swapnum(), you must define the formal arguments of swapnum() as references.

## Default Arguments in C++ Functions

In C++, you can provide default values for function arguments. All default argument names of a function are bound by declaring the function. OS/390 C/C++ checks the types of all functions at declaration, and evaluates them at each point of call.

### CBC3X06B

```
/**
** This example illustrates default function arguments
**/

#include <iostream.h>
int a = 1;
int f(int a) {return a;}
int g(int x = f(a)) {return f(a);}

int h()
{
    a=2;
    {
        int a = 3;
        return g();
    }
}

main()
{
    cout << h() << endl;
}
```

In this example, the `a` referred to in the declaration of `g()` is the one at file scope. It has the value 2 when `g()` is called. Consequently, this example prints 2 to standard output. The value of `a` is determined after entry into function `h()`, but before the call to `g()` is resolved.

A default argument can have any type.

A pointer to a function must have the same type as the function. Attempts to take the address of a function by reference without specifying the type of the function produce an error. Arguments with default values do not affect the type of a function.

The following example shows that a function with default arguments does not change its type. The default argument allows you to call a function without specifying all of the arguments. It does not allow you to create a pointer to the function that does not specify the types of all the arguments. You can call function `f` without an explicit argument, but you cannot define the pointer `badpointer` without specifying the type of the argument:

```
int f(int = 0);
void g()
{
    int a = f(1);           // ok
    int b = f();            // ok, default argument used
}
int (*pointer)(int) = &f;   // ok, type of f() specified (int)
int (*badpointer)() = &f;  // error, badpointer and f have
                           // different types. badpointer must
                           // be initialized with a pointer to
                           // a function taking no arguments.
```

## Restrictions on Default Arguments

Of the operators, only the function call operator and the operator `new` can have default arguments when you overloaded them.

Arguments with default values must be the trailing arguments in the function declaration argument list. For example:

```
void f(int a, int b = 2, int c = 3); // trailing defaults
void g(int a = 1, int b = 2, int c); // error, leading defaults
void h(int a, int b = 3, int c);    // error, default in middle
```

Once you provide a default argument in a declaration or definition, you cannot redefine that argument, even to the same value. However, you can add default arguments that are not given in previous declarations. For example, the last declaration below attempts to redefine the default values for `a` and `b`:

```
void f(int a, int b, int c=1);    // valid
void f(int a, int b=1, int c);    // valid, add another default
void f(int a=1, int b, int c);    // valid, add another default
void f(int a=1, int b=1, int c=1); // error, redefined defaults
```

You can supply any default argument values in the function declaration or in the definition. All subsequent arguments must have default arguments supplied in this declaration, or a previous declaration of the function.

You cannot use local variables in default argument expressions. For example, the C++ compiler generates errors for both function `g()` and function `h()` below:

```
void f(int a)
{
    int b=4;
    void g(int c=a); // Local variable "a" inaccessible
    void h(int d=b); // Local variable "b" inaccessible
}
```

## Evaluating Default Arguments

When you call a function that is defined with default arguments with the trailing arguments missing, OS/390 C/C++ evaluates the default expressions, for example:

```
void f(int a, int b = 2, int c = 3); // declaration
// ...
int a = 1;
f(a);           // same as call f(a,2,3)
f(a,10);        // same as call f(a,10,3)
f(a,10,20);     // no default arguments
```

OS/390 C/C++ checks the default arguments against the function declaration and evaluates them when you call the function. The order of default argument evaluation is undefined. Default argument expressions cannot use formal arguments of a function, for example:

```
int f(int q = 3, int r = q); // error
```

The value of `q` may not be known when it is assigned to `r`. Consequently, the argument `r` cannot be initialized with the value of the argument `q`. Consider rewriting the above function declaration as follows:

```
int q=5;
int f(int q = 3, int r = q); // error
```

## Default Arguments in C++ Functions

In the above example, the value of `r` in the function declaration still produces an error because the variable `q` defined outside of the function is hidden by the argument `q` declared for the function. Similarly:

```
typedef double D;  
int f(int D, int z = D(5.3) ); // error
```

Here, the compiler interprets the type `D` within the function declaration as the name of an integer. The example hides the type `D` in the argument `D`. The cast `D(5.3)` is therefore not interpreted as a cast because `D` is the name of the argument not a type.

In the following example, you cannot use the nonstatic member `a` as an initializer. The member `a` does not exist until an object of class `X` is constructed. You can use the static member `b` as an initializer, because OS/390 C++ creates `b` independently of any objects of class `X`. You can declare the member `b` after you use it as a default argument. The default values are not analyzed until after the final brace, `}`, of the class declaration.

```
class X  
{  
    int a;  
    f(int z = a) ; // error  
    g(int z = b) ; // valid  
    static int b;  
};
```

You must put parentheses around default argument expressions that contain template references:

```
class C {  
    void f(int i = X<int,5>::y);  
};
```

In the above example, the C++ compiler cannot process the default argument `X<int,5>::y` until the end of the class. Consequently, it cannot tell that the `<` represents the start of a template argument list and not the less than operator.

To avoid error messages, put parentheses around the expression that contains the default argument:

```
class C {  
    void f( int i = (X<int,5>::y) );  
};
```

---

## Function Return Values

A value must be returned from a function unless the function has a return type of `void`. A return statement specifies the return value. The following code fragment shows a function definition, including the return statement:

```
int add(int i, int j)  
{  
    return i + j; // return statement  
}
```

The function `add()` can be called, as shown in the following code fragment:

```
int a = 10,  
    b = 20;  
int answer = add(a, b); // answer is 30
```

In this example, the return statement initializes a variable of the returned type. The example initializes the variable `answer` with the `int` value 30. The compiler checks the type of the returned expression against the returned type. It performs all standard and user-defined conversions, as necessary.

The following return statements show different ways of returning values to a caller:

```
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
```

Other than `main()`, if a function that does not have type `void` returns without a value (as in the first return statement shown in the example above) the result returned is undefined. In C++, the compiler issues an error message as well.

If `main` has a return type of `int`, and does not contain a return expression, it returns the value zero.

Each time a function is called, new copies of its local variables are created. You can reuse the storage for a local variable after the function has terminated. Consequently, the function should not return a pointer to a local variable, or a reference to a local variable.

If the function returns a class object, you may create a temporary object if the class has copy constructors or a destructor. For more information, see “Temporary Objects” on page 333.

## Using References as Return Types

References can also be used as return types for functions. The reference returns the lvalue of the object to which it refers. This allows you to place function calls on the left side of assignment statements. Use referenced return values when overloading assignment operators and subscripting. This way, you can use the results of the overloaded operators as actual values.

**Note:** Returning a reference to an automatic variable gives unpredictable results.

For more information, see “Special Overloaded Operators” on page 319.

---

## Pointers to Functions

A pointer to a function points to the address of the function’s executable code. You can use pointers to call functions and to pass functions as arguments to other functions. You cannot perform pointer arithmetic on pointers to functions. Use the `__cdecl` keyword to declare a pointer to a function as a C linkage. For more information, refer to “`__cdecl` Keyword (C++ Only)” on page 123.

Both the return type and argument types of the function determine the type of a pointer to a function.

A declaration of a pointer to a function must have the pointer name in parentheses. Without them, the compiler interprets the statement as a function that returns a pointer to a specified return type. For example:

## Pointers to Functions

```
int *f(int a);      // function f returning an int*
int (*g)(int a);    // pointer g to a function returning an int
```

In the first declaration, OS/390 C/C++ interprets *f* as a function that takes an *int* as argument. It returns a pointer to an *int*. In the second declaration, OS/390 C/C++ interprets *g* as a pointer to a function that takes an *int* argument and that returns an *int*.

Under OS/390 C/C++, if you pass a function pointer to a function, or the function returns a function pointer, the declared or implied linkages must be the same. Use the *extern* keyword with declarations in order to specify different linkages. Refer to “extern Storage Class Specifier” on page 75 for more information.

The following example illustrates the correct and incorrect uses of function pointers under OS/390 C/C++ :

```
#include <stdlib.h>

extern "C"    int cf();
extern "C++"  int cxxf(); // C++ is included here for clarity;
                        // it is not required; if it is
                        // omitted, cxxf() will still have
                        // C++ linkage.

extern "C"    int (*c_fp)();
extern "C++"  int (*cxx_fp)();
typedef int (*dft_fp_T)();
typedef int (dft_f_T)();

extern "C" {
    typedef void (*cfp_T)();
    typedef int (*cf_pT)();
    void cfn();
    void (*cfp)();
}

extern "C++" {
    typedef int (*cxxf_pT)();
    void cxxfn();
    void (*cxxfp)();
}

extern "C" void f_cprm(int (*f)()) {
    int (*s)() = cxxf;      // error, incompatible linkages-cxxf has
                        // C++ linkage, s has C linkage as it
                        // is included in the extern "C" wrapper
    cxxf_pT j = cxxf;      // valid, both have C++ linkage
    int (*i)() = cf;       // valid, both have C linkage
}

extern "C++" void f_cxprm(int (*f)()) {
    int (*s)() = cf;       // error, incompatible linkages-cf has C
                        // linkage, s has C++ linkage as it is
                        // included in the extern "C++" wrapper
    int (*i)() = cxxf;     // valid, both have C++ linkage
    cf_pT j = cf;         // valid, both have C linkage
}

main() {
    c_fp = cxxf;           // error - c_fp has C linkage and cxxf has
                        // C++ linkage
    cxx_fp = cf;           // error - cxx_fp has C++ linkage and
                        // cf has C linkage
    dft_fp_T dftfpT1 = cf; // error - dftfpT1 has C++ linkage and
                        // cf has C linkage
}
```



```

dft_f_T *dftfT3 = cf;    // error - dftfT3 has C++ linkage and
                        // cf has C linkage
dft_fp_T dftfpT5 = cxxf; // valid
dft_f_T *dftfT6 = cxxf; // valid

c_fp    = cf;           // valid
cxx_fp  = cxxf;         // valid
f_cprm(cf);             // valid
f_cxprm(cxxf);          // valid

// The following errors are due to incompatible linkage of function
// arguments, type conversion not possible
f_cprm(cxxf);           // error - f_cprm expects a parameter with
                        // C linkage, but cxxf has C++ linkage
f_cxprm(cf);            // error - f_cxprm expects a parameter
                        // with C++ linkage, but cf has C linkage
}

```

For OS/390, linkage compatibility affects all C library functions that accept a function pointer as a parameter. The `qsort()` function is an example of these functions (see “extern Storage Class Specifier” on page 75 for a sample program). Refer to the *OS/390 C/C++ Run-Time Library Reference* for more information.

For more information on pointers, see “Pointers” on page 94 and “Pointer Conversions” on page 168.

---

## C++ Inline Functions

Use inline functions to reduce the overhead of a normal function call. Use the inline function specifier, or define a member function within a class or structure definition to declare a function.

The inline specifier is a suggestion to the C++ compiler that it can perform an inline expansion. Instead of transferring control to and from the function code segment, you may directly substitute a modified copy of the function body for the function call.

You can declare and simultaneously define an inline function. If you declared the function with the keyword `inline`, you can declare it 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; }
```

Both member functions and nonmember functions can be inline, and both have internal linkage.

The use of the inline specifier does not change the meaning of the function. The inline expansion of a function may not preserve the evaluation order of the actual arguments.



---

## Chapter 9. Statements

This chapter describes the C and C++ language statements that are listed below:

- “Labels”
- “Block” on page 198
- “break” on page 200
- “continue” on page 202
- “do” on page 203
- “Expression” on page 205
- “for” on page 206
- “goto” on page 208
- “if” on page 209
- “null” on page 210
- “return” on page 211
- “switch” on page 212
- “while” on page 216

### Related Information

- “Scope in C” on page 35
- “Scope in C++” on page 46
- “Chapter 5. Declarations” on page 69
- “Chapter 6. Expressions and Operators” on page 133
- “Chapter 8. Functions” on page 173

---

## 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.

A labelled statement has the form:

►►—*identifier*—:—*statement*—◄◄

The label is the *identifier* and the colon (:) character.

The case and default labels can only appear within the body of a switch statement.

## Examples

```
comment_complete : ;          /* null statement label */
test_for_null : if (NULL == pointer)
```

## Related Information

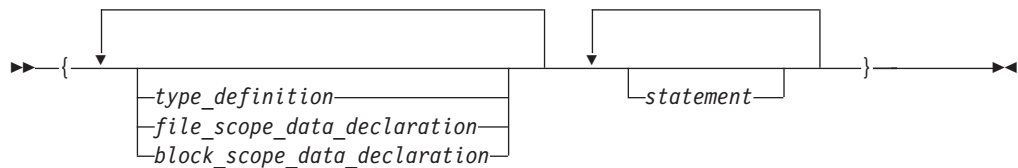
- “Scope in C” on page 35
- “Scope in C++” on page 46
- “goto” on page 208
- “switch” on page 212

---

## Block

A *block statement*, or *compound statement*, lets you group any number of data definitions, declarations, and statements into one statement. The compiler treats all definitions, declarations, and statements that are enclosed within a single set of braces as a single statement. You can use a block wherever a single statement is allowed.

A block statement has the form:



In C, any definitions and declarations must come before the statements.

Redefining a data object inside a nested block hides the outer object while the inner block runs. If a data object is usable within a block and your program does not redefine its identifier, all nested blocks can use that data object.

## Initialization within Block Statements

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.

You only initialize an auto or static local object once, when control passes through its declaration for the first time. OS/390 C/C++ initializes a static variable that is initialized with an expression other than a constant expression to 0 before entering its block for the first time.

**C++ Note:** Unlike ANSI/ISO C, in C++, jumping over a declaration or definition that contains an initializer is an error. For example, the following code produces an error in C++:

```
goto skiplabel;
int i=3        // error, jumped over declaration of i with initializer
skiplabel: i=4;
```

When control exits from a block, all objects with destructors that are defined in the block are destroyed. Your program calls the destructor for an auto or a static local object, only if it constructed the object. Your program must call the destructor before, or as part of, the `atexit` function.

Your program also destroys local variables that are declared in a block on exit. It destroys automatic variables defined in a loop at each iteration.

## Example

The following program shows how the values of data objects change in nested blocks:

### CBC3RAA1

```

1 /**
2  ** This example shows how data objects change in nested blocks.
3  **/
4  #include <stdio.h>
5
6  int main(void)
7  {
8      int x = 1;                /* Initialize x to 1 */
9      int y = 3;
10
11     if (y > 0)
12     {
13         int x = 2;            /* Initialize x to 2 */
14         printf("second x = %4d\n", x);
15     }
16     printf("first  x = %4d\n", x);
17
18     return(0);
19 }
```

The program produces the following output:

```

second x =    2
first  x =    1
```

The function `main` defines two variables that are named `x`. The definition of `x` on line 8 retains storage while `main` is running. However, because the definition of `x` on line 13 occurs within a nested block, line 14 recognizes `x` as the variable defined on line 13. Because line 16 is not part of the nested block, the compiler recognizes `x` as the variable defined on line 8.

## Related Information

- “Block Scope Data Declarations” on page 70
- “File Scope Data Declarations” on page 71
- “Storage Class Specifiers” on page 73
- “Type Specifiers” on page 85

## break

A *break statement* lets you end an iterative statement (do, for, while) or a switch statement, and exit from it at any point other than the logical end.

A break statement has the form:

```
►►—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 only appear in the body of an iterative statement or a switch statement.

## Examples

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 OS/390 C/C++ processes the break statement, the inner loop ends and control returns to the outer loop.

### CBC3RAA2

```
/**
** 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>
#include <ctype.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 (pointer = strings[i]; *pointer != '\0'; ++pointer) /* for each character */
        {                               /* if a number */
            if (isnum(*pointer))
                break;
            letter_count++;
        }
    printf("letter count = %d\n", letter_count);

    return(0);
}

```

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.

### CBC3RAA

```

/**
 ** This example shows a switch statement with break statements.
 **/

#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 Information

- “do” on page 203
- “for” on page 206
- “switch” on page 212
- “while” on page 216

## 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.

A continue statement has the form:

```
▶▶—continue—;—————▶▶
```

The continue statement ends the processing of the action part of an iterative (do, for, or while) statement. It also 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. It then moves 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.

## Examples

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.

### CBC3RAA3

```
/**
 ** 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 it encounters the `'\0'` escape sequence.

### CBC3RAA4

```
/**
 ** 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>
#include <ctype.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 (pointer = strings[i]; *pointer != '\0'; ++pointer) /* for each character */
        {                                     /* if a number */
            if (isdigit(*pointer))
                continue;
            letter_count++;
        }
    printf("letter count = %d\n", letter_count);

    return(0);
}
```

The program produces the following output:

```
letter count = 5
```

Compare this program with the program in 200, which shows the use of the `break` statement to perform a similar function.

## Related Information

- “do”
- “for” on page 206
- “while” on page 216

---

## do

A *do statement* repeatedly runs a statement until the test expression evaluates to 0. Because of the order of processing, OS/390 C/C++ runs the statement at least once.

A *do statement* has the form:

```
do—statement—while—(—expression—);—
```

## do

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 end the processing of a do statement, even when the while clause does not evaluate to 0.

## Example

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. The example contains error-checking code to verify that the user entered an integer value and to clear the input stream if an error occurs.

### CBC3X07E

```
/**
 ** This example illustrates the do statement.
 **/

#include <iostream.h>
void main()
{
    int reply1;
    char c;
    do
    {
        cout << "Enter a 1: ";
        cin >> reply1;
        if (cin.fail())
        {
            cerr << "Not a valid number." << endl;
            // Clear the error flag.
            cin.clear(cin.rdstate() & ~ios::failbit);
            cin.ignore(cin.rdbuf()->in_avail());
        }
    }
    while (reply1 != 1);
}
```

## Related Information

- “break” on page 200
- “continue” on page 202
- “while” on page 216

## Expression

An *expression statement* contains an expression. The expression can be null. “Chapter 6. Expressions and Operators” on page 133 describes expressions.

An expression statement has the form:



An expression statement evaluates the given expression. Use it to assign the value of the expression to a variable or to call a function.

### Examples

```
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 */
```

### Resolving Ambiguous Statements in C++

There are situations in C++ where a statement can be parsed as both a declaration and as an expression. Specifically, a declaration can look like a function call in certain cases. The compiler resolves these ambiguities by applying the following rules to the *whole* statement:

- If the compiler can parse the statement as a declaration but there are no declaration specifiers, and the statement is inside the body of a function. The compiler assumes the statement is an expression.

The following statement, for example, is a declaration at file scope of the function `f()` that returns type `int`. There is no declaration specifier and `int` is the default, but at function scope this is a call to `f()`:

```
f();
```

- In every other case, if the compiler can parse the statement as a declaration, it assumes the statement is a declaration. The following statement, for example, is a declaration of `x` with redundant parentheses around the declarator, not a function-style cast of `x` to type `int`:

```
int(x);
```

In some cases, C++ syntax does not distinguish between expression statements and declaration statements. The ambiguity arises when an expression statement has a function-style cast as its leftmost subexpression. (Note that, because C does not support function-style casts, this ambiguity does not occur in C programs.) If the compiler can interpret the statement both as a declaration and as an expression, it interprets the statement as a declaration statement.

**Note:** The ambiguity is resolved only on a syntactic level. The resolution does not use the meaning of the names, except to assess whether or not they are type names.

The compiler resolves the following expressions into expression statements because the ambiguous subexpression is followed by an assignment or an operator. In these expressions, `type_spec` can be any type specifier:

## Expression

```
type_spec(i)++;           // expression statement
type_spec(i,3)<<d;        // expression statement
type_spec(i)->l=24;       // expression statement
```

In the following examples, the ambiguity cannot be resolved syntactically, and the compiler interprets the statements as declarations. `type_spec` is any type specifier:

```
type_spec(i)(int);        // declaration
type_spec(j)[5];          // declaration
type_spec(m) = { 1, 2 };  // declaration
type_spec(*k) (float(3)); // declaration
```

The last statement above causes a compile-time error because you cannot initialize a pointer with a float value.

Any ambiguous statement that is not resolved by the above rules is by default a declaration statement. All of the following are declaration statements:

```
type_spec(a);             // declaration
type_spec(*b)();          // declaration
type_spec(c)=23;          // declaration
type_spec(d),e,f,g=0;     // declaration
type_spec(h)(e,3);        // declaration
```

C++ resolves another ambiguity between expression statements and declaration statements by requiring an explicit return type for function declarations within a block:

```
a();           // declaration of a function returning int
               // and taking no arguments
void func()
{
    int a();    // declaration of a function
    int b;      // declaration of a variable
    a();        // expression-statement calling function a()
    b;          // expression-statement referring to a variable
}
```

The last statement above does not produce any action. It is semantically equivalent to a null statement. However, it is a valid C++ statement.

---

## 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.

A *for statement* has the form:

```
►—for—( ————— ; ————— ; ————— ) —————►
      [expression1]  [expression2]  [expression3]

►—statement——————►
```

*expression1*

Is the *initialization expression*. OS/390 C/C++

evaluates it only before it processes the statement 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.

*expression2*

Is the *controlling part*. OS/390 C/C++ evaluates it before each iteration of the statement. It must evaluate to a scalar type.

If it evaluates to 0 (zero), the statement is not processed and control moves to the next statement following the for statement. If *expression2* does not evaluate to 0, OS/390 C/C++ processes the statement. If you omit *expression2*, it is as if the expression had been replaced by a nonzero constant. In addition, the for statement is not terminated by failure of this condition.

*expression3*

OS/390 C/C++ evaluates this after each iteration of the statement. 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.

**C++ Note:** In C++ programs, you can also use *expression1* to declare a variable as well as initialize it. If you declare a variable in this expression, the variable has the same scope as the for statement and is not local to the for statement.

## Examples

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, it increments count.

```
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

```
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. The first comma in the for expression is a punctuator for a declaration. It declares and initializes two integers, *i*, and *j*. The second comma, a comma operator, allows the program to increment both *i* and *j* at each step through the loop.

```
for (int i = 0, j = 50; i < 10; ++i, j += 50)
{
    cout << "i = " << i << "and j = " << j << endl;
}
```

The following example shows a nested for statement. It prints the values of an array that has the dimensions [5][3]:

```
for (row = 0; row < 5; row++)
    for (column = 0; column < 3; column++)
        printf("%d\n", table[row][column]);
```

OS/390 C/C++ processes the outer statement 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. It executes the statement of the inner for statement 3 times. The inner statement is executed as long as the value of *column* is less than 3.

## Related Information

- “break” on page 200
- “continue” on page 202

---

## goto

A *goto statement* causes your program to unconditionally transfer control to the statement that is associated with the label that is specified on the goto statement.

A goto statement has the form:

```
►►—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. Thus, the result is undefined. The label must appear in the same function as the goto.

If your program exits an active block by using a goto statement, OS/390 C/C++ destroys any local variables when it transfers control from that block.

## Example

The following example shows a goto statement that is used to jump out of a nested loop. You can write this function without using a goto statement.

### CBC3RAA6

```
/**
 ** This example shows a goto statement that is used to
 ** jump out of a nested loop.
 **/

#include <stdio.h>
void display(int matrix[3][3]);

int main(void)
{
    int matrix[3][3]={1,2,3,4,5,2,8,9,10};
    display(matrix);
    return(0);
}

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");
}
```

---

## if

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.

An if statement has the form:

```

▶▶ if (—expression—) —statement—
    |
    | else —statement—
    |
▶▶
```

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.

## if

### Examples

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;
cout << "salary is " << salary << endl;
```

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, you may have to use braces. The braces force a particular else clause to associate with the correct if statement. In this example, omitting the braces causes 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. OS/390 C/C++ performs the tests in order of their appearance. If it evaluates one test to a nonzero value, OS/390 C/C++ runs the statement, and ends the entire if statement.

```
if (value > 0)
    ++increase;
else if (value == 0)
    ++break_even;
else
    ++decrease;
```

---

## null

The *null statement* performs no operation. It has the form:

```
▶▶;—————▶▶
```

A null statement can hold the label of a labeled statement or complete the syntax of an iterative statement.



## Example

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)
    ;
```

You can use a `null` statement when you require a label before the end of a block statement, for example:

```
void func(void) {
    if (error_detected)
        goto depart;
    /* further processing */
    depart:: /* null statement required */
}
```

---

## return

A *return statement* ends the processing of the current function and returns control to the caller of the function.

A return statement has the form:

```
→ return expression ; →
```

A `return` statement in a function is optional. The compiler issues a warning if it does not find a `return` statement in a function that is declared with a return type. If the compiler reaches the end of a function without encountering a `return` statement, it passes control to the caller. The compiler passes this control as if it had encountered a `return` statement without an expression. A function can contain multiple `return` statements.

## Value of a return Expression and Function Value

If an expression is present on a `return` statement, OS/390 C/C++ returns the value of the expression to the caller. If the data type of the expression is different from the function return type, OS/390 converts the return value. It performs this conversion as if the value of the expression was assigned to an object with the same function return type.

If a `return` statement does not contain an expression, the value of the `return` statement is undefined. If a `return` statement in a function declared with a return type that is not `void` does not contain an expression, an error message is issued. The result of calling the function is unpredictable, for example:

```
int func1()
{
    return;
}
int func2()
{
    return (4321);
}
```

## return

```
void main() {  
    int a=func1(); // result is unpredictable!  
    int b=func2();  
}
```

You cannot use a return statement with an expression when you declare the function as returning type void.

**C++ Note:** In C++, if a function returns a class object with constructors, OS/390 C/C++ may construct a temporary class object. The temporary object is not in the scope of the function that returns the temporary object, but is local to the caller of the function.

When OS/390 C/C++ returns a function, it destroys all temporary local variables. If local class objects with destructors exist, OS/390 C/C++ calls destructors. For more details, see “Temporary Objects” on page 333.

## Examples

```
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 Information

- “Chapter 8. Functions” on page 173
- “Expression” on page 205

---

## 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:

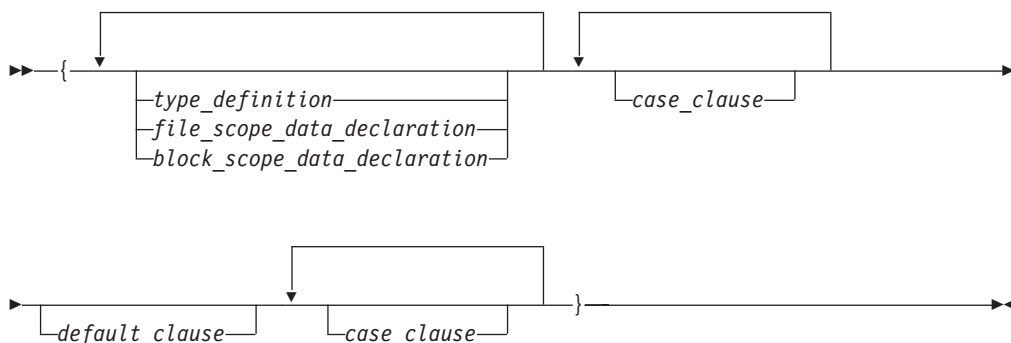
- A case label
- An optional default label
- A case expression
- A list of statements

If the value of the switch expression equals one of the case expression values, OS/390 C/C++ processes the statements that follow that case expression. If not, it processes any default label statements.

A switch statement has the form:

►► switch ( *expression* ) *switch\_body* ►►

You enclose the *switch body* in braces. The switch body 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* which is followed by any number of statements. A case clause has the form:

►► *case\_label* *statement* ►►

A *case label* contains the word *case*, followed by an integral constant expression and a colon. You can put multiple case labels anywhere that you can put one case label. A case label has the form:

►► *case* *integral\_constant\_expression* : ►►

A *default clause* contains a default label that is 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. A *default\_clause* has the form:

►► *case\_label* default : *case\_label* *statement* ►►

## switch

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. You can refer to this expression as the *switch expression*.

OS/390 C/C++ compares the value of the switch expression with the value of the expression in each case label. If it finds a matching value, it passes control 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 it does not find a matching value, and there is no default label anywhere in the switch body, it does not process any part of the switch body.

OS/390 C/C++ passes control to a statement in the switch body. It passes control out of the switch body only when it encounters a break statement, or encounters the last statement in the switch body.

If necessary, OS/390 C/C++ performs an integral promotion on the controlling expression. It also converts all expressions in the case statements 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. However, the compiler does not initialize auto and register variables at the beginning of a switch body.

**C++ Note:** You can have declarations in the body of the switch statement. In C++, you cannot normally transfer control over a declaration containing an initializer. However, you can transfer control if the declaration is located in an inner block that is completely bypassed by the transfer of control. You must contain all declarations within the body of a switch statement that contains initializers in an inner block.

## Examples

The following switch statement contains several case clauses and one default clause. Each clause contains a function call and a break statement. The break statements prevent control from passing down through each statement in the switch body.

If the switch expression evaluated to '/', the switch statement would call the function divide. Control would then pass to the statement following the switch body.

```
char key;

cout << "Enter an arithmetic operator\n");
cin >> key;
```

```

switch (key)
{
    case '+':
        add();
        break;

    case '-':
        subtract();
        break;

    case '*':
        multiply();
        break;

    case '/':
        divide();
        break;

    default:
        cout << "The key you pressed is not valid\n";
        break;
}

```

If the switch expression matches a case expression, OS/390 C/C++ processes the statements following the case expression. It processes these statements until it encounters a break statement, or reaches the end of the switch body. In the following example, break statements are not present. If the value of text[i] is equal to 'A', the compiler increments all three counters. 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 statement performs the same statements for more than one case label:

### CBC3RABI

```

/**
 ** This example contains a switch statement that performs
 ** the same statement for more than one case label.
 **/

#include <stdio.h>

int main(void)
{
    int month;

    /* Read in a month value */
    printf("Enter month: ");
    scanf("%d", &month);
}

```

## switch

```
/* Tell what season it 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);
}

return(0);
}
```

If the expression `month` has the value 3, OS/390 C/C++ passes control to the following statement:

```
printf("month %d is a spring month\n", month);
```

The `break` statement passes control to the statement following the `switch` body.

## Related Information

- “`break`” on page 200.

---

## while

A *while* statement repeatedly runs the body of a loop until the controlling expression evaluates to 0.

A *while* statement has the form:

►►—while—(—expression—)—statement—►►

OS/390 C/C++ evaluates the expression 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, OS/390 C/C++ processes the loop body. 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.

## Example

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.

### CBC3RAA7

```
/**
 ** This example illustrates the while statement.
 **/

#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 Information

- “break” on page 200
- “goto” on page 208
- “return” on page 211

**while**



---

## Chapter 10. Preprocessor Directives

This chapter describes the following topics on C preprocessor directives:

- “Preprocessor Overview”
- “Preprocessor Directive Format” on page 220
- “Phases of Preprocessing” on page 220
- “Macro Definition and Expansion (#define)” on page 221
- “Scope of Macro Names (#undef)” on page 225
- “Single Number Sign Operator (#)” on page 225
- “Macro Concatenation with the ## Operator” on page 226
- “Preprocessor Error Directive (#error)” on page 228
- “File Inclusion (#include)” on page 228
- “Predefined Macro Names” on page 229
- “Conditional Compilation Directives” on page 237
- “Line Control (#line)” on page 241
- “Null Directive (#)” on page 242
- “Pragma Directives (#pragma)” on page 243

---

### Preprocessor Overview

*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 source line number of the next line, and change the file name of the current file.

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

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

The following directives control the preprocessor:

<code>#define</code>	Defines a preprocessor directive.
<code>#undef</code>	Removes a preprocessor macro definition.
<code>#error</code>	Defines text for a compile-time error message.
<code>#include</code>	Inserts text from another source file.
<code>#if</code>	Conditionally suppresses portions of source code, depending on the result of a constant expression.
<code>#ifdef</code>	Conditionally includes source text if you define a macro name.
<code>#ifndef</code>	Conditionally includes source text if you do not define a macro name.

## Preprocessor Overview

<code>#else</code>	Conditionally includes source text if the previous <code>#if</code> , <code>#ifdef</code> , <code>#ifndef</code> , or <code>#elif</code> test fails.
<code>#elif</code>	Conditionally includes source text if the previous <code>#if</code> , <code>#ifdef</code> , <code>#ifndef</code> , or <code>#elif</code> test fails, depending on the value of a constant expression.
<code>#endif</code>	Ends conditional text.
<code>#line</code>	Supplies a line number for compiler messages.
<code>#pragma</code>	Specifies implementation-defined instructions to the compiler.

“Preprocessor Directive Format” defines the format of a preprocessor directive .

---

## Preprocessor Directive Format

Preprocessor directives begin with the `#` token that is 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 you can separate it 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.

---

## Phases of Preprocessing

Preprocessing appears as if it occurs in several phases.

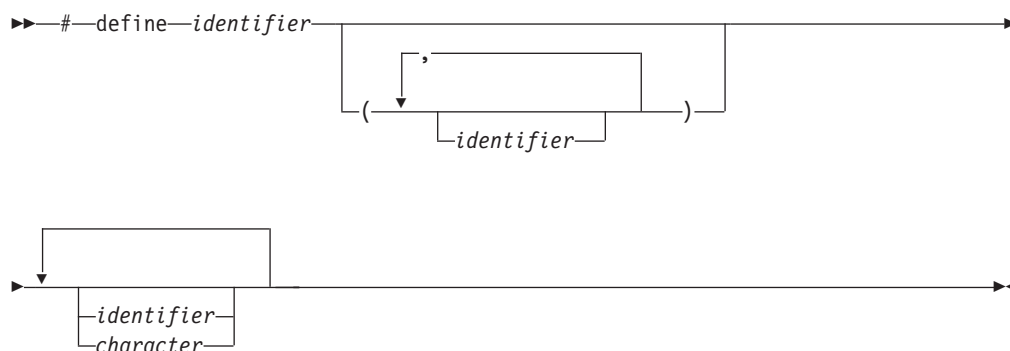
1. It introduces new-line characters as needed to replace system-dependent end-of-line indicators, and performs any other system-dependent character-set translations. It replaces trigraph (C and C++:) and digraph (C++ only) sequences with equivalent single characters.
2. It deletes each `\` (backslash) that is followed by a new-line character pair. It appends the next source line to the line that contained the sequence.
3. It decomposes the source text 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. It executes preprocessing directives, and expands macros.
5. It replaces escape sequences in character constants and string literals by their equivalent values.
6. It concatenates adjacent string literals.

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

## Macro Definition and Expansion (#define)

A *preprocessor define directive* directs the preprocessor to replace all subsequent occurrences of a macro with specified replacement tokens.

A preprocessor #define directive has the form:



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
```

Consider the following statement:

```
int arry[COUNT];
```

If the above statement appears after this definition and in the same file as the definition, the preprocessor changes the statement to the following statement, in the output of the preprocessor:

```
int arry[1000];
```

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. The preprocessor then replaces COUNT + 100 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
```

## #define

Identifiers that are partially built from a macro expansion may not be produced. Therefore, the following example does not produce the identifier `abcdefg`, and results in a syntax error:

```
#define d efg
abcd
```

## Function-Like Macros

### Function-like macro definition:

An identifier followed by a parameter list in parentheses and the replacement tokens. OS/390 C/C++ imbeds the parameters 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.

### Function-like macro invocation:

An identifier followed by a list of arguments in parentheses. A comma must separate each argument. Once the preprocessor identifies a function-like macro invocation, it substitutes an argument. It replaces a parameter in the replacement code by the corresponding argument. The preprocessor completely replaces any macro invocations that are contained in the argument itself, before the argument replaces its corresponding parameter in the replacement code.

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 would cause 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 follows:

```
c = (x + y);
c = d * (x + y);
```

Use parentheses to ensure correct evaluation of replacement text. For example, consider the following definition:

```
#define SQR(c) ((c) * (c))
```

The above definition requires parentheses around each parameter `c` in the definition. This way, it can correctly evaluate an expression such as the following one:

```
y = SQR(a + b);
```

The preprocessor expands this statement as follows:

```
y = ((a + b) * (a + b));
```

Without parentheses in the definition, the preprocessor does not preserve the correct order of evaluation, and its output is:

```
y = (a + b * a + b);
```

See “Operator Precedence and Associativity” on page 133, and “Parenthesized Expressions ( )” on page 137 for more information about using parentheses.

The preprocessor converts the arguments of the single number sign operator (#) and the double number sign operator ## *before* it replaces parameters in a function-like macro.

The number of arguments in a macro invocation must be the same as the number of parameters in the corresponding macro definition.

Commas in the macro invocation argument list do not act as argument separators when they are:

- In character constants
- In string literals
- Surrounded by parentheses.

Once defined, a preprocessor identifier remains defined and in scope independent of the scoping rules of the language. The scope of a macro definition begins at the definition and it does not end until it encounters a corresponding #undef directive. If there is no corresponding #undef directive, the scope of the macro definition lasts until the end of the compilation unit.

The preprocessor does not fully expand a recursive macro. For example, consider the following definition:

```
#define x(a,b) x(a+1,b+1) + 4
```

And assume the following macro definition:

```
x(20,10)
```

The above macro definition expands to the following, rather than trying to expand the macro x over and over, within itself:

```
x(20+1,10+1) + 4
```

After the preprocessor expands the macro x, the macro is a call to function x().

You do not require a definition to specify replacement tokens. The following definition removes all instances of the token debug from subsequent lines in the current file:

```
#define debug
```

You can change the definition of a defined identifier or macro with a second preprocessor #define directive. However, the second preprocessor #define directive must be preceded by a preprocessor #undef directive. This is described in “Scope of Macro Names (#undef)” on page 225. The #undef directive nullifies the first definition, so that you can use the same identifier in a redefinition.

Within the text of the program, the preprocessor does not scan character constants or string constants for macro invocations.

## #define

### Examples of #define Directives

The following program contains two macro definitions and a macro invocation that refers to both of the defined macros:

#### CBC3RAA8:

```
/**
 ** This example illustrates #define directives.
 ** Example CBC3RAA9 shows the effect of preprocessor
 ** macro replacement on this program.
 **/

#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 the preprocessor interprets this program, it is replaced by code that is equivalent to the following:

#### CBC3RAA9:

```
/**
 ** This example shows the effect of the preprocessor macro
 ** replacement on the program in example CBC3RAA8.
 **/

#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
```

## Scope of Macro Names (#undef)

A *preprocessor undef directive* causes the preprocessor to end the scope of a preprocessor definition.

A preprocessor #undef directive has the form:

```
▶▶—#—undef—identifier—————▶▶
```

If you have not currently defined the identifier as a macro, the preprocessor ignores #undef.

## Examples of #undef Directives

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
```

The preprocessor does not replace any occurrences of the identifiers BUFFER and SQR that follow these #undef directives 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.

## Single Number Sign Operator (#)

The # (single number sign) operator converts a parameter of a function-like macro into a character string literal. For example, consider defining the macro ABC by using the following directive:

```
#define ABC(x)  #x
```

The preprocessor expands all subsequent invocations of the macro ABC into a character string literal that contains the argument that is passed to ABC. For example:

Invocation	Result of Macro Expansion
ABC(1)	"1"
ABC>Hello there)	"Hello there"

Note that you can represent the single number sign character # by the trigraph ??=.

Do not confuse the # operator with the null directive documented in “Null Directive (#)” on page 242.

Use the # operator in a function-like macro definition according to the following rules:

## # Operator

- The preprocessor converts a parameter that follows the # operator in a function-like macro into a character string literal that contains the argument that is passed to the macro.
- The preprocessor deletes white-space characters that appear before or after the argument that is passed to the macro.
- The preprocessor uses a single space character to replace multiple white-space characters that are imbedded within the argument that is passed to the macro.
- If the argument that is passed to the macro contains a string literal, and if a \ (backslash) character appears within the literal, the preprocessor inserts a second \ character before the original one when it expands the macro.
- 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.

See “Function-Like Macros” on page 222 for more information about function-like macros.

## Examples of the # Operator

The following examples demonstrate the use of the # operator:

```
#define STR(x)      #x
#define XSTR(x)     STR(x)
#define ONE        1
```

Invocation	Result of Macro Expansion
STR(\n " \n" ' \n')	"\n \"\\n\" '\\\\n\\' "
STR(ONE)	"ONE"
XSTR(ONE)	"1"
XSTR("hello")	"\\\"hello\\\""

## Related Information

- “Macro Definition and Expansion (#define)” on page 221
- “Scope of Macro Names (#undef)” on page 225

---

## Macro Concatenation with the ## Operator

The double number sign operator (##) concatenates two tokens in a macro invocation (text or arguments), that a macro definition contains.

Consider a macro, XY, which is defined using the following directive:

```
#define XY(x,y)      x##y
```

The preprocessor concatenates the last token of the argument for x with the first token of the argument for y.



For example,

Invocation	Result of Macro Expansion
XY(1, 2)	12
XY(Green, house)	Greenhouse

Note that you can represent the # character by the trigraph `??=`.

## Double Number Sign Operator (##)

Use the double number sign 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 preprocessor concatenates the last token of the item in front of the ## operator with the first token of the item that follows the ## operator.
- Concatenation takes place before the preprocessor expands any macros in arguments.
- If the result of a concatenation is a valid macro name, the result is available for further replacement. It is available even if it appears in a context in which it is not normally available.
- 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.

### Examples of the ## Operator

The following examples demonstrate the use of the ## operator:

```
#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
```

Invocation	Result of Macro Expansion
ArgArg(lady, bug)	ladybug
ArgText(con)	conTEXT
TextArg(book)	TEXTbook
TextText	TEXTtext
ArgArg(Jitter, bug)	3

### Related Information

- “Macro Definition and Expansion (#define)” on page 221

## Preprocessor Error Directive (#error)

A *preprocessor error directive* causes the preprocessor to generate an error message and causes the compilation to fail.

The #error directive has the form:

```

>> #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. The directives prevent code generation if a section of the program is reached that should be bypassed.

For example, consider the following directive:

```
#error Error in TESTPGM1 - This section should not be compiled
```

The above directive generates the following error message:

```
Error in TESTPGM1 - This section should not be compiled
```

## Related Information

- “Conditional Compilation Directives” on page 237

## File Inclusion (#include)

A *preprocessor include directive* causes the preprocessor to replace the directive with the contents of the specified file.

A preprocessor #include directive has the form:

```

>> #include "file_name"
        //
        <file_name>
  
```

You can specify an OS/390 data set or an HFS file for *filename*. Use double slashes (//) before the *filename* to indicate that the file is an OS/390 data set. Use a single slash (/) anywhere in the *filename* to indicate an HFS file. See the *OS/390 C/C++ User's Guide* for more information on specifying include file names.

The preprocessor resolves macros that are contained in an #include directive. After macro replacement, the resulting token sequence must consist of a file name that is enclosed in either double quotation marks, or the characters < and >.

For example:

```
#define MONTH <july.h>
#include MONTH
```

If you enclose the file name in double quotation marks ("), the preprocessor searches the directories or libraries that contain the user source files. It then searches a standard or specified sequence of places, until it finds the specified file. For example:

```
#include "payroll.h"
```

If you enclose the file name in the characters < and >, the preprocessor searches only the standard or specified places for the specified file. For example:

```
#include <stdio.h>
```

The new-line and > characters cannot appear in a file name that is delimited by < and >. The new-line and " (double quotation marks) characters cannot appear in a file name that is delimited by double quotation marks. However, the > character can appear in such a file name.

For more information about include file search paths and compiler options, see the *OS/390 C/C++ User's Guide*.

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"
```

In the following example, a #define combines several preprocessor macros to define a macro. This macro 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> */
.
.
.
```

---

## Predefined Macro Names

OS/390 C/C++ provides the following predefined macro names:

- "ANSI/ISO Standard Predefined Macro Names" on page 230.
- "OS/390 C/C++ Predefined Macro Names" on page 231.

These predefined names cannot be subject to a #define or #undef preprocessor directive.

## ANSI/ISO Standard Predefined Macro Names

Both C and C++ provide the following predefined macro names as specified in the ANSI/ISO C language standard:

Macro Name	Description
<code>__DATE__</code>	<p>A character string literal that contains 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> <pre>"Mmm dd yyyy"</pre> <p>where:</p> <p>Mmm     Represents the month in an abbreviated form (Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, or Dec).</p> <p>dd       Represents the day. If the day is less than 10, the first d is a blank character.</p> <p>yyyy     Represents the year.</p>
<code>__FILE__</code>	<p>A character string literal that contains 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. You can set it with the <code>#line</code> directive, described in “Line Control (#line)” on page 241.</p>
<code>__LINE__</code>	<p>An integer that represents 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. You can set it with the <code>#line</code> directive, described in “Line Control (#line)” on page 241.</p>
<code>__STDC__</code>	<p>The integer 1 (one) indicates that the C compiler conforms to the ANSI/ISO standard. For C programs, the compiler sets this macro to the integer 1 (one) to indicate that the C compiler conforms to the ANSI/ISO standard. For C++ programs, the compiler does not define this macro. The macro has the integer value 0 when you use it in a <code>#if</code> statement. This indicates that the C++ language is not a proper superset of C, and that the compiler does not conform to ANSI/ISO C.</p> <p>For more information on how C++ differs from ANSI/ISO C, see “Appendix A. C and C++ Compatibility” on page 401. The integer 0 (zero) indicates that C++ does not conform to the ANSI/ISO C language standard.</p>

## Predefined Macro Names

**Note:** If you set the language level to anything other than ANSI, this macro is undefined.

`__TIME__`

A character string literal that contains the time when the source file was compiled.

The value of `__TIME__` changes as the compiler processes any include files that are part of your source program. The time is in the form:

"hh:mm:ss"

where:

hh      Represents the hour.

mm      Represents the minutes.

ss      Represents the seconds.

**Note:** The MVS command SET DATE does not affect the value returned by this macro, or the value returned by the `time()` function.

`__cplusplus`

For C++ programs, the preprocessor sets this macro to the integer 1, which indicates that the compiler is a C++ compiler. Note that this macro name has no trailing underscores.

## OS/390 C/C++ Predefined Macro Names

OS/390 C/C++ provides the following predefined macros. It defines the value of all these macros when you use the corresponding `#pragma` directive or compiler option.

### Macro Name

### Description

`__ANSI__`

C Only. This macro allows only language constructs that conform to ANSI/ISO C standard. It is defined as 1 by using the `#pragma langlvl(ansi)` directive or `LANGLVL(ANSI)` compile option.

`__BFP__`

This macro allows Language Environment headers to map functions such as `sin(x)` to appropriate LE calls. OS/390 C/C++ sets this macro to 1 when you specify binary floating point (BFP) mode by using the `FLOAT(IEEE)` compiler option.

`__EXTENDED__`

This macro allows additional language constructs that are provided by the OS/390 C/C++ implementation. It is defined by using the `#pragma langlvl(extended)` directive or `LANGLVL(EXTENDED)` compile option.

`__SAA__`

C Only. This macro allows only language constructs that conform to the most recent level of SAA C standards. It is defined as 1 by using the `#pragma langlvl(saa)` directive or `LANGLVL(SAA)` compile option.

`__SAA_L2__`

C Only. This macro allows only language

## Predefined Macro Names

`__CODESET__`

constructs that conform to SAA Level 2 C standards. It is defined as 1 by using the `#pragma langlvl(saa12)` directive or `LANGLVL(SAAL2)` compile option.

A string literal that represents the character code set of the `LOCALE` compile option. If you do not use the `LOCALE` compile option, the macro is undefined.

`__COMPAT__`

C++ Only. The macro is defined as 1 by using the `LANGLVL(COMPAT)` compile option or the `#pragma langlvl(compat)` directive for C++ language files. It indicates that the compiler allows language constructs that are compatible with earlier versions of the C++ language.

`__COMPILER_VER__`

The compiler version. The format of the version number that is provided by the macro is hex *PVRRMMMM*:

- *P* represents the compiler product
  - 0 for C/370
  - 1 for AD/Cycle C/370 and C/C++ for MVS/ESA
  - 2 for OS/390 C/C++
- *V* represents the version number
- *RR* represents the release number
- *MMMM* represents the modification number

In OS/390 C/C++ Version 2 Release 6, the value of the macro is `X'22060000'`.

`__COMMONC__`

C Only. Allows language constructs that are defined by XPG. The `__EXTENDED__` macro enables many of the constructs that `__COMMONC__` does. The compiler defines the `__COMMONC__` macro as 1 when you use the `#pragma langlvl(commonc)` directive or the `LANGLVL(COMMONC)` compile-time option.

`__DLL__`

This macro allows you to write conditional code that depends upon whether or not you have compiled your program as DLL code. For C++, the preprocessor always defines the macro as 1. For C, the preprocessor defines the macro as 1 if you specify the `DLL` compiler option. Otherwise, it is undefined.

`__FILETAG__`

A string literal that represents the character code set of the `filetag` pragma associated with the current file. If no `filetag` pragma is present, the macro is undefined.

The value of `__FILETAG__` changes as the compiler processes include files that are part of your source program.

`__FUNCTION__`

A character string that contains the name of the function that the OS/390 C/C++ is currently compiling.

`__HHW_370__`

Indicates that the host hardware is System/370.

## Predefined Macro Names

`__HOS_MVS__`

The preprocessor predefines this macro to a value of 1 for C and C++ compilers on System/370.

Indicates that the host operating system is OS/390. OS/390 C/C++ predefines this macro to have a value of 1.

`__IBMC__`

C only. This macro indicates the version number of the OS/390 C compiler. The format of the version number that is provided by the macro is integer *PVRRM*:

- *P* represents the compiler product
  - 0 for C/370
  - 1 for AD/Cycle C/370 and C/C++ for MVS/ESA
  - 2 for OS/390 C/C++
- *V* represents the version number
- *RR* represents the release number
- *M* represents the modification number

`__IBMCPP__`

In OS/390 C/C++ Version 2 Release 6, the value of the macro is 22060.

C++ Only. This macro indicates the version number of the OS/390 C++ compiler. The format of the version number that is provided by the macro is integer *PVRRM*:

- *P* represents the compiler product
  - 0 for C/370
  - 1 for AD/Cycle C/370 and C/C++ for MVS/ESA
  - 2 for OS/390 C/C++
- *V* represents the version number
- *RR* represents the release number
- *M* represents the modification number

`__LOCALE__`

In OS/390 C/C++ Version 2 Release 6, the value of the macro is 22060.

This macro contains a string literal that represents the locale of the `LOCALE` compile option. If you do not supply a `LOCALE` compile option, the macro is undefined.

The following example illustrates how to set the runtime locale to the compile-time locale:

```
main()
{
    setlocale(LC_ALL, __LOCALE__);
    :
}
```

`__LONGNAME__`

For C, the integer 1 indicates that you have specified the `LONGNAME` compile option or pragma. Otherwise the macro is undefined. For C++, the value of `__LONGNAME__` is always 1, even if you specify `NOLONGNAME`.

## Predefined Macro Names

`__MVS__`

**C++ Note:** In C++, long names are always in the compilation unit. The `LONGNAME` compile option in C++ controls whether non-C++ names will be truncated and uppercased, or left alone. You can use this option to interface with existing C code that was compiled with `NOLONGNAME`, so that the names match.

For OS/390 C/C++ programs, OS/390 C/C++ sets this macro to 1, which indicates that you are compiling the program on OS/390.

`__SOM_ENABLED__`

**Note:** This macro is the same as `__HOS_MVS__`.

This macro is defined when you use the SOM compile options. It indicates that OS/390 C/C++ supports native SOM. This option turns on implicit SOM mode, and includes the file `<som.hh>`.

`__STRING_CODE_SET__`

This macro allows you to change the code page that the compiler uses for character string literals (character data enclosed in double quotation marks). To use this macro, you must specify it with the `DEFINE` compiler option. The following example shows you how to do this:

```
DEFINE(__STRING_CODE_SET__="ISO8859-1")
```

This macro affects all source files that are processed within a compilation unit, including user header files, and system header files. All string literals within a compilation unit must use the same code page. Note that you can also use the `CONVLIT` compiler option instead of this macro. For more information on this option, see the *OS/390 C/C++ User's Guide*.

The macro does not affect the following types of string literals:

- String literals that are used in `#include` directives
- String literals that are used in `#pragma` directives
- String literals that are used to specify linkage, such as `extern "C"` (C++ only)

The following restrictions apply to this macro:

- You cannot specify this macro if the SOM compiler option is in effect.
- You cannot specify this macro if you have also used predefined macros (such as `__TIMESTAMP__`) that return string literals.

`__TEMPINC__`

**C++ Only.** This macro indicates that the compiler is using the template-implementation file method of resolving template functions. It is defined as 1 if you are using the `TEMPINC` compile option.



## Predefined Macro Names

`__TARGET_LIB__`

The target library version. The format of the version number provided is hex *PVRRMMMM*:

- *P* represents the C or C/C++ library product
  - 0 for C/370
  - 1 for Language Environment/370 and Language Environment for MVS & VM
  - 2 for OS/390 Release 2 and later
- *V* represents the version number
- *RR* represents the release number
- *MMMM* represents the modification number.

In OS/390 C/C++ Version 2 Release 6, the value of the macro is X'22060000'.

`__THW_370__`

This macro indicates that the target hardware is System/370. OS/390 C/C++ predefines this macro to have a value of 1 for C and C++ compilers targeting System/370.

`__TIMESTAMP__`

A character string literal that contains the date and time when the source file was last modified.

The value of `__TIMESTAMP__` changes as the compiler processes any include files that are part of your source program. The date and time are in the form:

"Day Mmm dd hh:mm:ss yyyy"

where:

- |      |  |
|------|--|
| Day  | Represents the day of the week (Mon, Tue, Wed, Thu, Fri, Sat, or Sun).                                       |
| Mmm  | Represents the month in an abbreviated form (Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, or Dec). |
| dd   | Represents the day. If the day is less than 10, the first d is a blank character.                            |
| hh   | Represents the hour.   |
| mm   | Represents the minutes.  |
| ss   | Represents the seconds.  |
| yyyy | Represents the year.   |

This macro is available for Partitioned Data Sets (PDSs/PDSEs) and HFS source files only. For PDSE or PDS members, the ISPF timestamp for the member is used if present. For PDSE/PDS members with no ISPF timestamp, sequential datasets, or in stream source in JCL, OS/390 C/C++ returns a dummy timestamp. For HFS files, OS/390 C/C++ uses the system timestamp on an HFS source file. Otherwise, it returns a dummy timestamp, "Mon Jan 1 0:00:01 1990".

## Predefined Macro Names

`__TOS_MVS__`

This macro indicates that the target operating system is OS/390. OS/390 C/C++ predefines this macro to a value of 1.

`__370__`

This macro indicates that the program is compiled or targeted to run on System/370. OS/390 C/C++ predefines this macro to a value of 1 for backward compatibility with earlier releases. For current programs, use `__370__`.

**Note:** OS/390 C/C++ does not provide a `_LONG_LONG` predefined macro. If your code requires this macro, you can define it by using the `DEFINE` option under TSO or batch, or the `-D` option in an OS/390 UNIX environment.

## Examples of Predefined Macros

### CBC3X08A

```
/**
 ** 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
```

### CBC3X08B

```
/**
 ** 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(int, char **)
In function foo (int)
```

**CBC3X08C**

```

/**
 ** This example illustrates the __FUNCTION__ predefined macro
 ** in a C++ program with virtual functions.
 **/
#include <stdio.h>
class X { public: virtual void func() = 0;};

class Y : public X {
    public: void func() { printf("In function %s \n", __FUNCTION__);}
};

main() {
    Y aaa;
    aaa.func();
}

```

The output of this example is:

In function Y::func()

**Related Information**

- “Macro Definition and Expansion (#define)” on page 221
- “Scope of Macro Names (#undef)” on page 225
- “Line Control (#line)” on page 241

---

## Conditional Compilation Directives

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

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

The preprocessor conditional compilation directive spans several lines:

- The condition specification line
- Lines containing code that the preprocessor passes on to the compiler if the condition evaluates to a nonzero value (optional)
- The #else line (optional)
- Lines containing code that the preprocessor passes on to the compiler if the condition evaluates to zero (optional)
- The preprocessor #endif directive

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

You can nest conditional compilation directives. The following directives match the first #else with the #if directive.

## Conditional Compilation Directives

```
#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 that start on the line that follows the directive and ends at the next conditional compilation directive at the same nesting level.

The preprocessor processes directives in the order in which it encounters them. If an expression evaluates to zero, the preprocessor ignores the block that follows the directive.

Consider when the preprocessor ignores a block following a preprocessor directive. In that case, the tokens are examined only to identify preprocessor directives within that block so that the preprocessor can determine the conditional nesting level. It ignores all tokens other than the name of the directive.

The preprocessor processes the first block whose expression is nonzero only. It ignores the remaining blocks at that nesting level. Consider if the preprocessor has not processed the blocks at that nesting level and there is a `#else` directive. In that case, it processes the block following the `#else` directive. If it has not processed any of the blocks at that nesting level and there is no `#else` directive, the preprocessor ignores the entire nesting level.

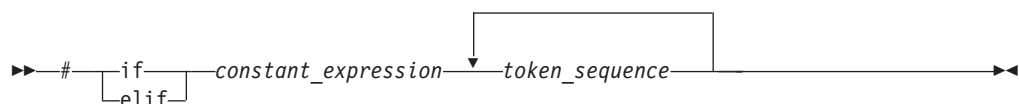
### **#if, #elif**

The `#if` and `#elif` directives compare the value of the expression to zero.

If the constant expression evaluates to a nonzero value, the preprocessor passes the tokens that immediately follow the condition to the compiler.

Consider when the expression evaluates to zero and the conditional compilation directive contains a preprocessor `#elif` directive. In that case, the preprocessor passes the source text located between the `#elif` and the next `#elif` or preprocessor `#else` directive 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:

- They must perform arithmetic using long int values.

## Conditional Compilation Directives

- 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 you have defined the *identifier* in the preprocessor; otherwise they evaluate to 0 (zero):

```
defined identifier  
defined(identifier)
```

For example:

```
#if defined(TEST1) || defined(TEST2)
```

**Note:** If you have not defined a macro, the preprocessor assigns a value of 0 (zero) to it. In the following example, `TEST` must be a macro identifier:

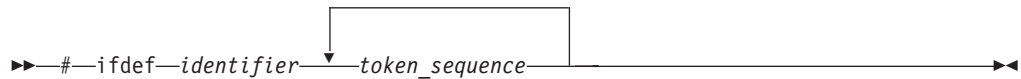
```
#if TEST >= 1  
    printf("i = %d\n", i);  
    printf("array[i] = %d\n", array[i]);  
#elif TEST < 0  
    printf("array subscript out of bounds \n");  
#endif
```

### #ifdef

The `#ifdef` directive checks for the existence of macro definitions.

If you have defined the identifier that is specified as a macro, the preprocessor passes the tokens that immediately follow the condition on to the compiler.

The preprocessor `#ifdef` directive has the form:



The following example defines `MAX_LEN` to be 75 if `EXTENDED` is defined for the preprocessor. Otherwise, the example defines `MAX_LEN` to be 50.

```
#ifdef EXTENDED  
#   define MAX_LEN 75  
#else  
#   define MAX_LEN 50  
#endif
```

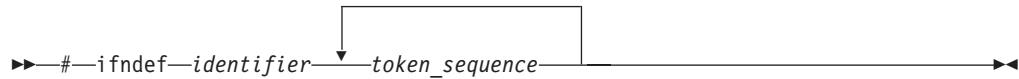
### #ifndef

The `#ifndef` directive checks for the existence of macro definitions.

If you have not defined the identifier that is specified as a macro, the preprocessor passes on the tokens that immediately follow the condition to the compiler.

The preprocessor `#ifndef` directive has the form:

## Conditional Compilation Directives



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, the example defines `MAX_LEN` to be 75.

```
#ifndef EXTENDED
#   define MAX_LEN 50
#else
#   define MAX_LEN 75
#endif
```

### **#else**

Consider when the condition specified in the `#if`, `#ifdef`, or `#ifndef` directive evaluates to 0, and the conditional compilation directive contains a `#else` directive. In that case, the preprocessor passes the source text located between the `#else` directive and the `#endif` directive to the compiler.

The preprocessor `#else` directive has the form:



### **#endif**

The preprocessor `#endif` directive ends the conditional compilation directive.

It has the form:



## Examples of Conditional Compilation Directives

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
```

The following program contains preprocessor conditional compilation directives:

### CBC3RABC

```
/**
 ** 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;

#ifdef TEST
        printf("i = %d\n", i);
        printf("array[i] = %d\n", array[i]);
    #endif

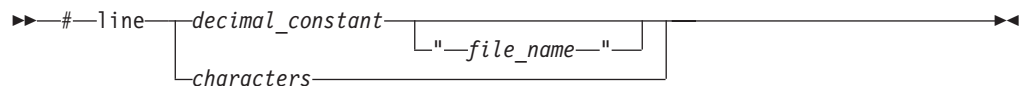
    }
    return(0);
}
```

---

## Line Control (#line)

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.

A preprocessor `#line` directive has the form:



In order for the compiler to produce meaningful references to line numbers in preprocessed source, the preprocessor inserts `#line` directives where necessary. For example, it inserts them at the beginning of and at the end of included text.

A file name specification that is 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 *file\_name* should be:

- A fully qualified sequential dataset
- A fully qualified PDS or PDSE member
- An HFS path name

The entire string is taken unchanged as the alternate source file name for the compilation unit (for example, for use by the debugger). Consider if you are using it to redirect the debugger to source lines from this alternate file. In this case, you

## #line

*must* ensure the file exists as specified and the line number on the #line directive matches the file contents. The compiler does not check this.

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 that is enclosed in double quotation marks.

If you do not specify *file\_name*, the preprocessor takes the line number to refer to the current source file.

**Note:** The compiler ignores #line directives when the EVENTS compiler option is in effect.

## Example of #line Directives

You can use #line control directives to make the compiler provide more meaningful error messages. The following program uses #line control directives to give each function an easily recognizable line number:

### CBC3RABD

```
/**
** This example illustrates #line directives.
**/

#include <stdio.h>
#define LINE200 200

int main(void)
{
    func_1();
    func_2();
}

#line 100
func_1()
{
    printf("Func_1 - the current line number is %d\n", __LINE__);
}

#line LINE200
func_2()
{
    printf("Func_2 - the current line number is %d\n", __LINE__);
}
```

This program produces the following output:

```
Func_1 - the current line number is 102
Func_2 - the current line number is 202
```

---

## Null Directive (#)

The *null directive* performs no action. It consists of a single # on a line of its own.

Do not confuse the null directive with the # operator or the character that starts a preprocessor directive.

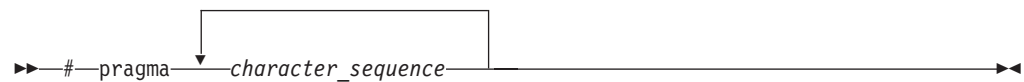


In the following example, if MINVAL is a defined macro name, the preprocessor takes no action. If MINVAL is not a defined identifier, the preprocessor defines MINVAL to 1.

```
#ifdef MINVAL
#
#else
#define MINVAL 1
#endif
```

## Pragma Directives (#pragma)

A *pragma* is an implementation-defined instruction to the compiler. It has the general form:



In the above syntax diagram, *character\_sequence* is a series of characters that gives a specific compiler instruction and arguments, if any.

The *character\_sequence* on a pragma is not subject to macro substitutions, unless otherwise stated.

You can specify more than one pragma construct on a single #pragma directive. The compiler ignores unrecognized pragmas.

The OS/390 C/C++ compiler recognizes the following pragmas:

chars	Sets the sign type of character data.
checkout	Controls the diagnostic messages that are generated by the C compiler CHECKOUT option, and the C++ compiler INFO option.
comment	Places a comment into the object module. Under some circumstances it places the comment in the load module as well. This pragma must appear before any C or C++ code.
convlit	Provides a means for changing the assumed code page for character string literals.
csect	Identifies the name for either the code, static, or test control section (CSECT). The IPA Link step does not use this name; it uses CSECT names that are specified in the IPA control file.
define	C++ Only. This pragma forces the definition of a template class without actually defining an object of the class.
disjoint	C Only. This pragma lists the identifiers that are not aliased to each other within the scope of their use.
environment	C Only. Use OS/390 C code as an assembler substitute.

## #pragma

export	Declares that an external function or variable is to be exported.
filetag	Specifies the code set in which the source code was entered.
hdrstop	Manually terminates the initial sequence of #include directives that are being considered for precompilation. This pragma must appear before any code.
implementation	C++ Only. This pragma tells the compiler the name of the file that contains the function template definitions. These definitions correspond to the template declarations in the include file that contains the pragma.
info	C++ Only. This pragma controls the diagnostic messages that are generated by the INFO compiler option.
inline	C Only. This pragma specifies that a C function is to be inlined.
isolated_call	Lists functions that do not alter data objects visible at the time of the function call.
langlvl	Selects the C or C++ language level for compilation.
linkage	C Only. This pragma identifies the linkage or calling convention that is used on a function call.
longname	Specifies that the compiler is to generate not-truncated and mixed case names in the object module that is produced by the compiler. It must appear before any code.
map	Tells the compiler to convert all references to an identifier to a new name.
margins	Specifies the columns in the input line to scan for input to the compiler.
noinline	Specifies that a C or C++ function is not to be inlined.
options	C Only. This pragma specifies options to the compiler in your source program.
pack	Specifies the alignment rules to use for the structures, unions, and classes that follow it.
page	C Only. This pragma skips pages of the generated source listing.
pagesize	C Only. This pragma sets the number of lines per page for the generated source listing.
priority	C++ Only. This pragma specifies the order in which OS/390 C/C++ initializes static objects at run time.
runopts	Specifies a list of run-time options for OS/390 C/C++ to use at execution time.

sequence	Defines the section of the input line that is to contain sequence numbers.
skip	C Only. This pragma skips lines of the generated source listing.
strings	Sets storage type for strings.
subtitle	C Only. This pragma places text on generated source listings.
target	C Only. This pragma specifies the operating system or run-time environment for which OS/390 C/C++ creates the object module. It must appear before any C code.
title	C Only. This pragma places text on generated source listings.
variable	Specifies that OS/390 C/C++ is to use the named object in a reentrant or non-reentrant fashion.
wsizeof	Specifies the behavior of the sizeof operator either to that prior to the C/C++ Version 1 Release 3 compilers, or to the OS/390 C/C++ compiler.

The following pragmas are used in Direct-to-SOM applications and are valid in OS/390 C++ only. Refer to the *OS/390 C/C++ Programming Guide* for information about these pragmas.

- SOM
- SOMAsDefault
- SOMAttribute
- SOMCallStyle
- SOMClassInit
- SOMClassName
- SOMClassVersion
- SOMDataName
- SOMDefine
- SOMMetaClass
- SOMMethodAppend
- SOMMethodName
- SOMNoDataDirect
- SOMNoMangling
- SOMNonDTS
- SOMReleaseOrder

## Restrictions on #pragma Directives

The following table lists the restrictions on using #pragma directives, and shows whether a directive is valid in C, C++, or both. A blank entry in the table indicates no restrictions.

Table 12. Restrictions on #pragmas

#pragma	Restriction on Number of Occurrences	Restriction on Placement	C	C++
chars	Once.	On the first #pragma directive, and before any code or directive, except for the pragmas filetag, longname, langlvl or target, which may precede this directive.	yes	yes

## #pragma

Table 12. Restrictions on #pragmas (continued)

#pragma	Restriction on Number of Occurrences	Restriction on Placement	C	C++
checkout			yes	yes
comment	The copyright directive can appear only once.	The copyright directive must appear before any C or C++ code.	yes	yes
csect	Three times. Once for code, once for static data, and once for debug information.		yes	yes
convlit			yes	yes
define		Wherever a declaration is allowed.		yes
disjoint		Wherever a declaration is allowed.	yes	
environment			yes	
export		Cannot export the main() function.	yes	yes
filetag	Once per file scope.	On the first #pragma directive, and before any code or directive, except for all conditional compilation directives (such as #if or #ifdef) which may precede this directive.	yes	yes
hdrstop			yes	yes
implementation		Wherever a declaration is allowed.		yes
info				yes
inline		At file scope.	yes	
isolated_call		Wherever a declaration is allowed.	yes	yes
noinline		At file scope.	yes	yes
langlvl	Once.	On the first #pragma directive, and before any code or directive, except for the pragmas filetag, longname, chars or target, which may precede this directive.	yes	yes
linkage	Can appear more than once for each function, as long as one #pragma does not contradict another #pragma.		yes	
longname	Once.	On the first #pragma directive, except for pragmas filetag, chars, langlvl or target, which may precede this directive.	yes	yes
map			yes	yes
margins			yes	yes
options		Before any C code.	yes	
pack			yes	yes
page			yes	
pagesize			yes	
priority				yes
runopts			yes	yes
sequence			yes	yes
skip			yes	
strings	Once.	Before any C or C++ code.	yes	yes

Table 12. Restrictions on #pragmas (continued)

#pragma	Restriction on Number of Occurrences	Restriction on Placement	C	C++
subtitle			yes	
target	Once.	On the first #pragma directive, and before any code or directive, except for pragmas filetag, chars, langlvl, or longname, which may precede this directive.	yes	
title			yes	
variable			yes	yes
wsizeof			yes	yes

## IPA Considerations

Interprocedural Analysis (IPA), through the IPA compiler option, is a mechanism for performing optimizations across the compilation units of your OS/390 C or C++ program. IPA also performs optimizations not otherwise available with the C/C++ compiler. Refer to the *OS/390 C/C++ Programming Guide* for an overview of IPA.

Many #pragma directives do not have any special behavior under IPA. They have the same effect on a program compiled with or without the IPA option.

You may see changes during the IPA Link step, due to the effect of a #pragma directive. The IPA Link step detects and resolves the conflicting effects of #pragma directives, and the conflicting effects of #pragma directives and compiler options that you specified for different compilation units. There may also be conflicting effects between #pragma directives and equivalent compiler options that you specified for the IPA Link step.

IPA resolves these conflicts similar to the way it resolves conflicting effects of compiler options that are specified for the IPA Compile step and the IPA Link step. The Compiler Options Map section of the IPA Link step listing shows the conflicting effects between compiler options and #pragma directives, along with the resolutions.

For those #pragma directives where there are special considerations for IPA, the following #pragma descriptions include IPA-related information.

## chars

The #pragma chars directive specifies that the compiler is to treat all char objects as signed or unsigned.

```

▶▶ #pragma chars ( [ unsigned | signed ] ) ▶▶

```

This pragma must appear on the first #pragma directive. It must also appear before any code or directive, except for the pragmas filetag, longname, langlvl or target. These pragmas may precede this directive. Once specified, it applies to the

## #pragma

rest of the file and you cannot turn it off. If a source file contains any functions that you want to compile without #pragma chars, place these functions in a different file.

The default character type behaves like an unsigned char.

## checkout

The #pragma checkout directive is a OS/390 C/C++ directive and an addition to the SAA Standard.

This pragma can appear anywhere that a preprocessor directive is valid.

►► #pragma checkout ( resume suspend ) ►►

With #pragma checkout, you can suspend the diagnostics that the CHECKOUT C compiler option or the INFO C++ compiler option performs during specific portions of your program. You can then resume the same level of diagnostics later in the file.

Nested #pragma checkout directives are allowed and behave as the following example demonstrates:

```
/* Assume CHECKOUT(PPTRACE) had been specified */
#pragma checkout(suspend) /* No CHECKOUT diagnostics are performed */
...
#pragma checkout(suspend) /* No effect */
...
#pragma checkout(resume) /* No effect */
...
#pragma checkout(resume) /* CHECKOUT(PPTRACE) diagnostics continue */
```

## comment

The #pragma comment directive places a comment into the object module. This pragma must appear before any C or C++ code or directive in a source file. The "token\_sequence" field in this pragma has a 1024-byte limit.

►► #pragma comment ►►

► ( compiler date timestamp copyright user ,—"token\_sequence"—" ) ►►

The comment type can be:

compiler	The compiler appends its name and version in an END information record at the end of the generated object module. OS/390 C/C++ does not include the name and version when it generates an executable, nor does it load the name and version into
----------	--

## #pragma

memory when it runs the program. This information can be printed out using the C370LIB utility with the MAP option.

**date** The compiler appends the date and time of compilation in an END information record at the end of the generated object module. OS/390 C/C++ does not include the date and time when it generates an executable nor does it load the date and time into memory when it runs the program. This information can be printed out using the C370LIB utility with the MAP option.

**timestamp** The compiler appends the date and time of the last modification of the source in an END information record at the end of the generated object module. OS/390 C/C++ does not include the date and time when it generates an executable nor does it load the date and time into memory when it runs the program. This information can be printed out using the C370LIB utility with the MAP option.

If OS/390 C/C++ cannot find the timestamp for a source file, the #pragma comment directive returns Mon Jan 1 0:00:01 1990.

**copyright** The compiler places text that is specified by the *token\_sequence*, if any, into the generated object module. When OS/390 C/C++ creates an executable, it includes the *token\_sequence* in the load module. The module is loaded into memory when OS/390 C/C++ runs the program.

**user** The compiler places the text that is specified by the *token\_sequence*, if any, into the generated object module. When OS/390 C/C++ creates an executable, the *token\_sequence* is included in the load module. Note that OS/390 C/C++ does *not* necessarily load it into memory when it runs the program. OS/390 C/C++ places the *token\_sequence* on END records in columns 34 to 71.

The characters in the *token\_sequence* field, if specified, must be enclosed in double quotation marks ("").

You can display the object-file comments by using the MAP option for the C370LIB utility.

## IPA Considerations

The #pragma comment directive affects the IPA Compile step only if the OBJECT suboption of the IPA compile option is in effect.

During the partitioning process in the IPA Link step, the compiler places the text string information #pragma comment at the beginning of partition 0. Partition 0 is the initialization partition.

## convlit

The #pragma convlit directive allows you to suspend the string literal conversion that the convlit compiler option performs during specific portions of your program. You can then resume the conversion at some later point in the file.

►► #pragma convlit ( — resume — ) ————— ►►  
                                  └ suspend ─┘

## #pragma

The pragma is effective only when you specify the CONVLIT compile option.

If you select the PPONLY option, OS/390 C/C++ echoes the convlit pragma to the expanded source file.

You can nest #pragma convlit directives. They behave as the following example demonstrates:

```
/* Assume CONVLIT (<codepage>) had been specified */
#pragma convlit(suspend) /* No string literal conversion */
...
#pragma convlit(suspend) /* No effect */
...
#pragma convlit(resume) /* No effect */
...
#pragma convlit(resume) /* String literal conversion continues */
```

Macros, user-defined and pre-defined, are replaced before tokenization; therefore, using #pragma convlit(suspend) and #pragma convlit(resume) around a macro definition would have no effect.

For example:

```
/* No effect on macro definition when using #pragma convlit(suspend)
   and #pragma convlit(resume)*/

main() {
    #pragma convlit (suspend)

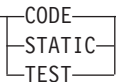
    #define str "Hello World!"
    puts(str);          /* macro str is not converted */

    #pragma convlit(resume)

    puts(str);          /* macro str is converted */
}
```

## csect

The #pragma csect directive identifies the name for either the code, static, or debug control section (CSECT).

►► #pragma csect ( , —" name "—) ►►

It is a OS/390 C/C++ specific pragma, and an addition to the SAA Standard.

**code** Specifies the CSECT that contains the executable code (C functions) and constant data.

**static** Designates the CSECT that contains all program variables with the static storage class and all character strings.

**test** Designates the CSECT that contains debug information. You must specify the TEST option.

The above syntax encloses the *name* in double quotation marks. This is the name that is used for the applicable CSECT (code, static, or test). OS/390 C/C++ does not map the name in any way, including uppercasing. If the name is greater than 8



characters, you must turn on the LONGNAME option. The name must not conflict with the name of an exposed name (external function or object) in a source file. In addition, it must not conflict with another #pragma csect directive or #pragma map directive. For example, the name of the code CSECT must differ from the name of the static and test CSECTs.

At most, three #pragma csect directives can appear in a source program as follows:

- One for the code CSECT
- One for the static CSECT
- One for the debug CSECT

Consider when there is no #pragma csect directive in the source file and you specify the CSECT compile option. In this case, OS/390 C/C++ automatically generates CSECT names from the source file name. For examples that show the file names that are generated when using either the #pragma csect or the CSECT compile option, see the section that describes the CSECT option in the *OS/390 C/C++ User's Guide*.

Private code has a disadvantage. When new code is linked to an executable containing old code, the new code replaces the old. The old code, however, is not discarded from the executable. The size of the executable will grow, and you may get duplicates of functions. Naming the CSECTs with this directive replaces the old code with the new, and removes the old code from the executable. If you want replacement and removal, name the code, static, and test CSECT.

## IPA Considerations

Use the #pragma csect directive when naming regular objects only if the OBJECT suboption of the IPA compile option is in effect. Otherwise, the compiler discards the CSECT names that #pragma csect generated.

Refer to the IPA Link Step chapter in the *OS/390 C/C++ User's Guide* for information on how the IPA Link step sets CSECT names.

## define (C++ Only)

The #pragma define directive forces the definition of a template class without actually defining an object of the class.

►► #pragma define (—*template\_class\_name*—) —————►

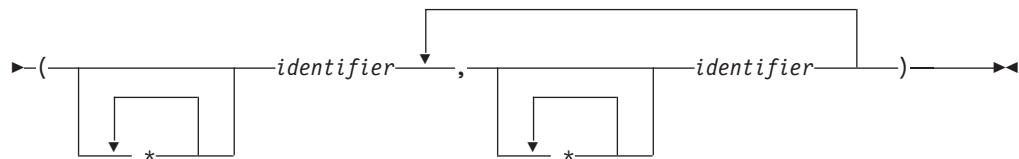
The pragma can appear anywhere that a declaration is allowed. Use the pragma to organize your program to efficiently or automatically generate template functions.

## disjoint (C Only)

The #pragma disjoint directive lists the identifiers that are not aliased to each other within the scope of their use. In the following syntax diagram, *identifier* is the name of a variable:

►► #pragma disjoint —————►

## #pragma



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 class, structure, or union
- A structure, union, or enumeration tag
- An enumeration constant
- A typedef name
- A label

You must declare the identifiers before using them in the pragma. Your program must not dereference a pointer in the identifier list nor use it as a function argument before it appears in the directive.

The following example shows the use of `#pragma disjoint`.

```
int a, b, *ptr_a, *ptr_b;

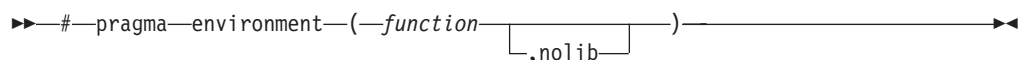
#pragma disjoint(*ptr_a, b) /* *ptr_a never points to b */
#pragma disjoint(*ptr_b, a) /* *ptr_b never points to a */
one_function()
{
    b = 6;
    *ptr_a = 7; /* Assignment will not change the value of b */

    another_function(b); /* Argument "b" has the value 6 */
}
```

External pointer `ptr_a` does not share storage with and never points to the external variable `b`. Consequently, assigning 7 to the object to which `ptr_a` points will not change the value of `b`. Likewise, external pointer `ptr_b` does not share storage with and never points to the external variable `a`. The compiler can assume that the argument to `another_function` has the value 6 and will not reload the variable from memory.

## environment (C Only)

The `#pragma environment` directive is an OS/390 C directive, and an addition to the SAA Standard.



With the #pragma environment directive, you can use OS/390 C code as an assembler substitute. See the *OS/390 C/C++ Programming Guide* for more information on this use. The directive allows you to do the following:

- Specify entry points other than main
- Omit setting up an OS/390 C environment on entry to this function
- Specify several system exits that are written in OS/390 C code in the same executable

If you specify nolib, the environment is established, and the library is not loaded at run time. If you do not specify anything, the library is loaded.

**Note:** If you specify any other value than nolib after the function name, behavior is not defined.

## export

The #pragma export directive declares that a function or variable is to be exported. It also specifies the name of the function or variable to be referenced outside the module. You can use this #pragma to export functions or variables from a DLL module.

►► #pragma export ( function  
variable ) ►►

#pragma export is an OS/390 C/C++ specific directive and an addition to the SAA standard.

With the #pragma export directive, you can export specific functions and variables to the users of your DLL. See the *OS/390 C/C++ Programming Guide* for more information on creating and using DLLs.

You can specify this pragma anywhere in the DLL source code, on its own line, or with other pragmas. You can also specify it before or after the definition of the variable or function. You must externally define the exported function or variable.

**Note:** You cannot export the main() function. You can also use the \_Export keyword to export a function.

## IPA Considerations

If you specify this #pragma in your source code in the IPA Compile step, you cannot override the effects of this #pragma on the IPA Link step.

## filetag

The #pragma filetag directive specifies the code set in which the source code was entered.

►► #pragma filetag (—"code set name"—) ►►

Since the # character is variant between code sets, use the trigraph representation ??= instead of # as illustrated below.

## #pragma

The `#pragma filetag` directive must appear at most once per source file. It must appear before the first statement or directive, except for all conditional compilation directives, which may precede this directive. For example:

```
??=ifdef COMPILER_VER          /* This is allowed. */
    ??=pragma filetag ("code set")

??=endif
```

It should not appear in combination with any other `#pragma` directives. For example, the directive is incorrect:

```
??=pragma filetag ("IBM-1047") export (baffle_1)
```

If there are comments before the `pragma`, OS/390 C/C++ does not translate them to the code page that is associated with the `LOCALE` option.

See the *OS/390 C/C++ Programming Guide* for details on using this directive with the `LOCALE` option.

## hdrstop

The `#pragma hdrstop` directive manually terminates the initial sequence of `#include` directives that are being considered for precompilation.

►►—#—pragma—hdrstop—►►

It has no effect under the following conditions:

- The initial sequence of `#include` directives has already ended
- You do not specify either the `GENPCH` option or the `USEPCH` option
- It does not appear in the primary source file

The *OS/390 C/C++ User's Guide* describes how to structure your files so the compiler can take full advantage of the precompiled headers.

## Examples

The following example only precompiles the header `h1.h` by using the file `default.pch` (provided you specify `USEPCH` or `GENPCH`). If you specify `USEPCH(dave.pch)` `GENPPCH(john.pch)`, the compiler will look for the precompiled headers in `john.pch` and will regenerate them if they are not found or not usable.

```
#include "h1.h"
#pragma hdrstop
#include "h2.h"
main () {}
```

The following example does not use nor does it generate precompiled headers for the compilation, even if you specify `GENPCH` or `USEPCH`.

```
#pragma hdrstop
#include "h1.h"
#include "h2.h"
main () {}
```

## implementation (C++ Only)

The `#pragma implementation` directive tells the compiler the name of the file containing the function-template definitions. These definitions correspond to the template declarations in the include file which contains the pragma.

```
▶▶ #pragma implementation (—string_literal—) ▶▶
```

This pragma can appear anywhere that a declaration is allowed. Use this pragma to organize your program to efficiently or automatically generate template functions.

**Note:** `#pragma implementation` is only effective if the `TEMPINC` option is in effect. If the `NOTEMPINC` option is in effect, you must test the value of the `__TEMPINC__` macro, and conditionally include the required source.

## info (C++ Only)

The `#pragma info` directive controls the diagnostic messages that are generated by the `INFO` compile option.

```
▶▶ #pragma info (—suspend—  
                —resume—) ▶▶
```

You can use this pragma directive in place of the `INFO` option.

Use `#pragma info suspend` to suspend the diagnostics that the `INFO` compiler option performs during specific portions of your program. You can then use `#pragma info resume` to resume the same level of diagnostics later in the file.

You can also use `#pragma checkout` to suspend or resume diagnostics.

The *OS/390 C/C++ User's Guide* describes the `INFO` option.

## inline (C Only) - also see noline

The `#pragma inline` directive specifies whether or not the *function* is to be inlined. The pragma can be anywhere in the source, but must be at file scope. `#pragma inline` has no effect if you have not specified the `INLINE` or the `OPT` compiler option.

```
▶▶ #pragma —inline—  
                —noinline— (—function—) ▶▶
```

The `#pragma inline` directive is an OS/390 C directive and is an addition to the SAA Standard.

The `#pragma noline` directive is an OS/390 C/C++ directive and is an addition to the SAA Standard.

## #pragma

If you specify `#pragma inline`, the function is inlined on every call. The function is inlined in both selective (NOAUTO) and automatic (AUTO) mode. For OS/390 C++, you can inline functions using the `inline` keyword.

If you specify `#pragma noline` in your C or C++ program, the function is never inlined when you call it. This pragma has no effect when you specify NOAUTO with the OS/390 C `INLINE` compile option.

The default when compiling with the `OPTIMIZE` option is to inline functions even if the OS/390 C++ `inline` keyword has not been specified. The default when compiling with the `NOOPTIMIZE` option is to only inline C++ functions that are:

- Implicitly inlined; that is when the code for a member function is included inside a class definition
- Explicitly inlined; that is when the `inline` keyword is used when declaring a function

For OS/390 C++, you can place the `#pragma noline` directive anywhere in the source. For OS/390 C it must be at file scope.

The `#pragma noline` directive is the only way to turn off inlining of functions that have been implicitly or explicitly inlined. It also takes precedence over the OS/390 C++ `inline` keyword.

## IPA Considerations

The compiler uses the IPA Link control file directive in the following cases:

- If you specify both the `#pragma noline` directive and the IPA Link control file `inline` directive for a function
- If you specify both the `#pragma inline` directive and the IPA Link control file `noline` directive for a function

## Example

### CBC3RABE:

```
/* this example shows how #pragma inline may be used */

#pragma csect(code,"MYCFIL")
#pragma csect(static,"MYSFIL")
#pragma options(INLINE)

#include <stdio.h>
#include <stdlib.h>

static int (writerecord) (int, char *);

#pragma inline (writerecord)

int main()
{
    int chardigit;
    int digit;

    printf("Enter a digit\n");
    chardigit = getchar();

    digit = chardigit - '0';
    if (digit < 0 || digit > 9)
    {
        printf("The digit you entered is not between 1 and 8\n");
        exit(99);
    }
}
```

```

    }
    switch(digit)
    {
        case 0:
            writerecord(0, "entered 0");
            break;
        case 1:
            writerecord(1, "entered 1");
            break;
        default:
            writerecord(9, "entered other");
    }
}

static int writerecord (int digit, char *phrase)
{
    switch (digit)
    {
        case 0:
            printf("writerecord 0: ");
            printf("%s\n", phrase);
            break;
        case 1:
            printf("writerecord 1: ");
            printf("%s\n", phrase);
            break;
        case 2:
            printf("writerecord 2: ");
            printf("%s\n", phrase);
            break;
        case 3:
            printf("writerecord 3: ");
            printf("%s\n", phrase);
            break;
        default:
            printf("writerecord X: ");
            printf("%s\n", phrase);
    }

    return 0;
}

```

## isolated\_call

The `#pragma isolated_call` directive lists functions that do not alter data objects visible at the time of the function call. In the following syntax diagram, *identifier* is a primary expression that can be an identifier, operator function, conversion function, or qualified name:



The pragma must appear before calls to the functions in the identifier list. You must declare the identifiers that are listed before using them in the pragma. They must be of type function, or a typedef of function. If a name refers to an overloaded function, all variants of that function declared before the pragma are marked as isolated calls.

## #pragma

The pragma informs the compiler that none of the functions listed has side effects. For example:

- Accessing a volatile object
- Modifying an external object
- Modifying a file

Otherwise, you can consider calling a function that does any of the above to be side effects.

Consider any change in the state of the run-time environment a side effect. Passing function arguments by reference is one side effect that OS/390 C/C++ allows. In general, however, 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. It also indicates that references to storage can be deleted from the calling function where appropriate. 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.

When a function is marked as isolated, the compiler can make more optimistic assumptions about what variables the function modifies. The compiler may move function calls to functions that are flagged as isolated to a different location in the code or even remove them entirely.

The following example routines shows you when to use the #pragma isolated\_call directive (routine addmult). It also shows you when not to use it (routines same and check):

```
#include <stdio.h>
#include <math.h>

int addmult(int op1, int op2);
#pragma isolated_call(addmult)

/* This routine is a good candidate to be flagged as isolated as its */
/* result is constant with constant input and it has no side effects. */
int addmult(int op1, int op2) {
    int rslt;

    rslt = op1*op2 + op2;
    return rslt;
}

/* The routine 'same' should not be flagged as isolated as its state */
/* (the static variable delta) can change when it is called.          */
int same(double op1, double op2) {
    static double delta = 1.0;
    double temp;

    temp = (op1-op2)/op1;
    if (fabs(temp) < delta)
        return 1;
    else {
        delta = delta / 2;
        return 0;
    }
}

/* The routine 'check' should not be flagged as isolated as it has a */
/* side effect of possibly emitting output.                             */
int check(int op1, int op2) {
```



```

if (op1 < op2)
    return -1;
if (op1 > op2)
    return 1;
printf("Operands are the same.\n");
return 0;
}

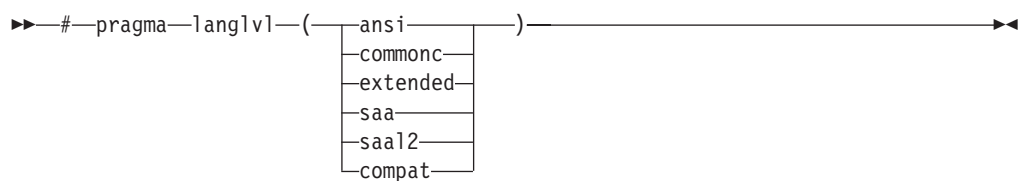
```

## IPA Considerations

If you specify this #pragma in your source code in the IPA Compile step, you cannot override the effects of this #pragma on the IPA Link step.

## langlvl

The #pragma langlvl directive selects the C or C++ language level for compilation.



You can only specify this pragma only once in a source file. It must appear before any statements in a source file. The compiler uses predefined macros in the header files to make declarations and definitions available that define the specified language level.

The default language level is EXTENDED.

ansi	Defines the predefined macros <code>__ANSI__</code> and <code>__STDC__</code> and undefines other <code>langlvl</code> variables. It allows only language constructs that conform to ANSI/ISO C standards.
extended	Defines the predefined macro <code>__EXTENDED__</code> and undefines other <code>langlvl</code> variables. The default language level is EXTENDED. OS/390 C/C++ defines the <code>__EXTENDED__</code> macro as 1. Note that #pragma <code>langlvl (EXTENDED)</code> has no effect in the OS/390 UNIX environment. In OS/390 UNIX, you must use the compile option <code>LANGVLV(EXTENDED)</code> instead of the pragma.
commonc	Defines the predefined macro <code>__COMMONC__</code> and <code>__EXTENDED__</code> and undefines other <code>langlvl</code> variables. This language level allows compilation of code that contains constructs defined by the X/Open Portability Guide (XPG) Issue 3 C language (referred to as Common Usage C). It is roughly equivalent to what is commonly known as K&R C. See "Appendix B. Common Usage C Language Level" on page 407 for more information about the OS/390 C/C++ implementation of Common Usage C.  OS/390 C/C++ does not support this macro for C++.
saa	Defines the predefined macro <code>__SAA__</code> and undefines other <code>langlvl</code> variables. OS/390 C/C++ does not support this macro for C++.

## #pragma

saal2	Defines the predefined macro <code>__SAA_L2__</code> and undefines other <code>langlvl</code> variables. OS/390 C/C++ does not support this macro for C++.
compat	Defines the predefined macro <code>__COMPAT__</code> and undefines other <code>langlvl</code> variables. This macro is not supported for C. It is provided for cfront compatibility.

The `#pragma langlvl(extended)` permits packed decimal types and it issues a warning message when it detects assignment between integral types and pointer types.

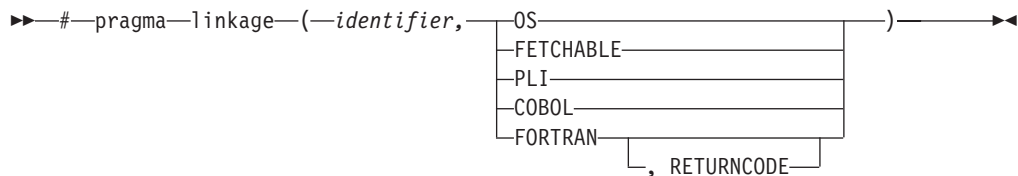
The `#pragma langlvl(ansi)` does not permit packed decimal types and issues an error message when it detects assignment between integral types and pointer types. Packed decimal applies to C only.

If you specify `#pragma langlvl(ansi)`, OS/390 C/C++ does not allow the `NOEXH` compile option, since `NOEXH` breaks ANSI conformance. The `EXH` and `NOEXH` compile options apply to C++ only.

The `LANGLVL` compile option has the same effect as this pragma. The *OS/390 C/C++ User's Guide* describes this option.

## linkage

The `#pragma linkage` directive identifies the entry point of modules that are used in interlanguage calls.



The *identifier* either identifies the name of the function that is to be the entry point of the module. Or, it identifies a typedef that will be used to define the entry point.

In C++, you accomplish this by using `extern "linkage-type"` when declaring an identifier, for example,

```
extern "FORTRAN" void f();
extern "COBOL" void g();
```

The `#pragma linkage` directive also designates other entry points within a program that you can use in a `fetch` operation.

The following are the linkage entry points:

FETCHABLE	Specifies a name, other than <code>main</code> , as an entry point within the program. This pragma also indicates that this name ( <i>identifier</i> in the syntax diagram) can be used in a <code>fetch()</code> operation. See the <i>OS/390 C/C++ Run-Time Library Reference</i> for more information on the use of the <code>fetch()</code> library function.
-----------	---

## #pragma

OS	Designates an entry point ( <i>identifier</i> in the syntax diagram) as an OS linkage entry point. OS linkage is the basic linkage convention that is used by the operating system.
PLI	Designates an entry point ( <i>identifier</i> in the syntax diagram) as a PL/I linkage entry point.
COBOL	Designates an entry point ( <i>identifier</i> in the syntax diagram) as a COBOL linkage entry point.
FORTTRAN	<p>Designates an entry point (<i>identifier</i> in the syntax diagram) as a FORTRAN linkage entry point.</p> <p>You can specify the RETURNCODE keyword with the FORTRAN keyword for C programs only. OS/390 C/C++ does not support it for C++. RETURNCODE indicates to the compiler that the routine named by <i>identifier</i> is a FORTRAN routine, which returns an alternate return code. It also indicates that the routine is defined outside the compilation unit. You can retrieve the return code by using the <code>fortrc()</code> function (refer to the <i>OS/390 C/C++ Run-Time Library Reference</i> for more information). If the compiler finds the function definition inside the compilation unit, it issues an error message. Note that you can define functions outside the compilation unit, even if you do not specify the RETURNCODE keyword.</p>

You can use a typedef in a #pragma linkage directive to associate a specific linkage convention with the typedef of a function.

```
typedef void func_t(void);
#pragma linkage (func_t,OS)
```

In the example, the #pragma linkage directive associates the OS linkage convention with the typedef `func_t`. This typedef can be used in C declarations wherever a function type specifies the type function of OS linkage type.

Refer to *OS/390 Language Environment Writing Interlanguage Applications* for more information about interlanguage calls.

## longname

The #pragma longname directive specifies that the compiler is to generate not-truncated and mixed case names in the object module that is produced by the compiler. These names can be up to 1024 characters in length.

```
►►—#—pragma—longname—►►
      |
      |no longname
```

If you use the #pragma longname directive for an OS/390 C or C++ program, you must either use the binder to produce a program object in a PDSE, or you must use the prelinker. The binder, IPA Link step, and prelinker support the long name directory that is generated by the Object Library utility for autocall.

If you specify the NOLONGNAME compile option, the compiler ignores the #pragma longname directive. If you specify the LONGNAME compile option, the compiler ignores the #pragma no longname.

## #pragma

**Note:** The OS/390 C compiler defaults to the NOLONGNAME compile option, and the OS/390 C++ compiler defaults to the LONGNAME compile option.

Under OS/390 C, if you specify the ALIAS compile option, the compiler creates a NAME control statement, but no ALIAS control statements. You can use the OS/390 C Object Library Utility to create a library of object modules with a long name directory which supports autocall of long name symbols.

If you have more than one preprocessor directive, #pragma longname may be preceded only by #pragma filetag, #pragma chars, #pragma langlvl, and #pragma target. Some directives, such as #pragma variable and #pragma linkage are sensitive to the name handling.

For OS/390 C++, you must specify #pragma longname and #pragma nolongname before any code. Otherwise, the compiler issues a warning message.

If you use #pragma map to associate an external name with an identifier, the external name is produced in the object module. That is, #pragma map has the same behavior with or without the #pragma longname directive.

The #pragma nolongname directive directs the compiler to generate truncated and uppercase names in the object module produced by the compiler. When the #pragma nolongname directive is specified, only functions that do not have C++ linkage are given truncated and uppercase names. More details on external name mapping are provided in the section, "map". Also, if you have more than one preprocessor directive, #pragma nolongname must be the first one.

If you specify either #pragma nolongname or the NOLONGNAME option, and this results in mapping of two different source code names to the same object code name, the compiler will not issue an error message.

## IPA Considerations

You must specify either the LONGNAME compile option or the #pragma longname preprocessor directive for the IPA Compile step (unless you are using the c89 utility). Otherwise, you receive an unrecoverable compiler error.

## map

The #pragma map directive tells the compiler to convert all references to an identifier to "*name*".

#pragma map is a OS/390 C/C++ directive and an addition to SAA standard. If you use the #pragma map directive, the C/C++ name in the source file is not visible in the object deck. The map name represents the object in the object deck.

### #pragma map for OS/390 C

For C, #pragma map has the form:

►►—#—pragma—map—(—*identifier*—,—"*name*"—)——————►◄

*identifier*                      A name of a data object or function with external linkage.

*name*                   The external name that the compiler binds to the given object or function.

The directive can appear anywhere within a single compilation unit. It can appear before any declaration or definition of the named object or function.

You should enclose *name* in double quotation marks. The maximum length for external names is 8 characters. This is because external names in object modules can be 8 characters at most without the LONGNAME compile option. The compiler keeps it as specified on the #pragma map directive in mixed case. It must not conflict with the name in another #pragma map or #pragma csect directive.

The map name is an external name, thus you must not use it in the source file to reference the object. If you use the map name in the source file to access the corresponding object, the compiler treats it as a new identifier.

The compiler produces an error message if you give more than one map name to an identifier. Two different identifiers can have the same map name.

The compiler resolves the identifiers appearing in the directive, including any type names used in the prototype argument list. The compiler resolves them as though the directive had appeared at file scope, independent of its actual point of occurrence.

For example:

```
extern "C" int func(int);
#pragma map(func, "funcname1")    // maps ::func
```

## #pragma map for OS/390 C++

For OS/390 C++, #pragma map has the form:

```
▶▶ #pragma map _____ ▶▶

▶▶ ( [ identifier _____ ] , "name" ) _____ ▶▶
    |
    | func_or_op_identifier ( argument_list )
    |_____
```

*identifier*                   A name of a data object or a nonoverloaded function with external linkage.

*func\_or\_op\_identifier*       A name of a function or operator with external linkage. The name can be qualified.

*argument\_list*              A prototype list for the named function or operator.

*name*                        The external name that is bound to the given object, function, or operator.

The directive can appear anywhere within a single compilation unit. It can appear before any declaration or definition of the named object, function, or operator. The compiler resolves the identifiers appearing in the directive, including any type names used in the prototype argument list. It resolves them as though the directive had appeared at file scope, independent of its actual point of occurrence.

## #pragma

For example:

```
int func(int);

class X
{
public:
    void func(void);
#pragma map(func, "funcname1")    // maps ::func
#pragma map(X::func, "funcname2") // maps X::func
};
```

In C++, you should not use #pragma map to map the following:

- C++ Member functions
- Overloaded functions
- Objects generated from templates
- Functions with C++ linkage, or builtin linkage

Such mappings override the compiler-generated names, which could cause IPA Link or binder errors.

### IPA Considerations

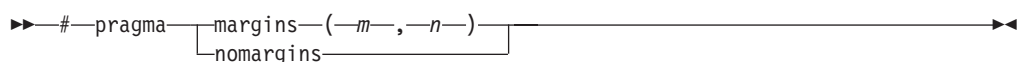
The use of the #pragma map directive for variables will inhibit the global coalescing optimization of these variables during the IPA Link step.

## margins

The #pragma margins directive specifies the margins in the source file that are to be scanned for input to the compiler. You cannot specify columns (*m,n*) for OS/390 C++. The #pragma nomargins directive specifies that the entire input source record is to be scanned for input to the compiler.

#pragma margins is a OS/390 C/C++ directive and an addition to the SAA Standard.

### #pragma margins for OS/390 C



### #pragma margins for OS/390 C++



In the syntax diagram, you can specify the following parameters for OS/390 C:

*m* The first column of the source input that contains a valid C program. The value of *m* must be greater than 0, and less than 32761.

Also, *m* must be less than or equal to the value of *n*.

*n* The last column of the source input that contains a valid C program. The value of *n* must be greater than 0, and less than 32761.

You can assign an asterisk (\*) to *n*. The asterisk indicates the last column of the input record. For example, if you specify `#pragma margins(8,*)`, the compiler scans from column 8 to the end of the record for input source statements.

You can use `#pragma margins` and `#pragma sequence` together. If they reserve the same columns, `#pragma sequence` has priority and it reserves the columns for sequence numbers. For example, assume columns 1 to 20 are reserved for the margin, and columns 15 to 25 are reserved for sequence numbers. In this case, the margin will be from column 1 to 14, and the columns reserved for sequence numbers will be from 15 to 25.

For more information on the `#pragma sequence` directive, refer to “sequence” on page 272.

The margin setting specified by the `#pragma margins` directive applies only to the source file or include file in which it is found. It has no effect on other `#include` files. The `#pragma margins` and the `#pragma nomargins` directives come into effect on the line following the directive. They remain in effect until the compiler encounters another `#pragma margins` or `#pragma nomargins` directive, or until the compiler reaches the end of the file.

If you use the compile options `MARGINS` or `NOMARGINS` with the `#pragma margins` or `#pragma nomargins` directives, the `#pragma` directives override the compile options. The compile option specified will be in effect up to, and including, the `#pragma margins` or `#pragma nomargins` directive.

For OS/390 C++, the `#pragma margins` specifies that columns 1 through 72 in the input record are to be scanned for input to the compiler. The input file can have fixed or variable record length. The compiler ignores any text in the source input that does not fall within the range.

For OS/390 C, the default setting is `MARGINS(1,72)` for fixed-length records, and `NOMARGINS` for variable-length records. For OS/390 C++, the default is `NOMARGINS`.

## **noinline (C and C++) - also see inline**

The `#pragma noinline` directive is an OS/390 C/C++ directive and is an addition to the SAA Standard.

The `#pragma noinline` specifies that the function is never inlined when you call it. This pragma has no effect when you specify `NOAUTO` with the OS/390 C `INLINE` compile option.

You can place the `#pragma noinline` directive anywhere in a C++ program. The directive must be at file scope in a C program.

The `#pragma noinline` directive is the only way to turn off inlining of functions that have been implicitly or explicitly inlined at compile time. It also takes precedence over the OS/390 C/C++ `inline` keyword.

See “inline (C Only) - also see noinline” on page 255 for more information. For more information on how to use `#pragma noinline`, refer to the *OS/390 C/C++ User's Guide*.

## #pragma

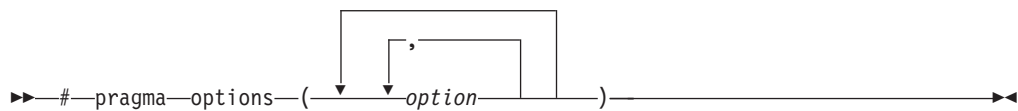
### IPA Considerations

If you use either the #pragma inline or the #pragma ninline directive in your source, you can later override them with an appropriate IPA Link control file directive during the IPA Link step. For example:

- If you specify both the #pragma ninline directive and the IPA Link control file inline directive for a function.
- If you specify both the #pragma inline directive and the IPA Link control file ninline directive for a function.

### options (C Only)

The #pragma options directive specifies a list of compile options that are to be processed as if you had typed them on the command line or on the CPARM parameter of the IBM-supplied cataloged procedures.



The only compile options that are allowed on a #pragma options directive are:

AGGREGATE NOAGGREGATE	ALIAS NOALIAS	ANS NOANS
ARCH	CHECKOUT NOCHECKOUT	DECK NODECK
GONUMBER NOGONUMBER	HWOPTS NOHWOPTS <sup>1</sup>	INLINE NOLIBANSI
LIBANSI NOLIBANSI	MAXMEM NOMAXMEM	OBJECT NOOBJECT
OPTIMIZE NOOPTIMIZE	RENT NORENT	SERVICE NOSERVICE
SPELL NOSPELL	START NOSTART	TEST NOTEST
UPCONV NOUPCONV	TUNE NOTUNE	XREF NOXREF

**Note:** <sup>1</sup> The compiler accepts the HWOPTS|NOHWOPTS option, but you should use the ARCHITECTURE option instead.

For a detailed description of these options refer to the *OS/390 C/C++ User's Guide*.

If you use a compile option that contradicts the options that are specified on the #pragma options directive, the compile option overrides the options on the #pragma options directive.

If you specify an option more than once, the compiler uses the last one you specified.

If you use one of the following compile options, the compiler inserts the option name at the bottom of your object module to indicate that it used the option:

ALIAS	ANSIALIAS	ARCHITECTURE
GONUMBER	HWOPTS	INLINE
LIBANSI	MAXMEM	OPTIMIZE (all levels)
RENT	SPELL	START
TARGET (all targets)	TEST	TUNE
UPCONV		



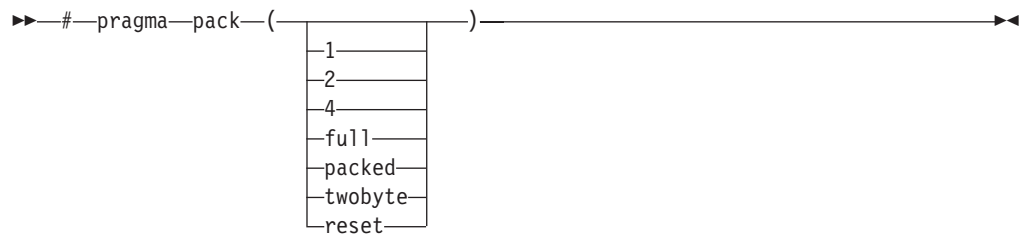
## IPA Considerations

You cannot specify the IPA compile-time option for #pragma options.

Refer to the *OS/390 C/C++ User's Guide* for descriptions of how different compile options affect IPA processing.

## pack

The #pragma pack directive specifies the alignment rules to use for the structures, unions, and classes that follow it. The C compiler performs packing on *definitions* if you specify `_Packed` and on *declarations* if you specify `#pragma pack`. The C++ compiler does not support `_Packed`, so it can only perform packing on *declarations*. This means that the packing applies to type-specifiers and not declarators.



where:

- `full` Is 4-byte boundary alignment. It is the system default boundary alignment. This is the same as `#pragma pack()` and `#pragma pack(4)`.
- `packed` Is 1-byte boundary alignment. This is the same as `#pragma pack(1)`.
- `twobyte` Is 2-byte boundary alignment. This is the same as `#pragma pack(2)`.
- `reset` Returns the alignment to the previous alignment rule.

The #pragma pack directive packs all structures and unions that follow it in the program along a boundary specified in the directive. It continues to pack until another #pragma pack directive changes the packing boundary. The #pragma pack directive does not apply to forward declarations of structures or unions. For example, in the following code fragment, the alignment for struct S is full. This is the rule when the declaration list is declared:

```
#pragma pack(packed)
struct S;
#pragma pack(full)
struct S { int i, j, k; };
```

The compiler packs declarations or types. This is different from the `_Packed` keyword in OS/390 C, where packing is also performed on definitions. For portability, you should use #pragma pack instead of the `_Packed` keyword.

The #pragma pack directive does not have the same effect as declaring a structure as `_Packed`. The `_Packed` keyword removes all padding between structure members, while the #pragma pack directive only specifies the boundaries to align the members.

Normal structure alignment aligns the structure members on their natural boundaries and ends the structure on its natural boundary. The alignment of the

## #pragma

structure is that of its strictest member. The compiler performs normal alignment when your program meets one of the following conditions:

- It does not specify the #pragma pack directive
- It specifies #pragma pack() before the structure declaration
- It specifies #pragma pack(full) before the structure declaration

To change the alignment back to what it was before the last #pragma pack, use the reset option.

Consider if, by default, the compiler packs data types along boundaries smaller than those specified by #pragma pack. The compiler still aligns them along the smaller boundaries. For example, the compiler always aligns type char along a 1-byte boundary, regardless of the value of #pragma pack.

Consider when more than one #pragma pack directive appears in a structure defined in an inlined function. In that case, the #pragma pack directive that is in effect at the beginning of the structure takes precedence.

If you are porting code from other platforms that contain #pragma pack directives or packed data, consider using the PORT compiler option to increase the syntax checking for the #pragma pack directive in your code. This option will allow you to adjust the error recovery action the compiler takes if the # pragma pack is incompatible with the OS/390 C/C++ # pragma pack. For more information on using the PORT option, see the *OS/390 C/C++ User's Guide*.

### Alignment of Nested Structures

A nested structure has the alignment that precedes its declaration, not the alignment of the structure in which it is contained.

```
#pragma pack ()           // full alignment
struct nested {
    int x;
    char y;
    int z;
};

#pragma pack(1)           // 1-byte alignment
struct packedcxx{
    char a;
    short b;
    struct nested s1;      // full alignment
};
```

### Alignment of Unions

You can also perform packing in a union. Each member starts at offset zero, and the entire union spans as many bytes as its largest element. The #pragma pack affects the total alignment restriction of the whole union. Consider the following example:

#### Without Packing:

```
union uu {
    short a;
    struct {
        char x;
        char y;
        char z;
    };
};
```

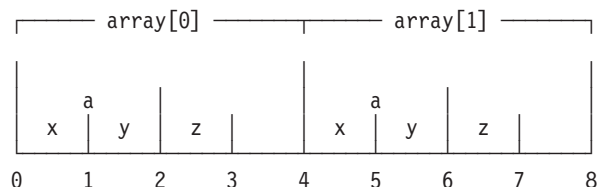
```

    } b;
};

union uu          array[2];

```

First, consider the non-packed array. Each of its elements is of type union uu. Since it is non-packed, every element has an alignment restriction of 2 bytes. The largest alignment requirement among the union members is that of short a. There is one byte of padding at the end of each element to enforce this requirement.



**With #pragma pack(packed):**

```

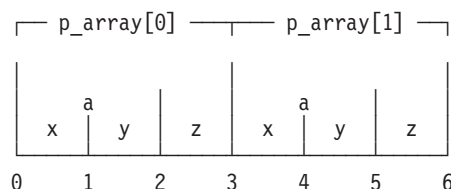
#pragma pack(packed)

union uu {
    short  a;
    struct {
        char x;
        char y;
        char z;
    } b;
};

union uu          p1_array[2];

```

Now consider the packed array `p1_array`. Since the example specifies `#pragma pack(packed)`, the alignment restriction of every element is the byte boundary. Therefore, each element has a length of only 3 bytes, as opposed to the 4 bytes of the previous case.



For information on calling C packed structures or unions from C++, see the *OS/390 C/C++ Programming Guide*.

For information on packing C structures, see “\_Packed Qualifier (C Only)” on page 122.

## Examples

**In a header file, file.h:**

```

#pragma pack(packed)

struct jeff{
    float bill;
    int *chris;
}
#pragma pack(reset)

```

/\* this structure is packed \*/  
/\* along 1-byte boundaries \*/  
/\* reset to previous alignment rule\*/

## #pragma

⋮

In a source file, file.cxx:

```
#pragma pack(full)

#include "file.h"           // inside the header file,
                           // the alignment rule is set to 1-byte
                           // and then reset to the system default

struct dor{                // this structure is packed
    double stephen;         // using the system default alignment
    long alex;
}
```

### page (C Only)

The #pragma page directive skips the number of pages that are specified by *pages* of the generated source listing. If you do not specify *pages*, it starts the next page.

►► #pragma page ( pages ) ◄◄

### pagesize (C Only)

The #pragma pagesize directive sets the number of lines per page to *n* for the generated source listing.

►► #pragma pagesize ( n ) ◄◄

The default page size is 60 lines. The minimum page size that you should set is 25.

### IPA Considerations

This #pragma has the same effect on the IPA Compile step as it does on a regular compilation. It has no effect on the IPA Link step.

### priority (C++ Only)

The #pragma priority directive specifies the order in which OS/390 C/C++ initializes static objects at run time.

►► #pragma priority ( -n ) ◄◄

*n* is an integer literal in the range of INT\_MIN to INT\_MAX. The default value is 0. A negative value indicates a higher priority; a positive value indicates a lower priority.

OS/390 C/C++ reserves the first 1024 priorities (INT\_MIN to INT\_MIN + 1023) for use by the compiler and its libraries. The priority value that is specified applies to all run-time static initialization in the current compilation unit.

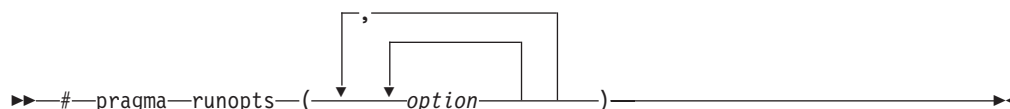
```
#pragma
```

OS/390 C/C++ constructs any global object declared before another object in a file first. Use `#pragma priority` to specify the construction order of objects across files.

To ensure that the objects are always constructed from top to bottom in a file, the compiler enforces a restriction. This restriction ensures that the priority specified for all objects before and all objects after it is at that priority until the next `#pragma`.

## runopts

The `#pragma runopts` directive specifies a list of runtime options that OS/390 C/C++ uses at execution time.



Specify your `#pragma runopts` directive in the compilation unit that contains `main`. If more than one compilation unit contains a `#pragma runopts` directive, unpredictable results can occur. The `#pragma runopts` directive only affects primary modules, and has no effect on a DLL.

If a sub-option to `#pragma runopts` is not a valid C token, you can surround the sub-options to `#pragma runopts` in double quotes. For example, use:

```
#pragma runopts ( " RPTSTG(ON) TEST(,,VADTCPIP&1.2.3.4:*) " )
```

instead of:

```
#pragma runopts ( RPTSTG(ON) TEST(,,VADTCPIP&1.2.4.3:*) )
```

Refer to “target (C Only)” on page 274 and the *OS/390 C/C++ User's Guide* for information about how `#pragma target` and the `TARGET` compile-time option affect `#pragma runopts`. Refer to the *OS/390 Language Environment Programming Guide* for descriptions of specific run-time options.

## IPA Considerations

This #pragma only affects the IPA Compile step if you specify the OBJECT suboption of the IPA compiler option.

The IPA Compile step passes the effects of this directive to the IPA Link step.

Consider if you specify `ARGPARSE|NOARGPARSE`, `EXECOPS|NOEXECOPS`, `PLIST`, or `REDIR|NOREDİR` either on the `#pragma runopts` directive or as a compile-time option on the IPA Compile step, and then specify the compile-time option on the IPA Link step. In this case, you override the value that you specified on the IPA Compile step.

If you specify the TARGET compile-time option on the IPA Link step, it has the following effects on #pragma runopts:

- It overrides the value you specified for `#pragma runopts(ENV)`. If you specify `TARGET(LE)` or `TARGET()`, the compiler sets the value of `#pragma runopts(ENV)` to `MVS`. If you specify `TARGET(IMS)`, the compiler sets the value of `#pragma runopts(ENV)` to `IMS`.

## #pragma

- It may override the value you specified for #pragma runopts(PLIST). If you specify TARGET(LE) or TARGET(), and you specified something other than HOST for #pragma runopts(PLIST), the compiler sets the value of #pragma runopts(PLIST) to HOST. If you specify TARGET(IMS), the compiler sets the value of #pragma runopts(PLIST) to IMS.

For #pragma runopts options other than those that are listed above, the IPA Link step follows these steps to determine which #pragma runopts value to use:

1. The IPA Link step uses the #pragma runopts specification from the main() routine, if the routine exists.
2. If no main() routine exists, the IPA Link step follows these steps:
  - a. If you define the CEEUOPT variable, the IPA Link step uses the #pragma runopts value from the first compilation unit that it finds that contains CEEUOPT.
  - b. If you have not defined the CEEUOPT variable in any compilation unit, the IPA Link step uses the #pragma runopts value from the first compilation unit that it processes.

The sequence of compilation unit processing is arbitrary.

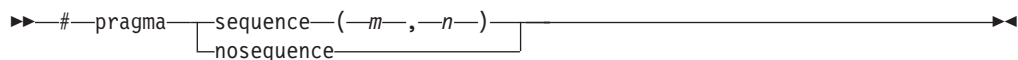
To avoid problems, you should specify #pragma runopts only in your main() routine. If you do not have a main() routine, specify it in only one other module.

## sequence

The #pragma sequence directive specifies the section of the input record that is to contain sequence numbers. The #pragma nosequence directive specifies that the input record does not contain sequence numbers.

#pragma sequence is an OS/390 C/C++ directive and an addition to the SAA Standard.

### #pragma sequence for OS/390 C



### #pragma sequence for OS/390 C++



In the syntax diagram you can specify the following parameters for OS/390 C:

- |           |   |
|-----------|---|
| <br> <br> | <p><i>m</i>      The column number of the left-hand margin. The value of <i>m</i> must be greater than 0, and less than 32761.</p> <p>Also, <i>m</i> must be less than or equal to the value of <i>n</i>.</p> |
| <br>      | <p><i>n</i>      The column number of the right-hand margin. The value of <i>n</i> must be greater than 0, and less than 32761.</p>   |

You can assign an asterisk (\*) to *n* that indicates the last column of the input record. For example, `SEQUENCE(74,*)` indicates that sequence numbers are between column 74 and the end of the input record.

You can use `#pragma sequence` and `#pragma margins` together. If they reserve the same columns, `#pragma sequence` has priority, and OS/390 C/C++ reserves the columns for sequence numbers. For example, consider if the columns reserved for the margin are 1 to 20 and the columns reserved for sequence numbers are 15 to 25. In this case, the margin will be from column 1 to 14, and the columns reserved for sequence numbers will be from 15 to 25. For more information on the `#pragma margins` directive, refer to “margins” on page 264.

The sequence setting specified by the `#pragma sequence` directive applies only to the file (source file or include file) that contains it. The setting has no effect on other `#include` files in the file. The sequence number area specified on the `#pragma sequence` directive comes into effect on the line following the directive. It remains in effect until it encounters another `#pragma sequence` or a `#pragma nosequence` directive or until it reaches the end of the file.

If you use the compile-time options `SEQUENCE|NOSEQUENCE` with the `#pragma sequence` or `#pragma nosequence` directives, the `#pragma` directive overrides the compile options. The compile option is in effect up to, and including, the `#pragma sequence` or the `#pragma nosequence` directive.

For OS/390 C++, the `#pragma sequence` directive defines that columns 73 through 80 of the input record (fixed or variable length) contain sequence numbers. You cannot specify columns (*m,n*). The default compile option for OS/390 C++ is `NOSEQUENCE`.

For OS/390 C, the default setting is `SEQUENCE(73,80)` for fixed-length records, and `NOSEQUENCE` for variable length records.

## skip (C Only)

The `#pragma skip` directive skips the specified number of lines in the generated source listing. The value of *lines* must be a positive integer less than 255. If you omit *lines*, the compiler skips one line.

```

▶▶ #pragma skip ( [lines] )

```

## strings

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

```

▶▶ #pragma strings ( [writable|writeable|readonly] )

```

## #pragma

This pragma must appear before any C or C++ code in a file.

C strings are read/write by default. C++ strings are read-only by default.

### IPA Considerations

During the IPA Link step, the compiler compares the #pragma strings specifications for individual compilation units. If it finds differences, it treats the strings as if you specified #pragma strings(writeable) for all compilation units.

## subtitle (C Only)

The #pragma subtitle directive places the text that is specified by *subtitle* on all subsequent pages of the generated source listing.

►► #pragma subtitle (—" *subtitle* "—) —————►►

## target (C Only)

The #pragma target directive specifies the operating system or run-time environment for which OS/390 C/C++ creates the object.

►► #pragma target ( 

LE
IMS

 ) —————►►

The compiler generates code to run under these options:

- |                     |  |
|---------------------|--|
| <br> <br> <br> <br> | LE       Generates code to run under the OS/390 Language Environment run-time library. This is the default behavior. |
|                     | IMS      Generates object code to run under IMS.   |

If you have more than one preprocessor directive, the only #pragma directives that can precede #pragma target are #pragma filetag, #pragma chars, #pragma langlvl, and #pragma longname.

Specifying #pragma target() or #pragma target(LE) has the following effects on #pragma runopts(ENV) and #pragma runopts(PLIST):

- If you did not specify values for #pragma runopts(ENV) or #pragma runopts(PLIST), the compiler sets the #pragmas to #pragma runopts(ENV(MVS)) and #pragma runopts(PLIST(HOST)).
- If you did specify values for #pragma runopts(ENV) or #pragma runopts(PLIST), the values do not change.

Specifying #pragma target(IMS) has the following effects on #pragma runopts(ENV) and #pragma runopts(PLIST):

- If you did not specify values for #pragma runopts(ENV) or #pragma runopts(PLIST), the compiler sets the #pragmas to #pragma runopts(ENV(IMS)) and #pragma runopts(PLIST(OS)).
- If you did specify values for #pragma runopts(ENV) or #pragma runopts(PLIST), the values do not change.



## IPA Considerations

This #pragma only affects the IPA Compile step if you specify the OBJECT suboption of the IPA compiler option.

The IPA Compile step passes the effects of this #pragma directive to the IPA Link step.

If you specify different #pragma target directives for different compilation units, the IPA Link step uses the ENV and PLIST information from the compilation unit containing main(). If there is no main(), it uses information from the first compilation unit it finds. If you specify the TARGET compile option for the IPA Link step, it overrides the #pragma target directive.

## title (C Only)

The #pragma title directive places the text that is specified by *title* on all subsequent pages of the generated source listing.

```
▶▶ #pragma title ("title") ▶▶
```

## variable

The #pragma variable directive specifies that OS/390 C/C++ is to use the named external object in either a reentrant or non-reentrant fashion. If an object is marked as RENT, its references or its definition will be in the writable static area that is in modifiable storage. If an object is marked as NORENT, its references or definition is in the code area and is in potentially read-only storage.

```
▶▶ #pragma variable (identifier, 

|        |
|--------|
| RENT   |
| NORENT |

) ▶▶
```

NORENT does not apply to, and has no affect on, program variables with static storage class. OS/390 C/C++ always includes these variables with the writable static variables. Variables are reentrant by default for C++ so that RENT has no affect.

The #pragma variable directive is an OS/390 C/C++ directive and an addition to the SAA Standard.

Refer to the section on *Reentrancy in OS/390 C/C++* in the *OS/390 C/C++ Programming Guide* for more information about reentrancy.

## wsizeof

The #pragma wsizeof directive toggles the behavior of the sizeof operator between that of the C and C++ compilers prior to and including the C/C++ MVS/ESA Version 3 Release 1 product, and the OS/390 C/C++ feature. As explained below, the difference occurs only when using sizeof on function return types. Other behaviors of sizeof remain the same.

## #pragma

Specify the pragma as follows:

►—#pragma—wsizeof—(—ON—RESUME—)——►

When using the sizeof operator, the C and C++ compilers prior to and including C/C++ MVS/ESA Version 3 Release 1, returned the size of the widened type instead of the original type for function return types. For example, in the following code fragment, using the older compilers, `i` has a value of 4.

```
char foo();
i = sizeof foo();
```

Using the OS/390 C/C++ compiler, `i` has a the value of 1, which is the size of the original type, `char`.

After a `#pragma wsizeof(on)` is encountered in a source program, all subsequent `sizeof` operators return the widened size for function return types. The behavior prior to the `#pragma wsizeof(on)`, which can be the old or current behavior, is saved. OS/390 C/C++ reinstates this saved behavior when it encounters a matching `#pragma wsizeof(resume)`. The saving action works on a stack. That is, a *resume* reinstates the most recently saved state as the following example demonstrates:

```
/* Normal behavior of sizeof to start with.          */
/* ... some code here ...                             */

#pragma wsizeof(on)      /* (1) old behavior of sizeof */
...
#pragma wsizeof(on)      /* (2) old behavior of sizeof */
...
#pragma wsizeof(resume) /* matches (2)                  */
                        /* still old behavior of sizeof */
...
#pragma wsizeof(resume) /* matches (1)                  */
                        /* normal behavior of sizeof    */
```

The compiler will match *on* and *resume* throughout the entire compile unit. That is, the effect of a `#pragma wsizeof(on)` can extend beyond a header file. Ensure the *on* and *resume* pragmas are matched in your compile unit.

**Note:** Dangling the *resume* pragma leads to undefined behavior. The effect of an unmatched *on* pragma can extend to the end of the source file.

Use the `wsizeof` pragma in old header files, where you require the old behavior of the `sizeof` operator. By guarding the header file with a `#pragma wsizeof(on)` at the start of the header, and a `#pragma wsizeof(resume)` at the end, you can use the old header file with new applications.

### Using the WSIZEOF compile option and #pragma wsizeof

The `WSIZEOF` compile option has *exactly* the same effect as inserting a `#pragma wsizeof(on)` at the beginning of the source file. If another `#pragma wsizeof` exists in the source code, OS/390 C/C++ toggles the behavior of the `sizeof` operator, as described above.

You can use the `WSIZEOF` compile option to save editing your source when you want the old behavior of the `sizeof` operator for your entire source file.

Refer to the *OS/390 C/C++ User's Guide* for information on the `WSIZEOF` compile option.

## **IPA Considerations**

During the IPA Compile step, the size of each function return value is resolved during source processing. The IPA Compile and Link steps do not alter these sizes. The IPA object code from compilation units with different `wsiz eof` settings is merged together during the IPA Link step.

**#pragma**

---

## Part 3. C++ Language Elements

This part of the Language Reference describes the language elements of C++.

### **Chapter 11. C++ Classes**

Describes the concept of classes in C++, including a description of the different types of classes, how to declare class objects, and the scoping rules for class objects.

### **Chapter 12. C++ Class Members and Friends**

Describes the scoping rules for class members and member access rules.

### **Chapter 13. C++ Overloading**

Describes the form and use of overloaded functions and overloaded operators.

### **Chapter 14. Special C++ Member Functions**

Describes the member functions that are used to create, destroy, convert, initialize, and copy class objects.

### **Chapter 15. C++ Inheritance**

Describes the concept of inheritance, including a description of access control for derived and base classes.

### **Chapter 16. C++ Templates**

Describes class templates and function templates.

### **Chapter 17. C++ Exception Handling**

Describes the facilities C++ provides for handling errors and other exceptions.



---

## Chapter 11. C++ Classes

This chapter discusses the following topics:

- “C++ Classes Overview”
- “Declaring Class Objects” on page 282
- “Scope of Class Names” on page 286

### Related Information

- “C++ Support for Object-Oriented Programming” on page 42
- “Chapter 12. C++ Class Members and Friends” on page 291
- “Chapter 15. C++ Inheritance” on page 343

---

## C++ Classes Overview

A C++ *class* is a mechanism for creating user-defined data types. It is similar to the C-language structure data type. A set of data members constitute a structure. In C++, a class type is like a C structure, except that a set of data members make up a class. In addition, in C++, you can perform an optional set of operations on the class.

In C++, you can declare a class type with the keywords `union`, `struct`, or `class`. A union object can hold any one of a named member set. Structure and class objects hold a complete set of members. Each class type represents a unique set of class members that includes data members, member functions, and other type names. The default access for members depends on the class key:

- The members of a class that is declared with the class key `class` are private by default. A class is inherited privately by default.
- The members of a class declared with the class key `struct` are public by default. A structure is inherited publicly by default.
- The members of a union that are declared with the class key `union` are public by default. You cannot use a union as a base class in derivation. “Chapter 15. C++ Inheritance” on page 343 describes base classes and derivation.

Once you create a class type, you can declare one or more objects of that class type.

For example:

```
class X
{ /* define class members here */ };
void main()
{
    X xobject1;      // create an object of class type X
    X xobject2;      // create another object of class type X
}
```

## Classes and Structures

The C++ class is an extension of the C-language structure. The only difference between a structure and a class is that structure members have public access by

## C++ Classes Overview

default and class members have private access by default. Consequently, you can use the keywords `class` or `struct` to define equivalent classes.

For example, in the following code fragment, the class `X` is equivalent to the structure `Y`:

### CBC3X10C

// In this example, class `X` is equivalent to struct `Y`

```
class X
{
    int a; // private by default
public:
    int f() { return a = 5; }; // public member function
};
struct Y
{
    int f() { return a = 5; }; // public by default
private:
    int a; // private data member
};
```

If you define a structure and then declare an object of that structure using the keyword `class`, the members of the object are still public by default. In the following example, `main()` has access to the members of `X` even though `X` is declared as using the keyword `class`:

### CBC3X10D

// This example declares a structure, then declares a class  
// that is an object of the structure.

```
#include <iostream.h>

struct x {
    int a;
    int b;
};

class x X;

void main() {
    X.a = 0;
    X.b = 1;
    cout << "Here are a and b " << X.a << " " << X.b << endl;
}
```

## Aggregate Classes

An *aggregate class* is a class that has no user-defined constructors, no private or protected members, no base classes, and no virtual functions.

“Initializers” on page 127 describes the initialization of aggregate classes.

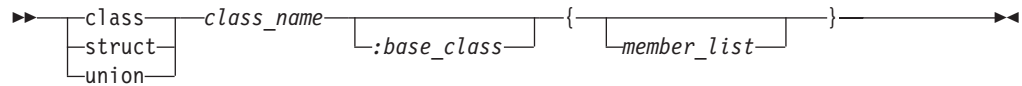
---

## Declaring Class Objects

A class declaration creates a unique type.



A *class specifier* is a type specifier that is used to declare a class. Once a class specifier has been seen and its members declared, a class is considered to be defined. This is so, even if the member functions of that class are not yet defined. A class specifier has the following form:



The *member\_list* is optional. It specifies the class members, both data and functions, of the class *class\_name*. If the *member\_list* of a class is empty, objects of that class have a nonzero size. You can use a *class\_name* within the *member\_list* of the class specifier itself, as long as you do not require the size of the class. For more information, see “Chapter 12. C++ Class Members and Friends” on page 291.

The *base\_class* is optional. It specifies the base class or classes from which the class *class\_name* inherits members. If the *base\_class* is not empty, call the class *class\_name* a *derived class*. See “Derivation” on page 346 for more information about derived classes.

The declarator for a class variable that is declared with the *class*, *struct*, or *union* keyword is an identifier. If the symbol *\** precedes the identifier, the identifier names a pointer to a class of the specified data type. If *\*\** precedes the identifier, the identifier names a pointer to a pointer to a class of the specified data type.

If a constant expression enclosed in [ ] (brackets) follows the identifier, the identifier names an array of classes of the specified data type. Consider if *\** precedes the identifier and a constant expression enclosed in [ ] follows the identifier. In that case, the identifier names an array of pointers to classes of the specified data type.

## Class Names

A *class name* is a unique identifier that becomes a reserved word within its scope. Once a class name is declared, it hides other declarations of the same name within the enclosing scope.

Consider a class name that is declared in the same scope as a function, enumerator, or object with the same name. You can refer to that class by using an *elaborated type specifier*. In the following example, the elaborated type specifier is used to refer to the class *print*. This class is hidden by the later definition of the function *print()*:

```

class print
{
    /* definition of class print */
};
void print (class print*);           // redefine print as a function
//      .                          // prefix class-name by class-key
//      .                          // to refer to class print
//      .
void main ()
{
    class print* paper;             // prefix class-name by class-key
    print(paper);                   // call function print
}

```

## Declaring Class Objects

You can use an elaborated type specifier with a class name to declare a class.

For more information on elaborated type specifiers, see “Incomplete Class Declarations” on page 287.

You can also qualify type names to refer to hidden type names in the current scope. You can reduce complex class name syntax by using a typedef to represent a nested class name.

The following example uses a typedef so that it can use the simple name nested in place of `outside::middle::inside`.

### CBC3X10B

```
// This example illustrates a typedef used to simplify
// a nested class name.

#include <iostream.h>

class outside {
public:
    class middle {
    public:
        class inside {
        private:
            int a;
        public:
            inside(int a_init = 0): a(a_init) {}
            void printa();
        };
    };
};

typedef outside::middle::inside nested;

void nested::printa() {
    cout << "Here is a " << this->a << endl;
}

void main() {
    nested n(9);
    n.printa();
}
```

For more information on nested classes, see “Nested Classes” on page 287

## Using Class Objects

You can use a class type to create instances or *objects* of that class type. For example, you can declare a class, structure, and union with class names X, Y, and Z respectively:

```
class X {          /* definition of class X */ };
struct Y {         /* definition of struct Y */ };
union Z {          /* definition of union Z */ };
```

Then you can declare objects of each class type. Remember that classes, structures, and unions are all types of C++ classes.

```
void main()
{
    X xobj;      // declare a class object of class type X
    Y yobj;      // declare a struct object of class type Y
    Z zobj;      // declare a union object of class type Z
}
```

In C++, unlike C, you do not need to precede declarations of class objects with the keywords `union`, `struct`, and `class` unless the name of the class is hidden. For example:

```
struct Y { /* ... */ };
class X { /* ... */ };
void main ()
{
    int X;          // hides the class name X
    Y yobj;         // valid
    X xobj;         // error, class name X is hidden
    class X xobj;   // valid
}
```

For more information on hidden names, see “Scope of Class Names” on page 286.

When you declare more than one class object in a declaration, the declarators are treated as if declared individually. For example, if you declare two objects that are of class `S` in a single declaration:

```
class S { /* ... */ };
//      .
//      .
//      .
void main()
{
    S S,T; // declare two objects of class type S
}
```

The above declaration is equivalent to the following:

```
class S { /* ... */ };
void main()
{
    S S;
    class S T; // keyword class is required
               // since variable S hides class type S
}
```

However, the above declaration is not equivalent to the following declaration:

```
class S { /* ... */ };
//      .
//      .
//      .
void main()
{
    S S;
    S T; // error, S class type is hidden
}
```

## Declaring Class Objects

You can also declare references to classes, pointers to classes, and arrays of classes. For example:

```
class X { /* ... */ };
struct Y { /* ... */ };
union Z { /* ... */ };
void main()
{
    X xobj;
    X &xref = xobj;           // reference to class object of type X
    Y *yptr;                 // pointer to struct object of type Y
    Z zarray[10];            // array of 10 union objects of type Z
}
```

You can assign or pass objects of class types that are not copy restricted as arguments to functions. Functions can also return these objects. For more information, see “Copy Restrictions” on page 340.

For more information on objects, see “Objects” on page 72. “Initialization by Constructor” on page 336 discusses initialization of classes.

---

## Scope of Class Names

A class declaration introduces the class name into the scope where it is declared. Any class, object, function or other declaration of that name in an enclosing scope is hidden. Consider a class name is declared in a scope where an object, function, or enumerator of the same name is also declared. In this case, you can only refer to the class by using the elaborated type specifier. The class key (class, struct, or union) must precede the class name to identify it.

For example:

### CBC3X10E

```
// This example shows the scope of class names.

class x { int a; };           // declare a class type class-name

x xobject;                   // declare object of class type x

int x(class x*)              // redefine x to be a function
{return 0;}                  // use class-key class to define
                             // a pointer to the class type x
                             // as the function argument

void main()
{
    class x* xptr;            // use class-key class to define
                             // a pointer to class type x
    xptr = &xobject;          // assign pointer
    x(xptr);                  // call function x with pointer to class x
}
```

You can use an elaborated type specifier in the declaration of objects and functions. See “Class Names” on page 283 for an example.

You can also use an elaborated type specifier in the incomplete declaration of a class type to reserve the name for a class type within the current scope.

## Incomplete Class Declarations

An *incomplete class declaration* is a class declaration that does not define any class members. You cannot declare any objects of the class type or refer to the members of a class until the declaration is complete. However, an incomplete declaration allows you to make specific references to a class prior to its definition as long as you do not require the size of the class.

For example, you can define a pointer to the structure first in the definition of the structure second. The following example declares the structure first in an incomplete class declaration prior to the definition of second. The definition of `oneptr` in structure second does not require the size of first:

```
struct first;           // incomplete declaration of struct first

struct second           // complete declaration of struct second
{
    first* oneptr;       // pointer to struct first refers to
                        // struct first prior to its complete
                        // declaration

    first one;           // error, you cannot declare an object of
                        // an incompletely declared class type

    int x, y;
};

struct first            // complete declaration of struct first
{
    second two;          // define an object of class type second
    int z;
};
```

If you declare a class with an empty member list, it is a complete class declaration. For example:

```
class X;                // incomplete class declaration
class Z {};             // empty member list
class Y
{
public:
    X yobj;              // error, cannot create an object of an
                        // incomplete class type
    Z zobj;              // valid
};
```

“Chapter 12. C++ Class Members and Friends” on page 291 describes class member lists.

## Nested Classes

You declare a *nested class* within the scope of another class. The name of a nested class is local to its enclosing class. Unless you use explicit pointers, references, or object names, declarations in a nested class can only use visible constructs. This includes type names, static members, and enumerators from the enclosing class and global variables.

Member functions of a nested class follow regular access rules and have no special access privileges to members of their enclosing classes. Member functions of the enclosing class have no special access to members of a nested class.

## Scope of Class Names

You can define member functions and static data members of a nested class in the global scope. For example, in the following code fragment, you can access the static members `x` and `y` by using a qualified type name. You can also access member functions `f()` and `g()` of the nested class `nested`. Qualified type names allow you to define a typedef to represent a qualified class name. Then you can use the typedef with the `::` (scope resolution) operator to refer to a nested class or class member.

The following example demonstrates this:

```
class outside
{
public:
    class nested
    {
public:
        static int x;
        static int y;
        int f();
        int g();
    };
};

int outside::nested::x = 5;
int outside::nested::f() { return 0; };

typedef outside::nested outnest;    // define a typedef
int outnest::y = 10;               // use typedef with ::
int outnest::g() { return 0; };    // . . .
```

## Local Classes

You declare a *local class* within a function definition. The local class is in the scope of the enclosing function scope. Declarations in a local class can only use type names, enumerations, static variables from the enclosing scope, as well as external variables and functions.

For example:

```
int x;                                // global variable
void f()                              // function definition
{
    static int y;                    // static variable y can be used by
    // local class
    int x;                          // auto variable x cannot be used by
    // local class
    extern int g();                 // extern function g can be used by
    // local class

    class local                    // local class
    {
        int g() { return x; }      // error, local variable x
        // cannot be used by g
        int h() { return y; }      // valid, static variable y
        int k() { return ::x; }    // valid, global x
        int l() { return g(); }    // valid, extern function g
    };
}

void main()
{
    local* z;                      // error, local is undefined
}
```

```
//      .
//      .
//      .
}
```

Define member functions of a local class within their class definition. Member functions of a local class must be inline functions. Like all member functions, those defined within the scope of a local class do not need the keyword `inline`.

For more information about inline functions, see “Inline Member Functions” on page 294.

A local class cannot have static data members. In the following example, an attempt to define a static member of a local class causes an error:

```
void f()
{
    class local
    {
        int f();           // error, local class has noinline
                           // member function
        int g() {return 0;} // valid, inline member function
        static int a;       // error, static is not allowed for
                           // local class
        int b;             // valid, nonstatic variable
    };
}
//      . . .
```

An enclosing function has no special access to members of the local class.

## Local Type Names

Local type names follow the same scope rules as other names. “Scope in C++” on page 46 describes scope rules. Type names that are defined within a class declaration have class scope. You cannot use them outside their class without qualification.

Consider if you use a class name, typedef name, or a constant name that is used in a type name, in a class declaration. You cannot redefine that name after the class declaration uses it.

For example:

```
void main ()
{
    typedef double db;
    struct st
    {
        db x;
        typedef int db; // error
        db y;
    };
}
```

The following declarations are valid:

```
typedef float T;
class s {
    typedef int T;
    void f(const T);
};
```

## Scope of Class Names

Here, function `f()` takes an argument of type `s::T`. However, the following declarations, which reverse the member order of `s`, cause an error:

```
typedef float T;
class s {
    void f(const T);
    typedef int T;
};
```

In a class declaration, you cannot redefine a name that is not a class name, or a typedef name to a class name or typedef name once you have used that name in the class declaration.



---

## Chapter 12. C++ Class Members and Friends

This chapter describes class members and friends, and includes the topics that are listed below:

- “Class Member Lists”
- “Data Members” on page 292
- “Class-Type Class Members” on page 292
- “Member Functions” on page 293
- “Member Scope” on page 295
- “Pointers to Members” on page 297
- “The this Pointer” on page 298
- “Static Members” on page 300
- “Member Access” on page 304
- “Friends” on page 306

### Related Information

- “Chapter 11. C++ Classes” on page 281
- “Chapter 15. C++ Inheritance” on page 343
- “Chapter 14. Special C++ Member Functions” on page 325

---

## Class Member Lists

An optional *member list* declares sub-objects called *class members*. Class members can be data, functions, classes, enumeration, bit fields, and typedef names. A member list is the only place you can declare class members. Friend declarations are not class members but must appear in member lists.

The member list follows the class name and is placed between braces. It can contain access specifiers, member declarations, and member definitions.

You can access members by using the class access `.` (dot) and `->` (arrow) operators. The class access operators are described in “Dot Operator (`.`)” on page 141 and “Arrow Operator (`->`)” on page 141.

A *member declaration* declares a class member for the class that contains the declaration. For more information on declarations, see “Chapter 5. Declarations” on page 69, and “Declaring Class Objects” on page 282.

An *access specifier* is one of the following:

- `public`
- `private`
- `protected`

“Member Access” on page 304 describes access specifiers.

## Class Member Lists

You can use a member declaration that is a qualified name followed by a ; (semicolon) to restore access to members of base classes. “Access Declarations” on page 351 describes how you do this.

A *member declarator* declares an object, function, or type within a declaration. It cannot contain an initializer. You can initialize a member by using a constructor. If the member belongs to an aggregate class, you can initialize it by using a brace initializer list in the declarator list. A brace initializer list is one that is surrounded by braces ({ }). You must explicitly initialize a class that contains constant or reference members with a brace initializer list. Or you can initialize it explicitly with a constructor.

A member declarator has the following form:

*[identifier] : constant-expression*

The above form specifies a bit field.

A *pure specifier* (= 0) indicates that a function has no definition. You can only use it with virtual member functions. It replaces the function definition of a member function in the member list. “Virtual Functions” on page 359 describes pure specifiers.

You can use the storage-class specifier `static` (but not `extern`, `auto`, or `register`) in a member list. For more information, see “Static Members” on page 300.

The order of class member mapping in a member list depends on the implementation. For the OS/390 C/C++ compiler, class members are allocated in the order they are declared.

---

## Data Members

Data members include members that are declared with any of the fundamental types, as well as other types, including pointer, reference, array types, and user-defined types. You can declare a data member the same way as a variable. However, you cannot place explicit initializers inside the class definition.

If you declare an array as a nonstatic class member, you must specify all the array dimensions.

---

## Class-Type Class Members

A class can have members that are of a class type or are pointers or references to a class type. Members that are of a class type must be of a class type that is previously declared. You can use an incomplete class type in a member declaration as long as you do not require the size of the class. For example, a member can be declared that is a pointer to an incomplete class type. For more information, see “Incomplete Class Declarations” on page 287.

A class `X` cannot have a member that is of type `X`. It can, however, contain pointers to `X`, references to `X`, and static objects of `X`. Member functions of `X` can take arguments of type `X` and have a return type of `X`. For example:

```
class X
{
    X();
```

```

    X *xptr;
    X &xref;
    static X xcount;
    X xfunc(X);
};

```

OS/390 C++ always processes the bodies of member functions *after* the definition of their class is complete. Consequently, the body of a member function can refer to the name of the class that owns it, even if this requires information about the class definition.

The language allows member functions to refer to any class member even if the member function definition appears before the declaration of that member in the class member list. For example,

```

class Y
{
public:
    int a;
    Y ();
private:
    int f() {return sizeof(Y);};
    void g(Y yobj);
    Y h(int a);
};

```

In this example, the inline function `f()` is permitted to make use of the size of class `Y`. See “Inline Member Functions” on page 294 for more information.

---

## Member Functions

*Member functions* are operators and functions that are declared as members of a class. Member functions do not include operators and functions that are declared with the friend specifier. Refer to these as *friends* of a class. For more information, see “Friends” on page 306.

The definition of a member function is within the scope of its enclosing class. OS/390 C++ analyzes the body of a member function after the class declaration so that the member function body can use members of that class. When the function `add()` is called in the following example, the data variables `a`, `b`, and `c` can be used in the body of `add()`.

```

class x
{
public:
    int add()           // inline member function add
    {return a+b+c;};
private:
    int a,b,c;
};

```

For information on static member functions, see “Static Member Functions” on page 303. For more general information on functions, see “Chapter 8. Functions” on page 173.

## const and volatile Member Functions

You can call a member function that is declared with the `const` qualifier for constant and nonconstant objects. You can call a nonconstant member function, but

## Member Functions

only for a nonconstant object. Similarly, you can call a member function that is declared with the `volatile` qualifier for volatile and nonvolatile objects. You can call a nonvolatile member function, but only for a nonvolatile object.

## Virtual Member Functions

Declare virtual member functions with the keyword `virtual`. They allow dynamic binding of member functions. Because all virtual functions must be member functions, you can refer to virtual member functions as virtual functions.

If the definition of a virtual function is replaced by a pure specifier in the declaration of the function, the function is said to be declared pure. An abstract class is one that contains at least one pure virtual function.

“Virtual Functions” on page 359 describes virtual functions in more detail.  
“Abstract Classes” on page 363 describes pure virtual functions.

## Special Member Functions

You can use *special member functions* to create, destroy, initialize, convert, and copy class objects. These include:

- Constructors
- Destructors
- Conversion constructors
- Conversion functions
- Copy constructors

Chapter 14. Special C++ Member Functions describes special member functions.

## Inline Member Functions

A member function that is both declared and defined in the class member list is called an *inline member function*. Usually, you declare member functions that contain a few lines of code as inline.

An equivalent way to declare an inline member function is to declare it outside of the class declaration using the keyword `inline` and the `::` (scope resolution) operator. These operators identify the class to which the member function belongs. For example consider the following class:

```
class Y
{
    char* a;
public:
    char* f() {return a;};
};
```

The above class is equivalent to the following class:

```
class Z
{
    char* a;
public:
    char* f();
};
//      .
//      .
//      .
inline char* Z::f() {return a;}
```

Consider when you declare an inline function without the `inline` keyword and do not define it in the class member list. In this case, you cannot call the function before you define it. In the above example, you cannot call `f()` until after its definition.

Inline member functions have internal linkage. Noninline member functions have external linkage.

For more information, see “C++ Inline Functions” on page 195.

## Member Function Templates

Any member function (inline or not inline) that is declared within a class template is implicitly a function template. When you declare a template class, the declaration implicitly generates template functions for each function that is defined in the class template. If instantiating a class template, OS/390 C++ only instantiates the function templates whose instantiations will be used by the resulting template class.

For more information about member function templates, see “Member Function Templates” on page 377.

---

## Member Scope

You can define member functions and static members outside their class declaration if you have already declared but not defined them in the class member list. By instantiating the class for nonstatic data members, you define the members. The declaration of a static data member is not a definition. The declaration of a member function is a definition if you also provide the body of the function.

Whenever the definition of a class member appears outside of the class declaration, you must qualify the member name by the class name. Use the `::` (scope resolution) operator.

The following example defines a member function outside of its class declaration.

### CBC3X11A

```
// This example illustrates member scope.

#include <iostream.h>
class X
{
public:
    int a, b;        // public data members
    int add();        // member function declaration only
};
int a = 10;          // global variable
// define member function outside its class declaration
int X::add() {return a + b;};
//
//
//
void main()
{
    int answer;
    X xobject;
```

## Member Scope

```
xobject.a = 1;
xobject.b = 2;
answer = xobject.add();
cout << xobject.a << " + " << xobject.b << " = " << answer<<endl;
}
```

The output for this example is: 1 + 2 = 3

All member functions are in class scope even if you define them outside their class declaration. In the above example, the member function `add()` returns the data member `a`, not the global variable `a`.

The name of a class member is local to its class. The class access operators are `.` (dot), `->` (arrow), or `::` (scope resolution). Unless you use one of the class access operators, you can only use a class member in a member function of its class and in nested classes. You can only use types, enumerations, and static members in a nested class without qualification with the `::` operator.

The order of search for a name in a member function body is:

1. Within the member function body itself
2. Within all the enclosing classes, including inherited members of those classes
3. Within the lexical scope of the body declaration

The search of the enclosing classes, including inherited members, is demonstrated in the following example:

```
class A { /* ... */ };
class B { /* ... */ };
class C { /* ... */ };
class Z : A {
    class Y : B {
        class X : C { int f(); /* ... */ };
    };
};
int Z::Y::X f()
{
    //      .
    //      .
    //      .
    //      j();
    //      .
    //      .
    //      .
}
```

In this example, the search for the name `j` in the definition of the function `f` follows this order:

1. In the body of the function `f`
2. In `X` and in its base class `C`
3. In `Y` and in its base class `B`
4. In `Z` and in its base class `A`
5. In the lexical scope of the body of `f` (in this case, this is global scope)

**Note:** When OS/390 C++ searches the containing classes, it only searches the definitions of the containing classes and their base classes. It does not search the scope that contains the base class definitions (global scope, in this example).

## Pointers to Members

Pointers to members allow you to refer to nonstatic members of class objects. You cannot use a pointer to member to point to a static class member because the address of a static member does not associate with any particular object. To point to a static class member, you must use a normal pointer.

You can use pointers to member functions in the same manner as pointers to functions. You can compare pointers to member functions, assign values to them, and use them to call member functions. Note that a member function does not have the same type as a nonmember function that has the same number and type of arguments and the same return type.

You can use pointers to members that are used as the following example demonstrates:

### CBC3X11B

```
// This example illustrates pointers to members.

#include <iostream.h>
class X
{
public:
    int a;
    void f(int b)
    {
        cout << "The value of b is "<< b << endl;
    }
};
// .
// .
// .
void main ()
{
    // declare pointer to data member
    int X::*ptiptr = &X::a;

    // declare a pointer to member function
    void (X::* ptfptr) (int) = &X::f;
    X xobject;           // create an object of class type X
    xobject.*ptiptr = 10; // initialize data member

    cout << "The value of a is " << xobject.*ptiptr << endl;
    (xobject.*ptfptr) (20); // call member function
}
```

The output for this example is:

```
The value of a is 10
The value of b is 20
```

## Pointers to Members

To reduce complex syntax, you can declare a typedef to be a pointer to a member. You can declare and use a pointer to a member as the following code fragment demonstrates:

```
typedef void (X::*ptfptr) (int);    // declare typedef

void main ()
{
    //      .
    //      .
    //      .
    ptfptr ptf = &X::f;
    // use typedef

    X xobject; (xobject.*ptf) (20);
                // call function
}
```

Use the pointer to member operators, `.*` and `->*`, to bind a pointer to a member of a specific class object. Because the precedence of `()` (function call operator) is higher than `.*` and `->*`, you must use parentheses to call the function pointed to by `ptf`.

For more information, see “C++ Pointer-to-Member Operators (`.*` `->*`)” on page 160.

---

## The this Pointer

The keyword `this` identifies a special type of pointer. When a nonstatic member function is called, the `this` pointer identifies the class object which the member function is operating on. You cannot declare the `this` pointer or make assignments to it.

The type of the `this` pointer for a member function of a class type `X`, is `X* const`. If the member function is declared with the constant qualifier, the type of the `this` pointer for that member function for class `X`, is `const X* const`. If the member function is declared with the volatile qualifier, the type of the `this` pointer for that member function for class `X` is `volatile X* const`.

The `this` pointer is passed as a hidden argument to all nonstatic member function calls. It is available as a local variable within the body of all nonstatic functions.

For example, you can refer to the particular class object that a member function is called by using the `this` pointer in the body of the member function. The following code example produces the output `a = 5`:

### CBC3X11C

```
// This example illustrates the this pointer

#include <iostream.h>
class X
{
    int a;
public:
    // The 'this' pointer is used to retrieve 'xobj.a' hidden by
    // the automatic variable 'a'
    void Set_a(int a) { this->a = a; }
    void Print_a() { cout << "a = " << a << endl; }
```



```
};
void main()
{
    X xobj;
    int a = 5;
    xobj.Set_a(a);
    xobj.Print_a();
}
```

Unless a class member name is hidden, using the class member name is equivalent to using the class member name qualified with the `this` pointer.

The following example shows code using class members without the `this` pointer. The comments on each line show the equivalent code with the hidden use of the `this` pointer.

## CBC3X11D

```
// This example uses class members without the this pointer.

#include <string.h>
#include <iostream.h>
#define BUFLN 100          // length of buffer to hold string
class X
{
    int len;
    char *ptr;
public:
    int GetLen()           // int GetLen (X* const this)
    { return len; }        // { return this->len; }
    char * GetPtr()        // char * GetPtr (X* const this)
    { return ptr; }        // { return this->ptr; }
    X& Set(char *);
    X& Cat(char *);
    X& Copy(X&);
    void Print();
};

X& X::Set(char *pc)       // X& X::Set(X* const this, char *pc)
{
    len = strlen(pc);     // this->len = strlen(pc);
    ptr = new char[BUFLN]; // allocate sufficient storage to hold
                          // strings in this example
    strcpy(ptr, pc);      // strcpy(this->ptr, pc);
    return *this;
}

X& X::Cat(char *pc)       // X& X::Cat(X* const this, char *pc)
{
    len += strlen(pc);    // this->len += strlen(pc);
    strcat(ptr, pc);      // strcat(this->ptr, pc);
    return *this;
}

X& X::Copy(X& x)          // X& X::Copy(X* const this, X& x)
{
    Set(x.GetPtr());      // this->Set(x.GetPtr(&x));
    return *this;
}

void X::Print()           // void X::Print(X* const this)
{
    cout << ptr << endl; // cout << this->ptr << endl;
}
```

## The this Pointer

```
void main()
{
    X xobj1;
    xobj1.Set("abcd").Cat("efgh");
    // xobj1.Set(&xobj1, "abcd").Cat(&xobj1, "efgh");

    xobj1.Print();          // xobj1.Print(&xobj1);
    X xobj2;
    xobj2.Copy(xobj1).Cat("ijkl");
    // xobj2.Copy(&xobj2, xobj1).Cat(&xobj2, "ijkl");

    xobj2.Print();          // xobj2.Print(&xobj2);
}
```

This example produces the following output:

```
abcdefgh
abcdefghijkl
```

---

## Static Members

You can declare class members by using the storage-class specifier `static` in the class member list. All the objects of a class in a program share only one copy of the static member. When you declare an object of a class that contains a static member, the static member is not part of the class object.

A typical use of static members is for recording data common to all objects of a class. For example, you can use a static data member as a counter to store the number of objects of a particular class type that are created. Each time a new object is created, this static data member can be incremented to track the total number of objects.

The declaration of a static member in the member list of a class is not a definition. The definition of a static member is equivalent to an external variable definition. You must define the static member outside of the class declaration.

For example:

```
class X
{
public:
    static int i;
};
int X::i = 0; // definition outside class declaration
//      .
//      .
//      .
```

You can access a static member from outside of its class only if you declare it with the keyword `public`. You can then access the static member by qualifying the class name by using the `::` (scope resolution) operator. In the following example, you can refer to the static member `f()` of class type `X` as `X::f()`:

```
class X
{
public:
    static int f();
};
//      .
//      .
//      .
void main ()
{
```

```

X::f();

}

```

For more information on the storage-class specifier `static`, see “static Storage Class Specifier” on page 82.

## Using the Class Access Operators with Static Members

You can also access a static member from a class object by using the class access operators `.` (dot) and `->` (arrow).

The following example uses the class access operators to access static members.

### CBC3X11E

```

// This example illustrates access to static
// members with class access operators.

#include <iostream.h>
class X
{
    static int cnt;
public:
    // The following routines all set X's static variable cnt
    // and print its value.
    void Set_Show (int i)
    {
        X::cnt = i;
        cout << "X::cnt = " << X::cnt << endl; }
    void Set_Show (int i, int j )
    {
        this->cnt = i+j;
        cout << "X::cnt = " << X::cnt << endl; }
    void Set_Show (X& x, int i)
    {
        x.cnt = i;
        cout << "X::cnt = " << X::cnt << endl; }
};
int X::cnt;
void main()
{
    X xobj1, xobj2;
    xobj1.Set_Show(11);
    xobj1.Set_Show(11,22);
    xobj1.Set_Show(xobj2, 44);
}

```

The above example produces the following output:

```

X::cnt = 11
X::cnt = 33
X::cnt = 44

```

When you access a static member through a class access operator, OS/390 C++ does not evaluate the expression to the left of the `.` or `->` operator.

You can refer to a static member independently of any association with a class object because there is only one static member that is shared by all objects of a class. A static member can exist, even if you have not declared any objects of its class.

When you access a static member, OS/390 C++ does not evaluate the expression that you use to access the member. In the following example, the external function `f()` returns class type `X`. The function `f()` can be used to access the static member

## Static Members

i of class X. The function f() itself is not called.

### CBC3X11F

```
// This example shows that the expression used to
// access a static member is not evaluated.
```

```
class X
{
public:
    static int i;
};
int X::i = 10;
X f() { /* ... */ }
void main ()
{
    int a;
    a = f().i;      // f().i does not call f()
}
```

## Static Data Members

Static data members of global classes have external linkage. You can initialize them in file scope, like other global objects. Static data members follow the usual class access rules; except that you can initialize them in file scope. Static data members and their initializers can access other static private and protected members of their class. The initializer for a static data member is in the scope of the class that declares the member.

The following example shows how you can initialize static members using other static members, even though these members are private:

```
class C {
    static int i;
    static int j;
    static int k;
    static int l;
    static int m;
    static int n;
    static int p;
    static int q;
    static int r;
    static int s;
    static int f() { return 0; }
    int a;
public:
    C() { a = 0; }
};

C c;

int C::i = C::f();      // initialize with static member function
int C::j = C::i;        // initialize with another static data member
int C::k = c.f();       // initialize with member function from an object
int C::l = c.j;         // initialize with data member from an object
int C::s = c.a;         // initialize with nonstatic data member
int C::r = 1;           // initialize with a constant value

class Y : private C {} y;

int C::m = Y::f();
int C::n = Y::r;
int C::p = y.r;         // error
int C::q = y.f();       // error
```

The initializations of C::p and C:: cause errors because y is an object of a class that is derived privately from C. Its members are not accessible to members of C.

You can only have one definition of a static member in a program. If you do not initialize a static data member, OS/390 C++ assigns a zero default value to it.

Local classes cannot have static data members.

The following example shows the declaration, initialization, use, and scope of the static data member `si` and static member functions `Set_si(int)` and `Print_si()`.

### CBC3X11G

```
// This example shows the declaration, initialization,
// use, and scope of a static data member.

#include <iostream.h>
class X
{
    int i;
    static int si;
public:
    void Set_i(int i) { this->i = i; }
    void Print_i() { cout << "i = " << i << endl; }
    // Equivalent to:
    // void Print_i(X* this)
    // { cout << "X::i = " << this->i << endl; }
    static void Set_si(int si) { X::si = si; }

    static void Print_si()
    {
        cout << "X::si = " << X::si << endl;
    }
    // Print_si doesn't have a 'this' pointer
};
int X::si = 77;      // Initialize static data member

void main()
{
    X xobj;
    // Non-static data members and functions belong to specific
    // instances (here xobj) of class X
    xobj.Set_i(11);
    xobj.Print_i();

    // static data members and functions belong to the class and
    // can be accessed without using an instance of class X
    X::Print_si();
    X::Set_si(22);
    X::Print_si();
}
```

This example produces the following output:

```
i = 11
X::si = 77
X::si = 22
```

## Static Member Functions

You cannot have static member functions and nonstatic member functions with the same names and with the same number and type of arguments.

A static member function does not have a `this` pointer. You can call a static member function using the `this` pointer of a nonstatic member function. In the following example, the nonstatic member function `printall()` calls the static member function `f()` using the `this` pointer:

## Static Members

### CBC3X11H

// This example illustrates a static member function f().

```
#include <iostream.h>
class c {
    static void f() { cout << "Here is i"
                      << i << endl;}

    static int i;
    int j;
public:
    c(int firstj): j(firstj) {}
    void printall();
};
void c::printall() {
    cout << "Here is j " << this->j << endl;
    this->f();
}
int c::i = 3;
void main() {
    class c C(0);
    C.printall();
}
```

You cannot declare a static member function with the keyword `virtual`.

A static member function can access only the names of static members, enumerators, and nested types of the class in which it is declared.

---

## Member Access

Member access determines if a class member is accessible in an expression or declaration. Note that accessibility and visibility are independent. The scoping rules of C++ determines visibility. A class member can be visible and inaccessible at the same time. This section describes how you control the access to the individual underived class members by using access specifiers when you declare class members in a member list.

## Classes and Access Control

C++ facilitates data abstraction and encapsulation by providing access control for members of class types.

For example, consider if you declare private data members and public member functions. As a consequence, a client program can only access the private members through the public member functions and friends of that class. Such a class has *data hiding* because client programs do not have access to implementation details. They are forced to use a public interface to manipulate objects of the class.

You can control access to class members by using access specifiers. In the following example, the class `abc` has three private data members `a`, `b`, and `c`, and three public member functions `add()`, `mult()`, and the constructor `abc()`. The `main()` function creates an object `danforth` of the `abc` class and then attempts to print the value of the member `a` for this object:

**CBC3X10A**

```
// This example illustrates class member access specifiers

#include <iostream.h>

class abc
{
private:
    int a, b, c;
public:
    abc(int p1, int p2, int p3): a(p1), b(p2), c(p3) {}
    int add() { return a + b + c ; }
    int mult() { return a * b * c; }
};

void main() {
    abc danforth(1,2,3);
    cout << "Here is the value of a " << danforth.a << endl;
        // This causes an error because a is not
        // a public member and cannot be accessed
        // directly
    }
```

Because class members are private by default, you can omit the keyword `private` in the definition of `abc`. Because `a` is not a public member, the attempt to access its value directly causes an error.

**Access Specifiers**

The three class member *access specifiers* have the following effect:

**public class members**

You can access them by any function, file, or class.

**private class members**

You can access them only by member functions and friends of the class in which the member is declared.

**protected class members**

You can only access them by member functions and friends of the class in which they are declared. You can also access them by member functions and friends of classes derived with public or protected access from the class in which you have declared the protected members. You can use the access specifier, `protected`, for class members that are not base members. It is, however, equivalent to `private` unless it is used in a base class member declaration, or in a base list. For more information, see “Protected Members” on page 350.

The default access for an individual class member depends on the class key that is used in the class declaration. Members of classes that are declared with the keyword `class` are private by default. Members of classes that are declared with the keyword `struct` or `union` are public by default.

The access specifier `protected` is meaningful only in the context of derivation. You can control the access to inherited members (that is, base class members) by including access specifiers in the base list of the derived class declaration. You can also restore the access to an inherited member from a derived class by using an access declaration.

“Inherited Member Access” on page 349 describes access for inherited members.

## Member Access

Member lists can include access specifiers as labels. Members that are declared after these labels have access as specified by the label they follow. An access specifier determines the access for members until another access specifier is used or until the end of the class declaration. You can use any number of access specifiers in any order.

The following example shows access specifiers in member lists.

```
class X
{
    int a;           // private data by default
public:
    void f(int);     // public function
    int b;           // public data
private:
    int c;           // private data
protected:
    void g(int);     // protected function
};
struct Y
{
    int a;           // public data by default
public:
    int b;           // public data
private:
    void g(int);     // private function
    int c;           // private data
};
```

---

## Friends

A friend of a class *X* is a function or class that is granted the same access to *X* as the members of *X*. Refer to functions that are declared with the friend specifier in a class member list as *friend functions* of that class. Refer to classes that are declared with the friend specifier in the member list of another class as *friend classes* of that class.

You must define a class *Y* before you can declare any member of *Y* as a friend of another class.

In the following example, the friend function `print` is a member of class *Y*. It accesses the private data members `a` and `b` of class *X*.

### CBC3X11I

```
// This example illustrates a friend function.

#include <iostream.h>
class X;
class Y
{
public:
    void print(X& x);
};
class X
{
public:
    X() {a=1; b=2;}
private:
    int a, b;
    friend void Y::print(X& x);
};
```



```

};
void Y::print(X& x)
{
    cout << "A is " << x.a << endl;
    cout << "B is " << x.b << endl;
}
void main ()
{
    X xobj;
    Y yobj;
    yobj.print(xobj);
}

```

You can declare an entire class as a friend.

In the following example, the friend class F has a member function print that accesses the private data members a and b of class X. It performs the same task as the friend function print in the above example. Any other members that are declared in class F also have access to all members of class X. In the example, you have not previously declared the friend class F, so the example uses an elaborated type specifier and a qualified type specifier to specify the class name.

## CBC3X11J

```

// This example illustrates a friend class.

#include <iostream.h>
class X
{
public:
    X() {a=1; b=2;}           // constructor
private:
    int a, b;
    friend class F;           // friend class
};
class F
{
public:
    void print(X& x)
    {
        cout << "A is " << x.a << endl;
        cout << "B is " << x.b << endl;
    }
//      .
//      .
//      .
};
void main ()
{
    X xobj;
    F fobj;
    fobj.print(xobj);
}

```

Both the above examples produce the following output:

```

A is 1
B is 2

```

If the you have not previously declared the class, use an elaborated type specifier and a qualified type specifier to specify the class name.

## Friends

If the friend class has been previously declared, you can omit the keyword `class`, as shown in the following example:

```
class F;
class X
{
public:
    X() {a=1; b=2;}
private:
    int a, b;
    friend F; // elaborated-type-specifier not required
};
//      .
//      .
//      .
```

## Friend Scope

The name of a friend function or class first introduced in a friend declaration is not in the scope of the class granting friendship (also called the *enclosing class*) and is not a member of the class granting friendship.

The name of a function first introduced in a friend declaration is in the scope of the first nonclass scope that contains the enclosing class. The body of a function that is provided in a friend declaration is handled in the same way as a member function defined within a class. Processing of the definition does not start until the end of the outermost enclosing class. In addition, OS/390 C++ searches the body of the function definition for unqualified names starting from the class that contains the function definition.

A class that is first declared in a friend declaration is equivalent to an extern declaration. For example:

```
class B {};
class A
{
    friend class B; // global class B is a friend of A
};
```

Consider when the name of a friend class has been introduced before the friend declaration. Then, the compiler searches for a class name that matches the name of the friend class, beginning at the scope of the friend declaration. When the declaration of a nested class is followed by the declaration of a friend class with the same name, the nested class is a friend of the enclosing class.

The scope of a friend class name is the first nonclass enclosing scope. Consider the following example:

```
class A {
    class B { // arbitrary nested class definitions
        friend class C;
    };
};
```

The above example is equivalent to the following:

```
class C;
class A {
    class B { // arbitrary nested class definitions
        friend class C;
    };
};
```

If the friend function is a member of another class, you need to use the class member access operators. For example:

```
class A
{
public:
    int f() { /* ... */ }
};
class B
{
    friend int A::f();
};
```

Any classes you derive from a base class do not inherit friends of that base class.

For more information about friend scope, see “Scope of Class Names” on page 286.

## Friend Access

A friend of a class can access the private and protected members of that class. Normally, you can only access the private members of a class through member functions of that class. In addition, you can only access the protected members of a class through member functions of a class, or classes that are derived from that class.

Access specifiers do not affect friend declarations.

For more information on access, see “Member Access” on page 304.

## Friends

---

## Chapter 13. C++ Overloading

*Overloading* enables you to redefine functions and most standard C++ operators. Typically, you overload a function or operator if you want to extend the operations the function or operator performs to different data types. This chapter discusses the following topics on overloading:

- “Overloading Functions”
- “Argument Matching in Overloaded Functions” on page 312
- “Overloading Operators” on page 315
- “Overloading Unary Operators” on page 317
- “Overloading Binary Operators” on page 318
- “Special Overloaded Operators” on page 319

### Related Information

- “Chapter 6. Expressions and Operators” on page 133
- “Chapter 8. Functions” on page 173
- “Chapter 11. C++ Classes” on page 281

---

## Overloading Functions

You can overload a function by having multiple declarations of the same function name in the same scope. The declarations differ in the type and number of arguments in the argument list. When you call an overloaded function, OS/390 C++ compares the types of the actual arguments with the types of the formal arguments. Thus, it selects the correct function.

Consider a function `print`, which displays an `int`. The following example demonstrates that you can overload this function to display other types, for example, `double`, and `char*`. You can have three functions with the same name, each performing a similar operation on a different data type.

### CBC3X12A

```
// This example illustrates function overloading.

#include <iostream.h>

void print(int i) { cout << " Here is int " << i << endl; }
void print(double f) { cout << " Here is float "
                        << f << endl; }
void print(char* c) { cout << " Here is char* " << c << endl; }
void main() {
    print(10);           // calls print(int)
    print(10.10);        // calls print(double)
    print("ten");        // calls print(char*)
}
```

### Declaration Matching

Two function declarations are identical if all of the following are true:

- They have the same function name
- They are declared in the same scope
- They have identical argument lists

When you declare a function name more than once in the same scope, the compiler interprets the second declaration of the function name as follows:

- If the return type, argument types, and number of arguments are identical for the two declarations, the compiler considers that the second declaration is the same as the first.
- If only the return types of the two function declarations differ, the second declaration is an error.
- If either the argument types or number of arguments of the two declarations differ, the function is an overloaded function.

### Restrictions on Overloaded Functions

- Functions that differ only in return type cannot have the same name.
- Two member functions that differ only in that one is declared with the keyword `static` cannot have the same name.
- A typedef is a synonym for another type, not a separate type. The following two declarations of `spadina()` are declarations of the same function:

```
typedef int I;  
void spadina(float, int);  
void spadina(float, I);
```

- A member function of a derived class is not in the same scope as a member function in a base class with the same name. A derived class member hides a base class member with the same name.
- Argument types that differ only in that one is a pointer `*` and the other is an array `[]` are identical. The following two declarations are equivalent:

```
f(char*);  
f(char[10]);
```

Only the second and subsequent array dimensions are significant.

- The `const` and `volatile` type-specifiers are ignored in distinguishing argument types when they appear at the outermost level of the argument type specification. The following declarations are equivalent:

```
int f (int);  
int f (const int);  
int f (volatile int);
```

Pointers and references to types are considered distinct parameter types.

For more information on functions, see “Chapter 8. Functions” on page 173.

---

## Argument Matching in Overloaded Functions

When you call an overloaded function or overloaded operator, the compiler chooses the function declaration with the *best match*. This match is based on all arguments from all the visible function declarations. The compiler compares the actual arguments of a function call with the formal arguments of all visible declarations of the function. For a best match to occur, the compiler must be able to distinguish a function that:

- Has at least as good a match on all arguments as any other function with the same name
- Has at least one better argument match than any other function with the same name

If no such function exists, OS/390 C++ does not allow the call. A call to an overloaded function has three possible outcomes. The compiler can find:

- An exact match
- No match
- An ambiguous match

An ambiguous match occurs when the actual arguments of the function call match more than one overloaded function.

The matching of arguments includes performing standard and user-defined conversions on the arguments in order to match the actual arguments with the formal arguments. OS/390 C++ only performs a single user-defined conversion in a sequence of conversions on an actual argument. In addition, OS/390 C++ performs the *best-matching* sequence of standard conversions on an actual argument. The best-matching sequence is the shortest sequence of conversions between two standard types. For example, you can shorten the following conversion:

```
int -> float -> double
```

to the best-matching conversion sequence:

```
int -> double
```

In the above example, the compiler allows the conversion from `int` to `double`.

Trivial conversions, that are described in “Trivial Conversions” on page 314, do not affect the choice of conversion sequence.

## Sequence of Argument Conversions

Argument-matching conversions occur in the following order:

1. An exact match, in which the actual arguments exactly match the type and number of formal arguments of one declaration of the overloaded function. This includes a match with one or more trivial conversions.
2. A match with promotions in which a match is found when one or more of the actual arguments is promoted
3. A match with standard conversions in which a match is found when one or more of the actual arguments is converted by a standard conversion
4. A match with user-defined conversions in which a match is found when one or more of the actual arguments is converted by a user-defined conversion
5. A match with ellipses

A match through promotion follows the rules for Integral Promotions and “Standard Type Conversions” on page 167.

You can override an exact match by using an explicit cast. In the following example, the second call to `f()` matches with `f(void*)`:

```
void f(int);  
void f(void*);  
//      .
```

## Argument Matching

```
//      .  
//      .  
void main()  
{  
    f(0xaabb);           // matches f(int);  
    f((void*) 0xaabb);   // matches f(void*)  
}
```

The implicit first argument for a nonstatic member function or operator is the `this` pointer. It refers to the class object for which the member function is called. When you overload a nonstatic member function, the first implicit argument, the `this` pointer, is matched with the object or pointer used in the call to the member function. User-defined conversions are not applied in this type of matching of arguments for overloaded functions or operators.

When you call an overloaded member function of class `X` using the `.` (dot) or `->` (arrow) operator, the `this` pointer has type `X* const`. The type of the `this` pointer for a constant object is `const X* const`. The type of the `this` pointer for a volatile object is `volatile X* const`.

See “The `this` Pointer” on page 298 for information on the `this` pointer. See “Dot Operator (`.`)” on page 141 and “Arrow Operator (`->`)” on page 141 for information on the class-member access operators.

## Trivial Conversions

The compiler cannot distinguish between functions if they have the same name and arguments which differ only in that one is declared as a reference to a type, and the other is that type. You cannot have two functions with the same name and with arguments that differ only in this respect. Because the following two declarations cannot be distinguished, the second one causes an error:

```
double f(double i); // declaration  
//      .  
//      .  
//      .  
double f(double &i); // error
```

Functions with the same name and arguments can be distinguished only if they have following differences::

- One is a pointer or reference, and the other is a pointer to `const` or `const` reference.
- One is a pointer or reference, and the other is a pointer to `volatile` or `volatile` reference.

To find a best match of arguments, functions with a `volatile` or `const` match (not requiring a trivial conversion) are better than those that have a `volatile` or `const` mismatch.

For more information on conversions, see “Standard Type Conversions” on page 167 and “User-Defined Conversions” on page 334.



## Overloading Operators

You can overload one of the standard C++ operators by redefining it to perform a particular operation when you apply it to an object of a particular class. Overloaded operators must have at least one argument that has class type. OS/390 C++ calls an overloaded operator an *operator function*. You declare it with the keyword `operator` that precedes the operator itself. Overloaded operators are distinct from overloaded functions. Like overloaded functions however, you distinguish them by the number and types of operands you used with the operator.

You can overload any of the following operators:

+	-	*	/	%	^	&		~
!	=	<	>	+=	--	*=	/=	%=
^=	&=	=	<<	>>	<<=	>>=	==	!=
<=	>=	&&		++	--	,	->*	->
( )	[ ]	new	delete					

where `()` is the function call operator and `[]` is the subscript operator.

Consider the standard `+` (plus) operator. When you use this operator with operands of different standard types, the operators have slightly different meanings. For example, C++ does not implement the addition of two integers in the same way as the addition of two floating-point numbers. C++ allows you to define your own meanings for the standard C++ operators when you apply them to class types. The following example defines a class that is called `complx` to model complex numbers. The example redefines the `+` (plus) operator in this class to add two complex numbers.

### CBC3X12B

```
// This example illustrates overloading the plus (+) operator.

#include <iostream.h>
class complx
{
    double real,
          imag;
public:
    complx( double real = 0., double imag = 0.); // constructor
    complx operator+(const complx&) const;      // operator+()
};
// define constructor
complx::complx( double r, double i )
{
    real = r; imag = i;
}
// define overloaded + (plus) operator
complx complx::operator+ (const complx& c) const
{
    complx result;
    result.real = (this->real + c.real);
    result.imag = (this->imag + c.imag);
    return result;
}
```

## Overloading Operators

```
void main()
{
    complx x(4,4);
    complx y(6,6);
    complx z = x + y; // calls complx::operator+()
}
```

### General Rules for Overloading Operators

You can overload both the unary and binary forms of:

+      -      \*      &

When an overloaded operator is a member function, C++ matches the first operand against the class type of the overloaded operator. It matches the second operand, if one exists, against the argument in the overloaded operator call.

When an overloaded operator is a nonmember function, at least one operand must have class or enumeration type. OS/390 C++ matches the first operand against the first argument in the overloaded operator call. It matches the second operand, if one exists, against the second argument in the overloaded operator call.

The argument-matching conventions and rules that are described in “Argument Matching in Overloaded Functions” on page 312 apply to overloaded operators.

### Operands of Overloaded Operators

An overloaded operator must be a member function, or take at least one argument of class, a reference to a class, an enumeration, or a reference to an enumeration. The following example demonstrates the operator as a member function:

```
class X
{
public:
    X operator!();
    X& operator =(X&);
    X operator+(X&);
};
X X::operator!() { /* ... */ }
X& X::operator=(X& x) { /* ... */ }
X X::operator+(X& x) { /* ... */ }
```

The following example shows the other case:

```
class Y;
{
//      .
//      .
//      .
};
class Z;
{
//      .
//      .
//      .
};
Y operator!(Y& y);
Z operator+(Z& z, int);
```

Usually, you invoke overloaded operators by using the normal operator syntax. You can also call overloaded operators explicitly by qualifying the operator name.

For example, for the class `complx`, as described above, you can call the overloaded `+` (plus) operator either implicitly or explicitly. The following example demonstrates this:

### CBC3X12C

```
// This example shows implicit and explicit calls
// to an overloaded plus (+) operator.

class complx
{
    double real,
          imag;
public:
    complx( double real = 0., double imag = 0.);
    complx operator+(const complx&) const;
};
//      .
//      .
//      .
void main()
{
    complx x(4,4);
    complx y(6,6);
    complx u = x.operator+(y); // explicit call
    complx z = x + y;          // implicit call to complx::operator+()
}
```

## Restrictions on Overloaded Operators

- You cannot overload the following C++ operators:  
`.` `.*` `::` `?:`
- You cannot overload the preprocessing symbols `#` and `##`.
- You cannot change the precedence, grouping, or number of operands of the standard C++ operators.
- An overloaded operator (except for the function call operator) cannot have default arguments or an ellipsis in the argument list.
- You must declare the overloaded `=`, `[]`, `()`, and `->` operators as nonstatic member functions to ensure that they receive lvalues as their first operands.
- The operators `new` and `delete` do not follow the general rules described in this section. Overloading `new` and `delete` is described in “Overloaded new and delete” on page 322.
- All operators except the `=` operator are inherited. “Copy by Assignment” on page 341 describes the behavior of the assignment operator.
- Unless explicitly mentioned in “Special Overloaded Operators” on page 319, overloaded unary, and binary operators follow the rules that are outlined in “Overloading Unary Operators” and “Overloading Binary Operators” on page 318.

For more information on standard C and C++ operators, see “Overloading Unary Operators”.

---

## Overloading Unary Operators

You can overload a prefix unary operator by declaring a nonmember function that takes one argument, or a nonstatic member function that takes no arguments.

When you prefix a class object with an overloaded unary operator, you can interpret the function call `!x` as: `x.operator!()` or `operator!(x)`. For example:

## Overloading Unary Operators

```
class X
{
//      .
//      .
//      .
};
void main ()
{
    X x;
    !x;      // overloaded unary operator
}
```

However, this depends on the declarations of the operator function. If you have declared both forms of the operator function, argument matching determines which interpretation OS/390 C++ uses.

For more information on standard unary operators, see “Unary Expressions” on page 142.

---

## Overloading Binary Operators

You can overload a binary operator by declaring a nonmember function that takes two arguments, or a nonstatic member function that takes one argument.

When you use a class object with an overloaded binary operator, for example:

```
class X
{
//      .
//      .
//      .
};
void main ()
{
    X x;
    int y=10;
    x*y;      // overloaded binary operator
}
```

you can interpret the operator function call `x*y` as:

`x.operator*(y)`

or

`operator*(x,y)`

depending on the declarations of the operator function. If you have declared both forms of the operator function, argument matching determines which interpretation OS/390 C++ uses.

For more information on standard binary operators, see “Binary Expressions” on page 152.

## Special Overloaded Operators

The following overloaded operators do not fully follow the rules for unary or binary overloaded operators:

- Assignment
- Function call
- Subscripting
- Class member access
- Increment and decrement
- new and delete

## Overloaded Assignment

You can only overload an assignment operator by declaring a nonstatic member function. The following example shows how you can overload the assignment operator for a particular class:

```
class X
{
public:
    X();
    X& operator=(X&);
    X& operator=(int);

    //      .
    //      .
    //      .
};
X& X::operator=(X& x) { /* ... */ }
X& X::operator=(int i) { /* ... */ }
//      .
//      .
//      .
void main()
{
    X x1, x2;
    x1 = x2;      // call x1.operator=(x2)
    x1 = 5;       // call x1.operator=(5)
}
```

You cannot declare an overloaded assignment operator that is a nonmember function.

Overloaded assignment operators are not inherited.

If you do not define a copy assignment operator function for a class, OS/390 C++ defines the function by default as a memberwise assignment of the class members. The compiler uses these operators when it generates default copy assignment operators, if assignment operator functions exist for the base classes or class members. See “Copy by Assignment” on page 341 for more information.

For more information on standard assignment operators, see “Assignment Expressions” on page 162.

## Overloaded Function Calls

The operands are *function\_name* and an optional *expression\_list*. The operator function `operator()` must be defined as a nonstatic member function. You cannot declare an overloaded function call operator that is a nonmember function.

## Special Overloaded Operators

If you make the following call for the class object *x*:

```
x (arg1, arg2, arg3)
```

OS/390 C++ interprets it as:

```
x.operator()(arg1, arg2, arg3)
```

You can provide default arguments and ellipses in the argument list for the function call operator, unlike all other overloaded operators. For example:

```
class X
{
public:
    X& operator() (int = 5);
};
//      .
//      .
//      .
```

For more information on the standard function call operator, see “Function Calls ( )” on page 139.

## Overloaded Subscripting

An expression that contains the subscripting operator has the following syntax of the form, and OS/390 C++ considers it a binary operator.:

*identifier* [ *expression* ]

The operands are *identifier* and *expression*. The operator function `operator[]` must be defined as a nonstatic member function. You cannot declare an overloaded subscript operator that is a nonmember function.

A subscripting expression for the class object *x*:

```
x [y]
```

is interpreted as `x.operator[] (y)`. It is not interpreted as `operator[] (x,y)` because it is defined as a nonstatic member function.

For more information on the standard subscripting operator, see “Array Subscript [ ] (Array Element Specification)” on page 140.

## Overloaded Class Member Access

An expression containing the class member access `->` (arrow) operator has the following syntax, and is considered a unary operator:

*identifier* `->` *name-expression*

The operator function `operator->()` must be defined as a nonstatic member function.

The following restrictions apply to class member access operators:

- You cannot declare an overloaded arrow operator that is a nonmember function.
- You cannot overload the class member access `.` (dot) operator.

Consider the following example of overloading the `->` (arrow) operator:

```

class Y
{
public:
    void f();
};
class X
{
public:
    Y* operator->();
};
X x;
//      .
//      .
//      .
x->f();

```

Here `x->f()` is interpreted as:

```
( x.operator->() )-> f()
```

`x.operator->()` must return either a reference to a class object, or a class object for which the overloaded `operator->` function is defined, or a pointer to any class. If the overloaded `operator->` function returns a class type, the class type must not be the same as the class that declares the function. The class type that is returned must contain its own definition of an overloaded `->` operator function.

For more information on the standard class member access arrow operator, see “Arrow Operator (`->`)” on page 141.

## Overloaded Increment and Decrement

You can overload the prefix increment operator (`++`) for a class type by declaring a nonmember function operator with one argument of class type or a reference to class type. You can also overload it by declaring a member function operator with no arguments.

The following example illustrates both ways of overloading the increment operator:

### CBC3X12D

// This example illustrates an overloaded prefix increment operator.

```

class X
{
    int a;
public:
    operator++();    // member prefix increment operator
};
class Y { /* ... */ };
operator++(Y& y);    // nonmember prefix increment operator
//      .
//      .
//      .
// Definitions of prefix increment operator functions
//      .
//      .
//      .

void main()
{
    X x;
    Y y;
    ++x;                // x.operator++

```

## Special Overloaded Operators

```
x.operator++();    // x.operator++
operator++(y);    // nonmember operator++
++y;              // nonmember operator++
}
```

You can overload the postfix increment operator `++` for a class type by declaring a nonmember function `operator operator++()` with two arguments. The first argument has class type, and the second has type `int`. Alternatively, you can declare a member function `operator operator++()` with one argument having type `int`. The compiler uses the `int` argument to distinguish between the prefix and postfix increment operators. For implicit calls, the default value is zero.

For example:

### **CBC3X12E**

// This example illustrates an overloaded postfix increment operator.

```
class X
{
    int a;
public:
    operator++(int);    // member postfix increment operator
};
operator++(X x, int i); // nonmember postfix increment operator
//      .
//      .
//      .
// Definitions of postfix increment operator functions
//      .
//      .
//      .
void main()
{
    X x;
    x++;                // x.operator++
                        // default zero is supplied by compiler
    x.operator++(0);    // x.operator++
    operator++(x,0);    // nonmember operator++
}
```

The prefix and postfix decrement operators follow the same rules as their increment counterparts.

For more information on the standard postfix and prefix increment operators, see “Increment (`++`)” on page 142. For more information on the standard postfix and prefix decrement operators, see “Decrement (`--`)” on page 143.

## Overloaded new and delete

You can implement your own memory management scheme for a class by overloading the operators `new` and `delete`. The overloaded operator `new` must return a `void*`, and its first argument must have type `size_t`. The overloaded operator `delete` must return a `void` type, and its first argument must be `void*`. The second argument for the overloaded `delete` operator is optional, and if present, it must have type `size_t`. You can only define one `delete` operator function for a class.

Type `size_t` is an implementation dependent unsigned integral type defined in `<stddef.h>`.



## Special Overloaded Operators

This example requires a size argument because a class can inherit an overloaded new operator. The derived class can be a different size than the base class. The size argument ensures that OS/390 C++ allocates or deallocates the correct amount of storage space for the object.

When new and delete are overloaded within a class declaration, they are static member functions whether they are declared with the keyword `static` or not. They cannot be virtual functions.

You can access the standard, nonoverloaded versions of new and delete within a class scope containing the overloading new and delete operators by using the `::` (scope resolution) operator to provide global access.

For more information on the class member operators new and delete, see “Free Store” on page 330. For more information on the standard new and delete operators, see “C++ new Operator” on page 147 and “C++ delete Operator” on page 151.



---

## Chapter 14. Special C++ Member Functions

This chapter introduces the special member functions that are used to create, destroy, convert, initialize, and copy class objects. It includes the following topics:

- “Constructors and Destructors Overview”
- “Constructors” on page 326
- “Destructors” on page 328
- “Free Store” on page 330
- “Temporary Objects” on page 333
- “User-Defined Conversions” on page 334
- “Initialization by Constructor” on page 336
- “Copying Class Objects” on page 340

### Related Information

- “Chapter 8. Functions” on page 173
- “Chapter 11. C++ Classes” on page 281
- “Chapter 12. C++ Class Members and Friends” on page 291
- “Chapter 13. C++ Overloading” on page 311
- “Chapter 15. C++ Inheritance” on page 343

---

## Constructors and Destructors Overview

Because classes have complicated internal structures, including data and functions, object initialization, and cleanup for classes is much more complicated than it is for simple data structures. Constructors and destructors are special member functions of classes that are used to construct and destroy class objects. Construction may involve memory allocation and initialization for objects. Destruction may involve cleanup and deallocation of memory for objects.

Like other member functions, constructors and destructors are declared within a class declaration. They can be defined inline or external to the class declaration. Constructors can have default arguments. Constructors can have member initialization lists, unlike other member functions. The following restrictions apply to constructors and destructors:

- Constructors and destructors do not have return types nor can they return values.
- You cannot use references and pointers on constructors and destructors because you cannot take their addresses.
- You cannot declare constructors with the keyword `virtual`.
- You cannot declare constructors and destructors as `static`, `const`, or `volatile`.
- Unions cannot contain class objects that have constructors or destructors.

Constructors and destructors obey the same access rules as member functions. For example, if you declare a constructor with the keyword `protected`, only derived classes and friends can use it to create class objects. “Member Access” on page 304 describes class member access.

## Constructors and Destructors Overview

The compiler automatically calls constructors when defining class objects and calls destructors when class objects go out of scope. A constructor does not allocate memory for the class object to which its `this` pointer refers. It may, however, allocate storage for more objects than its class object refers to. If objects require memory allocation, constructors can explicitly call the `new` operator. During cleanup, a destructor may release objects that are allocated by the corresponding constructor. To release objects, use the `delete` operator. The global `new` and `delete` operators are described in “C++ new Operator” on page 147 and “C++ delete Operator” on page 151.

Derived classes do not inherit constructors or destructors from their base classes, but they do call the constructor and destructor of base classes. You can declare destructors with the keyword `virtual`.

Constructors are also called when local or temporary class objects are created. Destructors are called when local or temporary objects go out of scope.

You can call member functions from constructors or destructors. You can call a virtual function, either directly or indirectly, from a constructor or destructor. In this case, the function that you call is the one defined in the class or base class containing the constructor (or destructor). However, it is not a function defined in any class derived from the class you are constructing. Therefore, you cannot access an object from a constructor or destructor, if that object is not constructed.

---

## Constructors

A *constructor* is a member function with the same name as its class. For example:

```
class X
{
public:
    X();      // constructor for class X
    // .
    // .
    // .
};
```

You can use constructors to create and initialize objects of their class type. “Initialization by Constructor” on page 336 describes the initialization of class objects using constructors.

## Default Constructors

A *default constructor* is a constructor that either has no arguments, or, if it has arguments, *all* the arguments have default values. If no user-defined constructor exists for a class and your program needs one, the compiler creates a default constructor, with public access, for that class. The compiler does not create a default constructor for a class that has any constant members or reference type members.

A constructor can have default arguments, like all functions. You can use them to initialize member objects. If the call to the constructor supplies default values, you can omit the trailing arguments in the expression list of the constructor. For more information, see “Default Arguments in C++ Functions” on page 190. Note that if a constructor has any arguments that do not have default values, it is not a default constructor.

## Copy Constructors

Use a *copy constructor* to make a copy of one class object from another class object of the same class type. You call a copy constructor with a single argument that is a reference to its own class type. You cannot use a copy constructor with an argument of the same type as its class; you must use a reference. You can provide copy constructors with additional default arguments. If a user-defined copy constructor does not exist for a class and your program needs one, the compiler creates a copy constructor, with public access, for that class. The compiler does not create a copy constructor for a class if any of its members or base classes have an inaccessible copy constructor.

The following code fragment shows two classes with constructors, default constructors, and copy constructors:

```
class X
{
public:
    X();                // default constructor, no arguments
    X(int, int , int = 0); // constructor
    X(const X&);        // copy constructor
    X(X);              // error, incorrect argument type
};
class Y
{
public:
    Y( int = 0);        // default constructor with one
                        // default argument
    Y(const Y&, int = 0); // copy constructor
};
```

## Construction Order of Class Objects

If a class contains a base class or members with constructors when it is constructed, OS/390 C++ calls the constructor for the base class. Next, it calls any constructors for members. It calls the constructor for the derived class last. The compiler constructs virtual base classes before nonvirtual base classes. When more than one base class exists, the compiler calls base class constructors in the order that their classes appear in the base list, as the following example demonstrates.

```
class B1 { public: B1(); };
class B2
{
public:
    B2();
    B1 blobj;
};
class B3 { public: B3(); };
//      .
//      .
//      .
class D : public B1, public B2, public B3
{
public:
    D();
    ~D();
};
//      .
//      .
//      .
```

## Constructors

```
void main ()
{
    D object;
}
```

In the above example, the compiler calls constructors for object in the following order:

```
B1();    // first base constructor declared
B1();    // member constructor for B2::blobj
B2();    // second base constructor declared
B3();    // last base constructor declared
D();     // derived constructor called last
```

Note that the construction of class D involves construction of the base classes B1, B2, and B3. The construction of base class B2 involves the construction of its class B1 member object. When OS/390 C++ constructs class B2, it calls the constructor for class B1 in addition to B2's own constructor.

The second call to the constructor of B1 followed by the call to the constructor of B2 is part of the construction of B2.

For more information, see "Construction Order of Derived Class Objects" on page 339.

## Explicitly Constructing Objects

You cannot call constructors directly. Use a function style cast to explicitly construct an object of the specified type. The following example uses a constructor as an initializer to create a named object.

```
#include <iostream.h>
class X
{
public:
    X (int, int , int = 0); // constructor with default argument
private:
    int a, b, c;
    int f();
};
X::X (int i, int j, int k) { a = i; b = j; c = k; }
//      .
//      .
//      .
void main ()
{
    X xobject = X(1,2,3); // explicitly create and initialize
                        // named object with constructor call
}
```

---

## Destructors

A *destructor* is a member function with the same name as its class that is prefixed by a ~ (tilde).

For example:

```
class X
{
public:
    X();           // constructor for class X
    ~X();          // destructor for class X
    // .
    // .
    // .
};
```

A destructor takes no arguments and has no return type. You cannot take its address. You cannot declare destructors as `const`, `volatile`, or `static`. You can declare a destructor as `virtual` or `pure virtual`. A union cannot have as a member an object of a class with a destructor.

Use destructors to deallocate memory and do other cleanup for a class object and its class members when you destroy the object. OS/390 C++ calls a destructor for a class object when that object passes out of scope or you explicitly delete it.

Class members that are class types can have their own destructors. Both base and derived classes can have destructors, although destructors are not inherited. Consider that a base class or a member of a base class has a destructor and a class derived from that base class does not declare a destructor. In that case, OS/390 C++ generates a default destructor. The default destructor calls the destructors of the base class and the members of the derived class. OS/390 C++ generates default destructors with default public access.

The compiler calls destructors in the reverse order to which it calls constructors:

1. It calls the destructor for a class object before it calls destructors for members and bases.
2. It calls destructors for nonstatic members before calling destructors for base classes.
3. It calls destructors for nonvirtual base classes before calling destructors for virtual base classes.

When your program throws an exception for a class object with a destructor, OS/390 C++ does not call the destructor for the temporary object that it throws until your program passes control out of the catch block. For more information, see “Constructors and Destructors in Exception Handling” on page 391.

OS/390 C++ calls destructors implicitly when an automatic or temporary object passes out of scope. It also calls them implicitly at program termination for constructed external and static objects. Destructors are invoked when you use the `delete` operator for objects that are created with the `new` operator.

## Destructors

For example:

```
#include <string.h>
class Y
{
private:
    char * string;
    int number;
public:
    Y(const char* n,int a); // constructor
    ~Y() { delete[] string; } // destructor
};
Y::Y(const char* n, int a) // define class Y constructor
{
    string = strcpy(new char[strlen(n) + 1 ], n);
    number = a;
}
void main ()
{
    Y yobj = Y("somestring", 10); // create and initialize
                                // object of class Y
    //      .
    //      .
    //      .
    // destructor ~Y is called before control returns from main()
}
```

Although you can use a destructor explicitly to destroy objects, you should not use this method. If an object has been placed at a specific address by the new operator, you can call the destructor of the object to destroy it. An explicitly called destructor cannot delete storage.

**Note:** You can only call destructors for class types. You cannot call destructors for simple types. The call to the destructor in the following example causes the compiler to issue a warning:

```
int * ptr;
ptr -> int::~~int(); // warning
```

---

## Free Store

Use *free store* to dynamically allocate memory. The new and delete operators are used to allocate and deallocate free store, respectively. You can define your own versions of new and delete for a class by overloading them. You can supply the new and delete operators with additional arguments. When new and delete operate on class objects, the class member operator functions new and delete are called, if they have been declared.

If you create a class object with the new operator, one of the operator functions operator new() or operator new[]() (if they have been declared) is called to create the object. An operator new() or operator new[]() for a class is always a static class member, even if it is not declared with the keyword static. It has a return type void\*, and its first argument must be the size of the object type and have type size\_t. It cannot be virtual.

Type size\_t is an implementation dependent unsigned integral type defined in <stddef.h>.

When you overload the new operator, you must declare it as a class member, that returns type void\*, with first argument size\_t, as described above. You supply



additional arguments in the declaration of operator new() or operator new[](). Use the placement syntax to specify values for these arguments in an allocation expression.

The following example shows how to use the overloaded new and delete operators and the overloaded new and delete vector operators.

```
#include <new.h>
#include <stdio.h>

char buff[10000];
char* cur=buff;

void* operator new(size_t bytes) {           // allocate storage for object
    printf(" In new\n");
    char* prv = cur;
    cur += bytes;
    return(prv);
}

void operator delete(void* ptr) {             // free storage for object
    printf(" In delete\n");
}

void* operator new[](size_t bytes) {         // allocate storage for array
    printf(" In new[]\n");                   // objects
    char* prv = cur;
    cur += bytes;
    return(prv);
}

void operator delete[](void* ptr) {          // free storage for array of
    printf(" In delete[]\n");               // objects
}

class X {
public:
    X() { printf("X constructed\n"); }       // default constructor for X
    ~X() { printf("X destroyed\n"); }        // default destructor for X
};

main() {
    // create, then delete different types of objects and
    // arrays-of-objects

    printf("New int\n");
    int* flat = new int;
    printf("New array-of-int\n");
    int* arr = new int[2];

    printf("New X\n");
    X* x = new X;
    printf("New array-of-X\n");
    X* xArr = new X[2];

    printf("Delete array-of-X\n");
    delete[] xArr;
    printf("Delete X\n");
    delete x;
    printf("Delete array-of-int\n");
    delete[] arr;
    printf("Delete int\n");
    delete flat;

    return(0);
}
```

## Free Store

The example produces the following output:

```
New int
  In new
New array-of-int
  In new[]
New X
  In new
X constructed
New array-of-X
  In new[]
X constructed
X constructed
Delete array-of-X
X destroyed
X destroyed
  In delete[]
Delete X
X destroyed
  In delete
Delete array-of-int
  In delete[]
Delete int
  In delete
```

The delete operator destroys an object that is created by the new operator. The operand of delete must be a pointer returned by new. If you call delete for an object with a destructor, OS/390 C++ invokes the destructor before it deallocates the object.

If you destroy a class object with the delete operator, the operator function operator delete() or operator delete[] () (if they have been declared) is called to destroy the object. An operator delete() or operator delete[] () for a class is always a static member, even if it is not declared with the keyword static. Its first argument must have type void\*. Because operator delete() and operator delete[] () have a return type void, they cannot return a value. They cannot be virtual.

When you overload the delete operator, you must declare it as a class member, returning type void. Its first argument must be type void\*, as described above. You can add a second argument of type size\_t to the declaration. You can only have one operator delete() or operator delete[] () for a single class.

Overloading new and delete is described in “Overloaded new and delete” on page 322.

The following example shows the declaration and use of the operator functions `operator new()` and `operator delete()`:

```
#include <stddef.h>
class X
{
public:
    void* operator new(size_t);
    void operator delete(void*);           // single argument
};
class Y
{
public:
    void operator delete(void*, size_t); // two arguments
};
//      .
//      .
//      .
void main ()
{
    X* ptr = new X;
    delete ptr;           // call X::operator delete(void*)
    Y* yptr;
//      .
//      .
//      .
    delete yptr;          // call Y::operator delete(void*, size_t)
                          // with size of Y as second argument
}
```

The result of trying to access a deleted object is undefined because the value of the object can change after deletion.

Consider if you call `new` and `delete` for a class object that does not declare the operator functions `new` and `delete`. Or, consider if you call them for a nonclass object. In these cases, OS/390 C++ uses the global operators `new` and `delete`. The global operators `new` and `delete` are provided in the C++ library.

**Note:** The C++ operators for allocating and deallocating arrays of class objects are `operator new[ ]()` and `operator delete[ ]()`. They are described in “C++ new Operator” on page 147 and “C++ delete Operator” on page 151.

---

## Temporary Objects

It is sometimes necessary for the compiler to create temporary objects. It uses these objects during reference initialization and during evaluation of expressions that includes standard type conversions, argument passing, function returns, and evaluation of the `throw` expression.

When the compiler creates a temporary object to initialize a reference variable, the name of the temporary object has the same scope as that of the reference variable. When it creates a temporary object during the evaluation of an expression, the object exists until there is a break in the control flow of the program. If the compiler creates a temporary object for a class with constructors, it calls the appropriate (matching) constructor to create the temporary object.

When it destroys a temporary object and a destructor is available, the compiler calls the destructor to destroy the temporary object. When you exit from the scope in which the temporary object was created, the object is destroyed. If a reference is

## Temporary Objects

bound to a temporary object, the temporary object is destroyed when the reference passes out of scope unless it was destroyed earlier by a break in the flow of control. For example, a temporary object that is created by a constructor initializer for a reference member is destroyed on leaving the constructor.

The following example shows two expressions in which temporary objects are constructed:

```
class Y
{
public:
    Y(int)={ };
    Y(Y&){ };
    ~Y()={ };
};
Y add(Y y) { return y; }
//      .
//      .
//      .
void main ()
{
    Y obj1(10);
    Y obj2 = add(Y(5));    // one temporary created
    obj1 = add(obj1);      // two temporaries created
}
```

The above example created a temporary object of class type `Y` to construct `Y(5)` before passing it to the function `add()`. Because `obj2` is being constructed, the function `add()` can construct its return value directly into `obj2`, so another temporary object is not created. OS/390 C++ creates a temporary object of class type `Y` when the example passes `obj1` to the function `add()`. Because `obj1` has already been constructed, the function `add()` constructs its return value into a temporary object. The example then assigns this second temporary object to `obj1` by using an assignment operator.

## Related Information

- “Initializing References” on page 130
- “Chapter 6. Expressions and Operators” on page 133
- “Standard Type Conversions” on page 167
- “Chapter 8. Functions” on page 173
- “User-Defined Conversions”

---

## User-Defined Conversions

*User-defined conversions* allow you to specify object conversions with constructors or with conversion functions. OS/390 C++ implicitly uses user-defined conversions in addition to standard conversions for conversion of initializers, functions arguments, function return values, expression operands, expressions controlling iteration, selection statements, and explicit type conversions.

There are two types of user-defined conversions:

- Conversion by constructor
- Conversion functions.

For more information, see “Standard Type Conversions” on page 167.

## Conversion by Constructor

You can call a class constructor with a single argument to convert from the argument type to the type of the class.

For example:

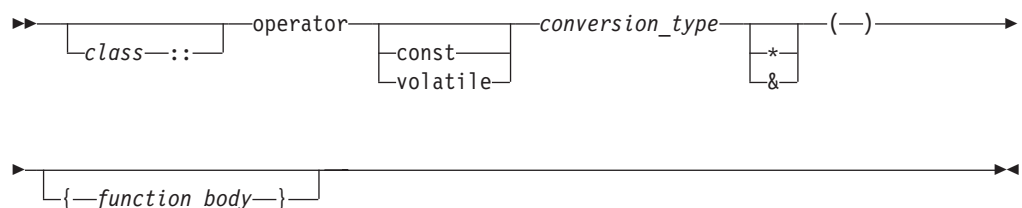
```
class Y
{
    int a,b;
    char* name;
public:
    Y(int i);
    Y(const char* n, int j = 0);
};

void add(Y);
//      .
//      .
//      .
void main ()
{
    // code                      equivalent code
    Y obj1 = 2;                  // obj1 = Y(2)
    Y obj2 = "somestring";      // obj2 = Y("somestring",0)
    obj1 = 10;                  // obj1 = Y(10)
    add(5);                     // add(Y(5))
}
```

OS/390 C++ applies, at most, one user-defined conversion, either a constructor or conversion function, to a class object. Assume you call a constructor with an argument, and you have not defined a constructor that accepts that argument type. OS/390 C++ only uses standard conversions to convert the argument to another argument type that is acceptable to a constructor for that class. It does not call other constructors or conversion functions to convert the argument to a type that is acceptable to a constructor that is defined for that class.

## Conversion Functions

You can define a member function of a class that is called a *conversion function*. A conversion function converts from the type of its class to another specified type.



The conversion function specifies a conversion from the class type of which the conversion function is a member, to the type specified by the name of the conversion function. Classes, enumerations, and typedef names cannot be declared or defined as part of the function name.

The following code fragment shows a conversion function called `operator int()`:

```
class Y
{
    int b;
```

## User-Defined Conversions

```
public:
    operator int();
};
Y::operator int() {return b;}
void f(Y obj )
{
    // each value assigned is converted by Y::operator int()
    int i = int(obj);
    int j = (int)obj;
    int k = i + obj;
}
```

Conversion functions have no arguments, and the return type is implicitly the conversion type. Conversion functions can be inherited. You can have virtual conversion functions, but not static ones.

OS/390 C++ implicitly applies only one user-defined conversion to a single value. User-defined conversions must be unambiguous, or OS/390 C++ does not call them.

If you declare a conversion function with the keyword `const`, the keyword does not affect the function. Except, it does act as a tiebreaker when there is more than one conversion function that you could apply. Specifically, if more than one conversion function could be applied, OS/390 C++ compares all of these functions. If you declare any of these functions with the keyword `const`, OS/390 C++ ignores `const` for the purposes of this comparison. If one of these functions is a best match, it is applied. If there is no best match, OS/390 C++ compares the functions again, but this time it does not ignore `const`.

---

## Initialization by Constructor

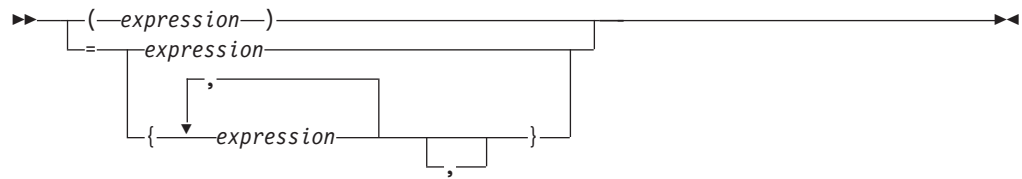
You must explicitly initialize a class object with a constructor or it must have a default constructor. Explicit initialization by using a constructor is the only way, except for aggregate initialization, to initialize nonstatic constant and reference class members.

An aggregate is a class object that has no constructors, no virtual functions, no private or protected members, and no base classes. “Structures” on page 106 and “Unions” on page 113 describes aggregates.

## Explicit Initialization

You can initialize class objects with constructors by using a parenthesized expression list. The call to a constructor uses this list as an argument list to initialize the class. You can also call a constructor with a single initialization value by using the `=` operator. Because this type of expression is an initialization, not an assignment, OS/390 C++ does not call the assignment operator function if one exists. It uses this value as a single argument for the call of a constructor. The type of the single argument must match the type of the first argument to the constructor. If the constructor has remaining arguments, these arguments must have default values.

The syntax for an initializer that explicitly initializes a class object with a constructor is:



The following example shows the declaration and use of several constructors that explicitly initialize class objects:

### CBC3X13A

```
// This example illustrates explicit initialization
// by constructor.

#include <iostream.h>
class complx
{
    double re, im;
public:
    complx(); // default constructor
    complx(const complx& c) {re = c.re; im = c.im;}
    // copy constructor
    complx( double r, double i = 0.0) {re = r; im = i;}
    // constructor with default trailing argument
    void display()
    {
        cout << "re = " << re << " im = " << im << endl;
    }
};
// .
// .
// .
void main ()
{
    complx one(1); // initialize with complx(double, double)
    complx two = one; // initialize with a copy of one
    // using complx::complx(const complx&)
    complx three = complx(3,4); // construct complx(3,4)
    // directly into three
    complx four; // initialize with default constructor
    complx five = 5; // complx(double, double) & construct
    // directly into five

    one.display();
    two.display();
    three.display();
    four.display();
    five.display();
}
```

The previous example produces the following output:

```
re = 1 im = 0
re = 1 im = 0
re = 3 im = 4
re = 0 im = 0
re = 5 im = 0
```

Constructors can initialize their members in two ways. A constructor can use the arguments you pass to it to initialize member variables in the constructor definition:

```
complx( double r, double i = 0.0) {re = r; im = i;}
```

## Initialization by Constructor

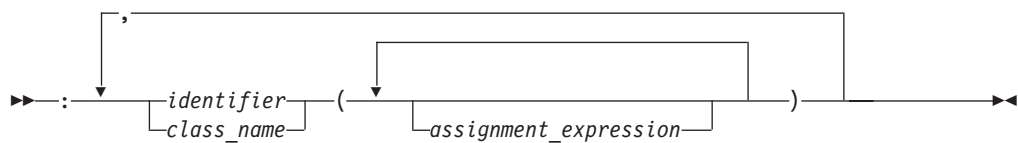
A constructor can have an initializer list within the definition but prior to the function body:

```
complex ( double r, double i = 0) : re(r), im(i) { /* ... */ }
```

Both methods assign the argument values to the appropriate data members of the class. You must use the second method to initialize base classes from within a derived class to initialize constant and reference members, and members with constructors.

## Initializing Base Classes and Members

You can initialize immediate base classes and derived class members that are not inherited from base classes. To do this, specify initializers in the constructor definition prior to the function body. The syntax for a constructor initializer is:



In a constructor that is not inline, include the initialization list as part of the function definition, not as part of the class declaration.

For example:

```
class B1
{
    int b;
public:
    B1();
    B1(int i) : b(i) { /* ... */ }    // inline constructor
};
class B2
{
    int b;
protected:
    B2();
    B2(int i);                        // noinline constructor
};
// B2 constructor definition including initialization list
B2::B2(int i) : b(i) { /* ...*/ }
//      .
//      .
//      .
class D : public B1, public B2
{
    int d1, d2;
public:
    D(int i, int j) : B1(i+1), B2(), d1(i) {d2 = j;}
};
```

If you do not explicitly initialize a base class or member that has constructors by calling a constructor, the compiler automatically initializes the base class or member with a default constructor. Consider in the above example, if you leave out the call `B2()` in the constructor of class `D` (as shown below). In that case, a constructor initializer with an empty expression list is automatically created to initialize `B2`. The constructors for class `D`, that is shown above and below, result in the same construction of a class `D` object.



```

class D : public B1, public B2
{
    int d1, d2;
public:
    // call B2() generated by compiler
    D(int i, int j) : B1(i+1), d1(i) {d2 = j;}
};

```

**Note:** You must declare base constructors with the access specifiers `public` or `protected` to enable a derived class to call them.

For example:

```

class B1
{
    int b;
public:
    B1();
    B1(int i) : b(i) { /* ... */ }
};
class B2
{
    int b;
protected:
    B2();
    B2(int i);
};
B2::B2(int i) : b(i) { /* ... */ }
class B4
{
public:
    B4();           // public constructor for B4
    int b;
private:
    B4(int);        // private constructor for B4
};
//      .
//      .
//      .

class D : public B1, public B2, public B4
{
    int d1, d2;
public:
    D(int i, int j) : B1(i+1), B2(i+2) ,
                    B4(i) {d1 = i; d2 = j; }
                    // error, attempt to access private constructor B4()
};

```

To ensure a valid call, you can change the code as follows:

```

public:
    D(int i, int j) : B1(i+1), B2(i+2) {d1 = i; d2 = j;}
                    // valid, calls public constructor for B4

```

## Construction Order of Derived Class Objects

When you create a derived class object using constructors, OS/390 C++ creates it in the following order:

1. It initializes the virtual base classes in the order they appear in the base list.
2. It initializes nonvirtual base classes in the order of declaration.
3. It initializes class members in the order of declaration (regardless of their order in the initialization list).

## Initialization by Constructor

4. It executes the body of the constructor.

The following code fragment calls the constructor for class B1, before it initializes the member d1. The value it passes to the constructor for class B1 is undefined.

```
class B1
{
    int b;
public:
    B1();
    B1(int i) {b = i;}
};
//      .
//      .
//      .
class D : public B1
{
    int d1, d2;
public:
    D(int i, int j) : d1(i), B1(d1) {d2 = j;}
    // d1 is not initialized in call B1::B1(d1)
};
```

---

## Copying Class Objects

You can copy one class object to another object of the same type by either assignment or initialization.

Copy by assignment is implemented with an assignment operator function. If you do not define the assignment operator, the compiler defines it as a *memberwise assignment*.

Copy by initialization is implemented with a copy constructor. If you do not define a copy constructor, OS/390 C++ defines it as a *memberwise initialization* of its class members.

Memberwise assignment and memberwise initialization mean the following. If a class has a member that is a class object, the assignment operator and copy constructor of that class object are used to implement assignment and initialization of the member.

## Copy Restrictions

You cannot generate a default assignment operator for a class that has the following:

- A nonstatic constant or a reference data member
- A nonstatic data member or base class whose assignment operator is not accessible
- A nonstatic data member or base class with no assignment operator and for which a default assignment operator cannot be generated

You cannot generate a default copy constructor for a class that has:

- A nonstatic data member or base class, whose copy constructor is not accessible
- A nonstatic data member or base class with no copy constructor and for which a default copy constructor cannot be generated

## Copy by Assignment

If you do not define an assignment operator and your code requires one, the compiler defines a default assignment operator. If you do not define an assignment operator and your code does not require one, the compiler declares a default assignment operator, but does not define it. If an assignment operator that takes a single argument of a class type exists for a class, the compiler does not generate a default assignment operator.

You can use copy by assignment only in an assignment expression.

You can define an assignment operator for a class with a single argument that is a constant reference to that class type. However, all its base classes and members must have assignment operators that accept constant arguments.

For example:

```
class B1
{
public:
    B1& operator=(const B1&);
};
class D: public B1
{
public:
    D& operator=(const D&);
};
D& D::operator=(const D& dobj) {D dobj2 = dobj;
                                return dobj2;}
```

Otherwise, you can define an assignment operator for a class with a single argument that is a reference to that class type. For example:

```
class Z
{
public:
    Z& operator=( Z&);
};
Z& Z::operator=(Z& zobj) {Z zobj2 = zobj;
                           return zobj2;}
```

The default assignment operator for a class is a public class member. The return type is a reference to which the class type it is a member.

For more information on standard C and C++ assignment operators, see “Assignment Expressions” on page 162. For more information on assignment operator functions, see “Overloaded Assignment” on page 319.

## Copy by Initialization

If you do not define a copy constructor and your program requires one, the compiler creates a default copy constructor. If you do not define a copy constructor, and your program does not require one, the compiler declares a default copy constructor but does not define it. If a class defines a copy constructor, the compiler does not generate a default copy constructor.

Use copy by initialization only in initialization.

## Copying Class Objects

You can define a copy constructor for a class with a single argument that is a constant reference to a class type. However, all of its base classes and members must have copy constructors that accept constant arguments, for example:

```
class B1
{
public:
    B1(const B1&) { /* ... */ }
};

class D: public B1
{
public:
    D(const D&);
};

D::D(const D& dobj):B1(dobj) { /* ... */ }
```

Otherwise, you can define a copy constructor with a single reference to a class type argument. For example:

```
class Z
{
public:
    Z(Z&);
};

Z::Z(Z&) { /* ...*/ }
```

The default copy constructor for a class is a public class member. For more information on copy constructors, see “Constructors” on page 326, and “Initialization by Constructor” on page 336.

---

## Chapter 15. C++ Inheritance

In C++, you can create classes from existing classes by using the object-oriented programming technique that is called *inheritance*. Inheritance allows you to define an *is a* relationship between classes. When members are inherited, you can use them as if they are members of the class that inherits them. This chapter discusses the following topics on inheritance:

- “Inheritance Overview”
- “Derivation” on page 346
- “Inherited Member Access” on page 349
- “Multiple Inheritance” on page 356
- “Virtual Functions” on page 359
- “Abstract Classes” on page 363

### Related Information

- “Chapter 8. Functions” on page 173
- “Chapter 11. C++ Classes” on page 281
- “Chapter 12. C++ Class Members and Friends” on page 291

---

## Inheritance Overview

C++ implements inheritance through the mechanism of *derivation*. Derivation allows you to reuse code by creating new classes, called *derived classes*, that inherit properties from one or more existing classes, called *base classes*. A derived class inherits the properties, that includes data and function members, of its base class. You can also add new data members and member functions to the derived class. You can modify the implementation of existing member functions or data by overriding base class member functions or data in the newly derived class.

Suppose that you have defined a shape class to describe and operate on geometric shapes. Now suppose that you want to define a circle class. Because you have existing code that operates on the shape class, you can use inheritance to create the circle class. You can redefine operations in the derived circle class that were originally defined in the shape base class. When you manipulate an object of the circle class, OS/390 C++ uses these redefined function implementations.

## Inheritance

For example:

```
class shape
{
    char* name;
    int xpoint, ypoint;
public:
    virtual void rotate(int);
    virtual void draw();
    void display() const;
};

class circle: public shape    // derive class circle from class shape
{
    int xorigin, yorigin;
    int radius;
public:
    void rotate(int);
    void draw();
    void display() const;
};
//      .
//      .
//      .
```

In the above example, class `circle` inherits the data members `name`, `xpoint`, and `ypoint`, as well as the member functions `display()`, `rotate()`, and `draw()` from class `shape`. The example declares member functions `rotate()` and `draw()` in class `shape` with the keyword `virtual`. Consequently, you can provide an alternative implementation for the member functions in class `circle`.

You can also provide an alternative implementation for the nonvirtual member function `display()` in class `circle`. Suppose you manipulate an argument of type `circle` using a pointer to `shape`, and you call a virtual member function. The member function defined in the derived class overrides the base-class member function. A similar call to a nonvirtual member function calls the member function that is defined in the base class. In addition to inheriting the members of class `shape`, class `circle` has declared its own data members, `xorigin`, `yorigin`, and `radius`.

The key difference between virtual and nonvirtual member functions is as follows. When you treat the `circle` class as if it were a `shape`, the program uses the implementations of the virtual functions `rotate()` and `draw()` that are defined in class `circle` rather than those originally defined in class `shape`. Because `display()` is a nonvirtual member function, the original implementation of `display()` defined in class `shape` is used.

## Multiple Inheritance

*Multiple inheritance* allows you to create a derived class that inherits properties from more than one base class.

For example, in addition to the above `shape` class, you could also have a `symbol` class. Because a `circle` is both a `shape` and a `symbol`, you can use multiple inheritance to reflect this relationship. If your program derives the `circle` class from both the `shape` and `symbol` classes, the `circle` class inherits properties from both classes.

```

class symbol
{
    char* language;
    char letter;
    int number;
public:
    virtual void write();
    virtual void meaning();
};
class shape
{
    char* name;
    int xpoint, ypoint;
public:
    virtual void rotate(int);
    virtual void draw();
    void display() const;
};
class circle: public symbol, public shape
{
    int xorigin, yorigin;
    int radius;
public:
    void rotate(int);
    void draw ();
    void write();
    void meaning();
    void display() const;
};
//      .
//      .
//      .

```

In the previous example, class `circle` inherits the members `name`, `xpoint`, `ypoint`, `display()`, `rotate()`, and `draw()` from class `shape`. It also inherits the members `language`, `letter`, `number`, `write()`, and `meaning()` from class `symbol`.

Because a derived class inherits members from all its base classes, ambiguities can result. For example, if two base classes have a member with the same name, the derived class cannot implicitly differentiate between the two members. Note that, when you are using multiple inheritance, the access to names of base classes may be ambiguous.

## The Inheritance Design Process

Multiple inheritance allows you to have more than one base class for a single derived class. You can create an interconnected *inheritance graph* of inherited classes by using derived classes as base classes for other derived classes. You can build an inheritance graph through the process of specialization in which derived classes are more specialized than their base classes. You can also work in the reverse direction and build an inheritance graph through generalization. If you have a number of related classes that share a group of properties, you can generalize and build a base class to embody them. The group of related classes becomes the derived classes of the new base class.

## Direct and Indirect Base Classes

A *direct base class* is a base class that appears directly as a base specifier in the declaration of its derived class. A direct base class is analogous to a parent in a hierarchical graph. In the above example, both `shape`, and `symbol`, are direct base classes of class `circle`.

## Inheritance

An *indirect base class* is a base class that does not appear directly in the declaration of the derived class but is available to the derived class through one of its base classes. An indirect base class is analogous to a grandparent or great grandparent or great-great grandparent in a hierarchical graph. For a given class, all base classes that are not direct base classes are indirect base classes.

## Polymorphism

Polymorphic functions are functions that you can apply to objects of more than one type. C++ implements polymorphic functions in two ways:

- Overloaded functions are statically bound at compile time, as discussed in “Overloading Functions” on page 311.
- C++ provides virtual functions. A *virtual function* is a function that you can call for a number of different user-defined types that are related through derivation. Virtual functions are bound dynamically at run time.

Typically, a base class has several derived classes, each requiring its own customized version of a particular operation. It is difficult for a base class to implement member functions that are useful for all of its derived classes. A base class would have to determine which derived class an object belonged to before it could execute the applicable code for that object. When you call a virtual function, the compiler executes the function implementation that is associated with the object for which you call the function. The implementation of the base class is only a default that is used when the derived class does not contain its own implementation.

---

## Derivation

C++ implements inheritance through the mechanism of derivation. Derivation allows you to derive a class, called a *derived class*, from another class, called a *base class*.

In the declaration of a derived class, you list the base classes of the derived class. The derived class inherits its members from these base classes. All classes that appear in the list of base classes must be previously defined classes.

Base lists do not allow incompletely declared classes.

For example:

```
class X; // incomplete declaration of class X
class Y: public X    // error
{
//      .
//      .
//      .
};
```

When you derive a class, the derived class inherits class members of the base class. You can refer to inherited members (base class members) as if they were members of the derived class.

Consider the following example.



**CBC3X14A**

```
// This example illustrates references
// to base class members.

class base
{
public:
    int a,b;
};
class derived : public base
{
public:
    int c;
};
void main()
{
    derived d;
    d.a = 1;      // base::a
    d.b = 2;      // base::b
    d.c = 3;      // derived::c}
```

The derived class can also add new class members and redefine existing base class members. In the above example, the two inherited members, a and b, of the derived class d, in addition to the derived class member c, are assigned values. If you redefine base class members in the derived class, you can still refer to the base class members by using the :: (scope resolution) operator.

Consider the following example.

**CBC3X14B**

```
// This example illustrates references to base class
// members with the scope resolution (::) operator.
#include <iostream.h>
class base
{
public:
    char* name;
    void display(char* i) {cout << i << endl;}
};
class derived : public base
{
public:
    char* name;
    void display(char* i){cout << i << endl;}
};
void main()
{
    derived d;                // create derived class object
    d.name = "Derived Class";  // assignment to derived::name
    d.base::name = "Base Class"; // assignment to base::name

    // call derived::display(derived::name)
    d.display(d.name);

    // call base::display(base::name)
    d.base::display(d.base::name);
}
```

“C++ Scope Resolution Operator (::)” on page 137 describes the :: (scope resolution) operator.

## Derivation

You can manipulate a derived class object as if it was a base class object. You can use a pointer or a reference to a derived class object in place of a pointer or reference to its base class. For example, you can pass a pointer or reference to a derived class object `D` to a function that expects a pointer or reference to the base class of `D`. You do not need to use an explicit cast to achieve this; the compiler performs a standard conversion. You can implicitly convert a pointer to a derived class to point to a base class. You can also implicitly convert a reference to a derived class to a reference to a base class.

The following example assigns `d`, a pointer to a derived class object, to `bptr`, a pointer to a base class object. A call is made to `display()` using `bptr`. Even though `bptr` has a type of pointer to base, in the body of `display()` the name member of derived is manipulated.

### CBC3X14C

```
// This example illustrates how to make a pointer
// to a derived class point to a base class.

#include <iostream.h>
class base
{
public:
    char* name;
    void display(char* i) {cout << i << endl;}
};
class derived : public base
{
public:
    char* name;
    void display(char* i){cout << i << endl;}
};
void main()
{
    derived d;

    // standard conversion from derived* to base*
    base* bptr = &d;

    // call base::display(base::name;)
    bptr->display(bptr->name);
}
```

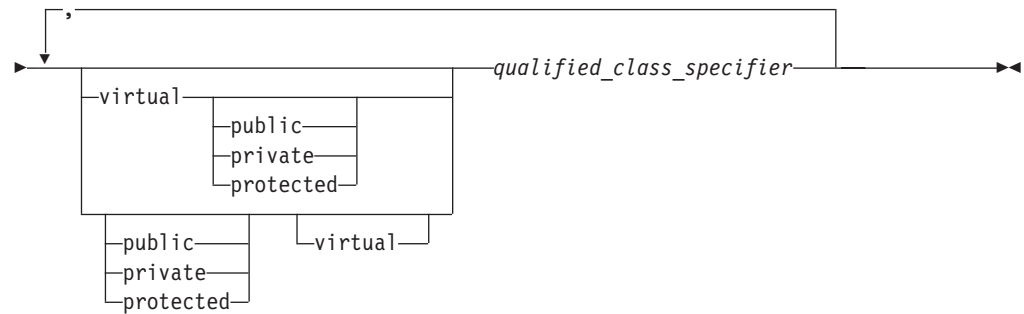
You cannot implicitly convert a pointer or a reference to a base class object, to a pointer or reference to a derived class.

If a member of a derived class and a member of a base class have the same name, the base class member is hidden in the derived class. If a member of a derived class has the same name as a base class, the program hides the base class name in the derived class. In both cases, refer to the name of the derived class member as the *dominant* name.

## Syntax of a Derived Class Declaration

The syntax for the list of base classes is:

►►—*derived\_class*—:—————►



The *qualified class specifier* must be a class that you have previously declared in a class declaration, as “Class Names” on page 283 describes. The access specifiers (public, private, and protected) are described in “Member Access” on page 304.

You can use the virtual keyword to declare virtual base classes. For more information, see “Virtual Base Classes” on page 357.

The following example shows the declaration of the derived class D and the base classes V, B1, and B2. The class B1 is derived from class V and is a base class for D. Consequently, it is both a base class and a derived class.

```
class V { /* ... */ };
class B1 : virtual public V { /* ... */ };
class B2 { /* ... */ };
class D : public B1, private B2 { /* ... */ };
```

## Inherited Member Access

Access specifiers, as described in “Access Specifiers” on page 305, control the level of access to noninherited class members. The access for an inherited member is controlled in three ways:

- When you declare a member in a base class, you can specify a level of access using the keywords public, private, and protected.
- When you derive a class, you can specify the access level for the base class in the base list.
- You can also restore the access level of inherited members. See “Derivation Access of Base Classes” on page 350 for an example.

Resolution of member names does not depend on the level of access that is associated with each class member. Consider the following example:

```
class A {
    private:
        int a;
};
class B {
    public:
        int a;
};
class C : public A, public B {
    void f() { a = 0; } // ambiguous - is it A::a or B::a?
};
```

In this example, class A has a private member a, and class B has a public member a. The example derives class C from both A and B. C does not have access to A::a,

## Inherited Member Access

but `a` in the body of `f()` or `B::` can still resolve to either `A::a` or `B::a`. For this reason, `a` is ambiguous in the body of `f()`.

## Protected Members

If you publicly derive a class from a base class, members and friends of any classes that are derived from that class can access a protected static base class member. Members and friends of any classes derived from that base class can access a protected nonstatic base class member by using one of the following items:

- A pointer to a directly or indirectly derived class
- A reference to a directly or indirectly derived class
- An object of a directly or indirectly derived class

If you derive a class privately from a base class, all protected base class members become private members of the derived class.

The access specifier `protected` is also described in “Access Specifiers” on page 305.

## Derivation Access of Base Classes

When you declare a derived class, an access specifier can precede each base class in the base list of the derived class. This does not alter the access attributes of the individual members of a base class as seen by the base class. Instead, it but allows the derived class to restore the access attributes of the members of a base class.

You can derive classes by using any of the following three access specifiers:

- In a `public` base class, `public` and `protected` members of the base class remain `public` and `protected` members of the derived class.
- In a `private` base class, `public` and `protected` members of the base class become `private` members of the derived class.
- In a `protected` base class, `public` and `protected` members of the base class are `protected` members of the derived class.

In all cases, `private` members of the base class remain `private`. The derived class cannot use `private` members of the base class unless friend declarations within the base class explicitly grant access to them.

The following example derives class `d` publicly from class `b`:

```
class b
{
//      .
//      .
//      .
};
class d : public b // public derivation
{
//      .
//      .
//      .
};
```

You can use both a structure and a class as base classes in the base list of a derived class declaration. If you declare the base class with the keyword `class`, its default access specifier in the base list of a derived class is `private`. If you declare the base class with the keyword `struct`, its default access specifier in the base list of a derived class is `public`.

The following example uses private derivation by default because it does not use an access specifier in the base list:

```
struct bb
{
    //      .
    //      .
    //      .
};
class dd : bb // private derivation
{
    //      .
    //      .
    //      .
};
```

Members and friends of a class can implicitly convert a pointer to an object of that class to a pointer to either:

- A direct private base class
- A protected base class (either direct or indirect)

## Access Declarations

You can restore access to members of a base class by using an *access declaration*. It allows you to change the access to a public member in a private or protected base class back to public. You can also change the access to a protected member in a private base class back to protected. Use the base class member qualified name in the public or protected declarations of the derived class to adjust access.

You only use access declarations to restore base class access. You cannot use them to do the following tasks:

- Give more access to a member than originally declared
- Change the access of a private member to public or to protected
- Change the access of a protected member to public
- Restrict access to a member that is accessible in a base class

Using an access declaration to change the access to a public member of a public base class to public is redundant. Using an access declaration to change the access to a protected member of a protected base class to protected is also redundant.

The following example declares the member `b` of the base class `base` as public in its base class declaration. The example derives the class, `derived`, privately from class `base`. The access declaration in the public section of the class, `derived`, restores the access level of the member `b` back to public.

## Inherited Member Access

### CBC3X14D

// This example illustrates using access declarations  
// to restore base class access.

```
#include <iostream.h>
class base
{
    char a;
public:
    char c, b;
    void bprint();
};

class derived: private base
{
    char d;
public:
    char e;
    base::b; // restore access to b in derived
    void dprint();
    derived(char ch) { base::b = ch; }
};

void print(derived& d)
{
    cout << " Here is d " << d.b << endl;
}

void main()
{
    derived obj('c');
    print(obj);
}
```

The external function `print(derived&)` can use the member `b` of base because OS/390 C++ restores the access of `b` to public. The external function `print(derived&)` can also use the members `e` and `dprint()` because they are declared with the keyword `public` in the derived class. The derived class member `dprint()` can use the members of its own class, `d` and `e`. This is in addition to the inherited members, `b`, `c`, and `bprint()` that the example declares with the keyword `public` in the base class. The base class member `bprint()` can use all the members of its own class, `a`, `b`, and `c`.

Use access declarations only to adjust the access to a member in a base class. The derived class can directly or indirectly inherit the base class in which an access declaration appears.

You can also use an access declaration in a nested class. For example:

```
class B
{
public:
    class N          // nested class
    {
    public:
        int i;      // public member
    };
};

class D: private B::N // derive privately
{
public:
    B::N::i; // restores access to public
};
```

You cannot adjust the access to a base class member if a member with the same name exists in a class that is derived from that base class.

You cannot convert a pointer to a derived class object to a pointer to a base class object if the base class is private or protected. For example:

```
class B { /* ... */ };
class D : private B { /* ... */ };    // private base class

void main ()
{
    D d;
    B* ptr;
    ptr = &d;    // error
}
```

If you use an access declaration to adjust the access to an overloaded function, the function adjusts the access for all functions with that name in the base class.

## Access Resolution

Access resolution is the process by which the accessibility of a particular class member is determined. Accessibility is dependent on the context. For example, a class member can be accessible in a member function but inaccessible at file scope. The following describes the access resolution procedure that is used by the compiler.

In general, two *scopes* must be established before access resolution is applied. These scopes reduce an expression or declaration into a simplified construct to which the access rules are applied. “Member Access” on page 304 describes access rules. These scopes are:

<b>Call scope</b>	The scope that encloses the expression or declaration that uses the class member.
<b>Reference scope</b>	The scope that identifies the class.

For example, in the following code the reference scope for member is the type of aobject, that is class type A:

### CBC3X14E

// This example illustrates access resolution.

```
class B { public: int member; };    // declaration
class A : B {}                    // declaration
void main()
{
    A aobject;                    // declaration
    aobject.member = 10;          // expression
}
```

Choose reference scope by simplifying the expression (or declaration) that contains the member. An expression can be thought of as being reduced to a simple expression of the form `obj.member` where `obj` is the reference scope. Select reference scope as follows:

1. Consider if the member is qualified with `.` (dot) or `->` (arrow). If it is, the reference scope is the type of the object that is immediately to the left of the `.` or `->` operator closest to the member. OS/390 C++ treats unqualified members as if they are qualified with `this->`.

## Inherited Member Access

2. Consider if the member is a type member or a static member and is qualified with `::` (the scope resolution operator). Then, the reference scope is the type immediately to the left of the `::` operator closest to the member.
3. Otherwise, the reference scope is the call scope.

The call scope and the reference scope determine the accessibility of a class member. Once these scopes are resolved, the *effective access* of the member is determined. Effective access is the access of the member as you see it from the reference scope. You can determine effective access. Take the original access of the member in its scope as the effective access, and change it as you traverse the class hierarchy from the member's class to the reference scope. Traversing the class hierarchy for each derivation by the following, alters effective access:

- The derivation access of a base class (see "Derivation Access of Base Classes" on page 350)
- Access declarations that are applied to the members (see "Access Declarations" on page 351)
- Friendships that are granted to the call scope (see "Member Access" on page 304)

After effective access is determined for a member, the access rules are applied as if the effective access was the original access of the member. A member is only accessible if the access rules say that it is.

The following example demonstrates the access resolution procedure.

```
class A
{
public:
    int a;
};
class B : private A
{
    friend void f (B*);
};
void f(B* b)
{
    b->a = 10; // is 'a' accessible to f(B*) ?
}
//      .
//      .
//      .
```

The following steps occur to determine the accessibility of `A::a` in `f(B*)`:

1. The call scope and reference scope of the expression `b->a` are determined:
  - a. The call scope is the function `f(B*)`.
  - b. The reference scope is class B.
2. The effective access of member `a` is determined:
  - a. Because the original access of the member `a` is public in class A, the initial effective access of `a` is public.
  - b. Because B inherits from A privately, the effective access of `a` inside class B is private.
  - c. Because class B is the reference scope, the effective access procedure stops here. The effective access of `a` is private.
3. The access rules are applied. The rules state that a friend or a member of the member's class can access a private member. Because `f(B*)` is a friend of class B, `f(B*)` can access the private member `a`.



## Access Summary

The following example demonstrates inherited member access rules.

### CBC3X14F

// This example illustrates inherited member access rules.

```

class B
{
    int a;
public:
    int b,c;
    void f(int) {}
protected:
    int d;
    void g(int) {}
};

class D1 : public B
{
    int a;
public:
    int b;
    void h(int i )
    {
        g(i);           // valid, protected B::g(int)
        B::b = 10;       // valid, B::b (not local b)
        d = 5 ;         // valid, protected B::d
    }
};

class D2 : private B
{
    int e;
public:
    B::c;               // modify access to B::c
    void h(int i) { d = 5; } // valid,protected B::d
};

void main( )
{
    int i= 1;           // declare and initialize local variable
    D1 d1;              // create object of class d1
    D2 d2;              // create object of class d2

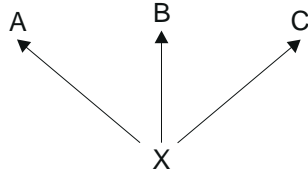
    d1.a = 5;           // error, D1::a is private in class D1
    d2.b = 10;          // error, B::b is inherited private to
                        // derived class D2
    d2.c = 5;           // valid, modified access from private to public
    d2.B::c = 5;        // valid, public B::c
    d1.c = 5;           // valid, B::c is inherited publicly
    d1.d = 5;           // error, B::d is protected in base class
    d2.e = 10;          // error, private D2::e
    d1.g(i);            // error, g(int) is protected in base class
    d1.h(i);            // valid
    d2.h(i);            // valid
}

```

## Multiple Inheritance

You can derive a class from more than one base class. *Multiple inheritance* means to derive a class from more than one direct base class.

In the following example, classes A, B, and C are direct base classes for the derived class X:



```

class A { /* ... */ };
class B { /* ... */ };
class C { /* ... */ };
class X : public A, private B, public C { /* ... */ };
  
```

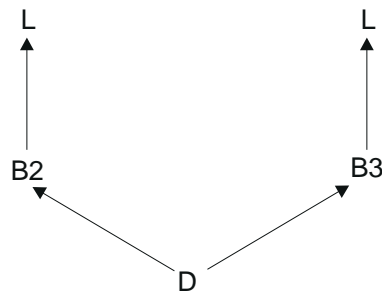
The order of derivation is relevant only to determine the order of default initialization by constructors and cleanup by destructors. For more information, see “Initialization by Constructor” on page 336.

A direct base class cannot appear in the base list of a derived class more than once:

```

class B1 { /* ... */ };           // direct
base class
class D : public B1, private B1 { /* ... */ }; // error
  
```

However, a derived class can inherit an indirect base class more than once, as shown in the following example:



```

class L { /* ... */ };           // indirect base class
class B2 : public L { /* ... */ };
class B3 : public L { /* ... */ };
class D : public B2, public B3 { /* ... */ }; // valid
  
```

In the above example, class D inherits the indirect base class L once through class B2 and once through class B3. However, this may lead to ambiguities because two objects of class L exist, and both are accessible through class D. You can avoid this ambiguity by referring to class L by using a qualified class name, for example, B2::L or B3::L.

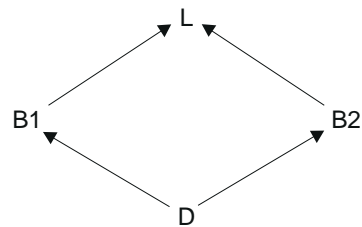
You can also avoid this ambiguity by using the base specifier `virtual` to declare a base class.

## Virtual Base Classes

If you have an inheritance graph in which two or more derived classes have a common base class, you can use a virtual base class to ensure that the two classes share a single instance of the base class.

In the following example, an object of class D has two distinct objects of class L, one through class B1, and another through class B2. You can use the keyword `virtual` in front of the base class specifiers in the *base lists* of classes B1 and B2. This indicates that only one class L, shared by class B1 and class B2, exists.

For example:

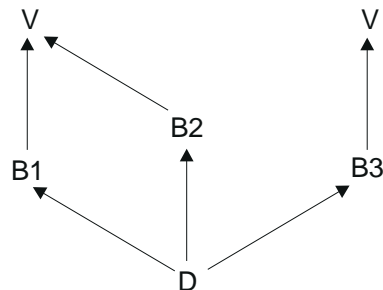


```

class L { /* ... */ }; // indirect base class
class B1 : virtual public L { /* ... */ };
class B2 : virtual public L { /* ... */ };
class D : public B1, public B2 { /* ... */ }; // valid
  
```

Using the keyword `virtual` in this example ensures that an object of class D inherits only one object of class L.

A derived class can have both virtual and nonvirtual base classes. For example:



```

class V { /* ... */ };
class B1 : virtual public V { /* ... */ };
class B2 : virtual public V { /* ... */ };
class B3 : public V { /* ... */ };
class D : public B1, public B2, public B3 { /* ... */ };
  
```

In the above example, class D has two objects of class V. One is shared by classes B1 and B2, and one is shared through class B3.

## Multiple Access

If an inheritance graph contains virtual base classes, you can access a name that can be reached through more than one path, through the path that gives the most access.

## Multiple Inheritance

For example:

```
class L { public: void f(); };
class B1 : private virtual L { /* ... */ };
class B2 : public virtual L { /* ... */ };
class D : public B1, public B2
{
public:
    void f() {L::f();} // L::f() is accessed through B2 and is public
};
```

In the above example, the function `f()` is accessed through class `B2`. Because class `B2` is inherited publicly and class `B1` is inherited privately, class `B2` offers more access.

## Accessible Base Classes

An *accessible base class* is a publicly derived base class that is neither hidden nor ambiguous in the inheritance hierarchy.

## Ambiguous Base Classes

When you derive classes, ambiguities can result if base and derived classes have members with the same names. Access to a base class member is ambiguous if you use a name or qualified name that does not refer to a unique function, object, type, or enumerator. The declaration of a member with an ambiguous name in a derived class is not an error. OS/390 C++ flags the ambiguity as an error if you use the ambiguous member name.

For example, if two base classes have a member of the same name, an attempt to access the member by the derived class is ambiguous. You can resolve ambiguity by qualifying a member along with its class name by using the `::` (scope resolution) operator.

### CBC3X14G

// This example illustrates ambiguous base classes.

```
class B1
{
public:
    int i;
    int j;
    int g( );
};
class B2
{
public:
    int j;
    int g( );
};
//      .
//      .
//      .
class D : public B1, public B2
{
public:
    int i;
};
```

```

void main ()
{
    D dobj;
    D *dptr = &dobj;
    dptr -> i = 5;           // valid, D::i
    dptr -> j = 10;          // error, ambiguous reference to j
    dptr->B1::j = 10;        // valid, B1::j
    dobj.g( );              // error, ambiguous reference to g( )
    dobj.B2::g( );          // valid, B2::g( )
}

```

The compiler checks for ambiguities at compile time. Because ambiguity checking occurs before access control or type checking, ambiguities may result even if only one of several members with the same name is accessible from the derived class.

Conversions (either implicit or explicit) from a derived class pointer or reference to a base class pointer or reference must refer unambiguously to the same accessible base class object. For example:

```

class W { /* ... */ };
class X : public W { /* ... */ };
class Y : public W { /* ... */ };
class Z : public X, public Y { /* ... */ };
void main ()
{
    Z z;
    X* xptr = &z;           // valid
    Y* yptr = &z;           // valid
    W* wptr = &z;           // error, ambiguous reference to class W
                             // X's W or Y's W ?
}

```

You can use virtual base classes to avoid ambiguous reference. For example:

```

class W { /* ... */ };
class X : public virtual W { /* ... */ };
class Y : public virtual W { /* ... */ };
class Z : public X, public Y { /* ... */ };
void main ()
{
    Z z;
    X* xptr = &z;           // valid
    Y* yptr = &z;           // valid
    W* wptr = &z;           // valid, W is virtual therefore only one
                             // W subobject exists
}

```

For more information, see “Virtual Base Classes” on page 357.

---

## Virtual Functions

In C++, the mechanism of virtual functions supports dynamic binding. Virtual functions must be members of a class. Use virtual functions when you expect a class to be used as a base class in a derivation and the derived class may override the function implementation. You can declare a member function with the keyword `virtual` in its class declaration.

## Virtual Functions

For example:

```
class B
{
    int a,b,c;
public:
    virtual int f();
};
//      .
//      .
//      .
```

You can re-implement a virtual member function, like any member function, in any derived class. When you make a call to a virtual function, the implementation that OS/390 C++ executes depends on the type of the object for which it is called. If you call a virtual member function for a derived class object and the function is redefined in the derived class, your program executes the definition in the derived class. In this case, the redefined derived class function *overrides* the base class function.

Overriding occurs even if the access to the function is through a pointer or reference to the base class. Calling a virtual function with a pointer that has base class type but points to a derived class object, calls the member function of the derived class. However, calling a nonvirtual function with a pointer that has base class type calls the member function of the base class, regardless of whether or not the pointer points to a derived class object.

For example:

```
class B
{
public:
    virtual int f();
    virtual int g();
    int h();
};
class D : public B
{
public:
    int f();
    int g(char*);    // hides B::g()
    int h();
};
//      .
//      .
//      .
void main ()
{
    D d;
    B* bptr = &d;

    bptr->f();        // calls D::f() because f() is virtual
    bptr->h();        // calls B::h() because h() is nonvirtual
    bptr->g();        // calls B::g()
    d.g();            // error, wrong number and type of arguments
    d.g("string");   // calls D::g(char*)
}
```

If the argument types or the number of arguments of the two functions are different, the functions are considered different. In addition, the function in the derived class does not override the function in the base class. The function in the derived class hides the function in the base class.

The return type of an overriding virtual function can differ from the return type of the overridden virtual function. However, the following restrictions apply:

- The return type of the overridden virtual function must be a pointer or a reference to a class B.
- The return type of the overriding virtual function must be a pointer or a reference to a class D.
- The return types of the overridden and overriding virtual functions must both be pointers to classes B and D respectively. Or, they must both be references to classes B and D respectively.
- Class B must be an accessible base class of class D. Also, with the OS/390 C++ compiler, class D must not have multiple or virtual inheritance.

For more information, see “Function Return Values” on page 192.

A virtual function cannot be global or static. By definition, a virtual function is a member function of a base class and relies on a specific object to determine which implementation of the function is called. You can declare a virtual function to be a friend of another class. “Friends” on page 306 describes friends.

If you declare a function as virtual in its base class, you can still access it directly by using the `::` (scope resolution) operator. In this case, OS/390 C++ suppresses the virtual function call mechanism, and uses the function implementation that is defined in the base class. If you do not redefine a virtual member function in a derived class, a call to that function uses the function implementation that is defined in the base class.

A virtual function must be one of the following:

- Defined
- Declared pure
- Defined and declared pure

A base class that contains one or more pure virtual member functions is an *abstract class*. For more information, see “Abstract Classes” on page 363.

## Ambiguous Virtual Function Calls

It is an error to override one virtual function with two or more ambiguous virtual functions. This can happen in a derived class that inherits from two nonvirtual bases that are derived from a virtual base class.

For example:

```
class V
{
public:
    virtual void f() { /* ... */ };
};
class A : virtual public V
{
    void f() { /* ... */ };
};
class B : virtual public V
{
    void f() { /* ... */ };
};
class D : public B, public A { /* ... */ }; // error
void main ()
{
```

## Virtual Functions

```
D d;
V* vptr = &d;
vptr->f();           // which f(), A::f() or B::f()?
}
```

In class A, only `A::f()` will override `V::f()`. Similarly, in class B, only `B::f()` will override `V::f()`. However, in class D, both `A::f()` and `B::f()` attempt to override `V::f()`. The compiler does not allow this attempt because it is not possible to decide which function to call if a D object is referenced with a pointer to class V. The above example illustrates this point. The compiler flags this situation as an error, as only one function can override a virtual function.

A special case occurs when the ambiguous overriding virtual functions come from separate instances of the same class type. In the following example, there are two objects (instances) of class L. There are two data members `L::count`, one in class A and one in class B. If the compiler allows the declaration of class D, incrementing `L::count` in a call to `L::f()` with a pointer to class V is ambiguous.

```
class V
{
public:
    virtual void f();
};
class L : virtual public V
{
    int count;
    void f();
};
void L::f() {++count;}
class A : public L
{ /* ... */ };
class B : public L
{ /* ... */ };
class D : public A, public B { /* ... */ }; // error
void main ()
{
    D d;
    V* vptr = &d;
    vptr->f();
}
```

In the above example, the function `L::f()` is expecting a pointer to an L object; that is, the this pointer for class L, as its first implicit argument. Because there are two objects of class L in a D object, there are two this pointers that could be passed to `L::f()`. Because the compiler cannot decide which this pointer to pass to `L::f()`, the declaration of class D is flagged as an error.

## Virtual Function Access

You specify the access for a virtual function when you declare it. The access rules for the function that later overrides the virtual function do not affect the access rules for a virtual function. In general, the access of the overriding member function is not known.

If you call a virtual function with a pointer or reference to a class object, the compiler does not use the type of the class object to determine the access of the virtual function. Instead, it uses the type of the pointer or reference to the class object.

Consider the following example. When you call the function `f()` using a pointer having type `B*`, OS/390 C++ uses `bptr` to determine the access to the function.



Although the definition of `f()`, which is defined in class `D` is executed, the access of the member function `f()` in class `B` is used. When the function `f()` is called using a pointer having type `D*`, `dptr` is used to determine the access to the function `f()`. This call produces an error because `f()` is declared private in class `D`.

```
class B
{
public:
    virtual void f();
};
class D : public B
{
private:
    void f();
};
//      .
//      .
//      .
void main ()
{
    D dobj;
    B *bptr = &dobj;
    D *dptr = &dobj;
    bptr->f();    // valid, virtual B::f() is public,
                // D::f() is called
    dptr->f();    // error, D::f() is private
}
```

---

## Abstract Classes

An *abstract class* is a class that is designed to be specifically used as a base class. An abstract class contains at least one *pure virtual function*. Pure virtual functions are inherited. You can declare a function to be pure by using a pure specifier in the declaration of the member function in the class declaration.

For example:

```
class AB          // abstract class
{
public:
    virtual void f()= 0; // pure virtual member function
};
class D: public AB
{
public:
    void f();
};
//      .
//      .
//      .
void main ()
{
    D d;
    d.f();    // calls D::f()
    AB ab;    // error, cannot create an object of an
              // abstract class type
}
```

A function that is declared pure typically has no definition and cannot be executed. Attempting to call a pure virtual function that has no implementation is undefined; however, such a call does not cause an error. You cannot create objects of an abstract class, as the above example demonstrates.

## Abstract Classes

**Note:** Because destructors are not inherited, a virtual destructor that is declared pure must have a definition.

Virtual member functions are inherited. Consider if a base class contains a pure virtual member function and a class derived from that base class does not redefine that pure virtual member function. In that case, the derived class itself is an abstract class. Any attempt to create an object of the derived class type produces an error.

For example:

```
class AB // abstract class
{
public:
    virtual void f()= 0; // pure virtual member function
};
class D2: public AB
{
    int a,b,c;
public:
    void g();
};
//      .
//      .
//      .
void main ()
{
    D2 d;
    // error, cannot declare an object of abstract class D2
}
```

To avoid the error in the above example, provide a declaration of `D2::f()`.

You cannot use an abstract class as the type of an explicit conversion, as an argument type, or as the return type for a function. You can declare a pointer or reference to an abstract class.

---

## Chapter 16. C++ Templates

This chapter describes the C++ template facility. A *template* specifies how you can construct an individual class, function, or static data member by providing a blueprint description of classes or functions within the template.

Unlike an ordinary class or function definition, a template definition contains the `template` keyword. It uses a *type argument*, instead of a type, in one or more of the constructs used to define the class or function template. You can then generate individual classes or functions simply by specifying the template name and by naming the type for the particular class or function as the type argument of the template. You can use templates to define a family of types or functions.

See the *OS/390 C/C++ Programming Guide* for programming hints on using templates in C++ programs.

This chapter describes the following topics:

- “Templates Overview”
- “Structuring Your Program Using Templates” on page 367
- “Class Templates” on page 369
- “Function Templates” on page 373
- “Differences between Class and Function Templates” on page 377
- “Member Function Templates” on page 377
- “Friends and Templates” on page 379
- “Static Data Members and Templates” on page 380

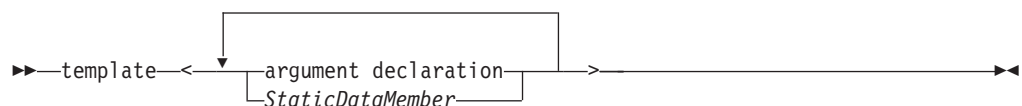
### Related Information

- “Chapter 8. Functions” on page 173
- “Chapter 11. C++ Classes” on page 281
- “Type Specifiers” on page 85
- “define (C++ Only)” on page 251
- “implementation (C++ Only)” on page 255

---

## Templates Overview

The syntax for a template is:



## Templates Overview

The *declaration* in a template declaration must define or declare one of the following:

- A class
- A function
- A static member of a template class

The *identifier* of a *type* is a *type\_name* in the scope of the template declaration. A template declaration can appear as a global declaration only.

The template arguments (within the < and > delimiters) specify the types and the constants within the template that you must specify when the template is instantiated.

Given the following template:

```
template<class L> class Key
{
    L k;
    L* kptr;
    int length;
public:
    Key(L);
    // ...
};
```

The following table shows what the classes Key<int>, Key<char\*>, and Key<mytype> look like:

class Key<int> i;	class Key<char*> c;	class Key<mytype> m;
class Key<int>{ int k; int * kptr; int length; public: Key(int); // ... };	class Key<char*> { char* k; char** kptr; int length; public: Key(char*); // ... };	Class Key<mytype> { mytype k; mytype* kptr; int length; public: Key(mytype); // ... };

The declarations create the following objects:

- i of type Key<int>
- c of type Key<char\*>
- m of type Key<mytype>

Note that these three classes have different names. The types that are contained within the angle braces are not arguments to the class names, but part of the class names themselves. Key<int> and Key<char\*> are class names. Within the context of the example, a class that is called Key (with no template argument list) is undefined.

OS/390 C++ permits default initializers in template arguments under the following conditions:

- They only apply to nontype template arguments.
- They only apply to trailing arguments, like functions.
- Subsequent template declarations can add default initializers, but cannot redefine existing default initializers.
- They only apply to class template declarations, not to function template declarations.

**Note:** OS/390 C++ treats a template that defines a member function of a class template as a function template. Such a template cannot have default initializers.

The following example shows a valid template declaration with default initializers:

### CBC3X15A

```
// This example shows a template declaration
// with default initializers.

#include <stdio.h>

template <class T, int i=1> class X
{
public:
    T s;
    X(int j=4);
    int val(T&)
    {
        return i;
    };
};

template <class T, int i> X<T,i>::X(int j):s(i){
    printf("i=%d    j=%d\n",i,j);
}

void main()
{
    X<int>    myX(2);
    X<int,3> myX2(4);
}
```

---

## Structuring Your Program Using Templates

You can structure your program three ways, by using templates:

- Include the function template definition (both the .h and .c files) in all files that may reference the corresponding template functions.
- Include the function template declaration (the .h file only) in all files that may reference the corresponding template functions. However, include the function definition (both the .h and .c files) in one file only.
- Include the declaration of the function templates in a header file and the definition in a source file that has the same name. When you include the header file in your source, the compiler automatically generates the template functions.

The following examples use two files to illustrate all three methods:

## Structuring Your Program Using Templates

### File stack.h

```
#ifndef _STACK_TPL_H
#define _STACK_TPL_H

template<class T>
class stack
{
private:
    T* v;
    T* p;
    int sz;

public:
    stack( int );
    ~stack();
    void push( T );
};
#endif
```

### File stackdef.h

```
#include "stack.h"

template<class T> stack<T>::stack( int s )
{
    v = p = new T[sz=s];
}

template<class T> stack<T>::~~stack()
{
    delete [] v;
}

template<class T> void stack<T>::push( T a )
{
    *p++ = a;
}
```

To instantiate a stack of 50 ints, you would declare the following in each source file that requires it:

```
stack<int> intStack(50);
```

For method 1, each source file that uses the template should include both `stack.h` and `stackdef.h`.

For method 2, every source file should include `stack.h`. However, only one of the files needs to include `stackdef.h`.

For method 3, every source file should include `stack.h`. The compiler automatically generates the template functions in the TEMPINC PDS. You can use the TEMPINC option to set your own TEMPINC PDS.

You should use the LSEARCH option to include the two PDSs, `USR.INCLUDE.C` and `USR.INCLUDE.H`, which contain the `stack.c` and `stack.h` files, respectively. The syntax for this is:

```
LSEARCH('USR.INCLUDE.+')
```

For information about include files, and the TEMPINC and LSEARCH options, see the *OS/390 C/C++ User's Guide*.

## Class Templates

The relationship between a class template and an individual class is like the relationship between a class and an individual object. An individual class defines how a group of objects can be constructed, while a class template defines how a group of classes can be generated.

Note the distinction between the terms *class template* and *template class*:

*Class template* Is a template used to generate template classes. A class template can be only a declaration, or it can be a definition of the class.

*Template class* Is an instance of a class template.

A template definition is identical to any valid class definition that the template might generate, except for the following:

- The following syntax precedes the class template definition:

```
template < template-argument-list >
```

In the above syntax, *template-argument-list* can include zero or more *type-arguments*, and zero or more *argument-declarations*. The *template-argument-list* must contain at least one argument.

- Types, variables, constants and objects within the class template can be declared with arguments of user-defined type as well as with explicit types (for example, `int` or `char`).
- The *template-argument-list* can include *argument-declarations* (for example, `int a` or `char* b`), which are generally used to define constant values within the created class.

A class template can declare a class without defining it by using an elaborated type specifier, for example:

```
template <class L,class T> class key;
```

Using the type specifier reserves the name as a class template name. All template declarations for a class template must have the same types and number of template arguments. OS/390 C++ allows only one template declaration that contains the class definition.

You can instantiate the class template by declaring a template class. If the template class member function definitions are not inline, you have to define them. When you instantiate a template class, its argument list must match the argument list in the class template declaration.

```
template <class L,class T> class key
{
//      .
//      .
//      .
};

template <class L> class vector
{
//      .
//      .
//      .
};
```

## Class Templates

```
void main ()
{
class key <int, vector<int> >; // instantiate template
}
```

**Note:** When you have nested template argument lists as in the above example, you must have a separating space between the > at the end of the inner list and the one at the end of the outer list. Otherwise, there is an ambiguity between the output operator >> and two template list delimiters >.

Any of the techniques that are used to access ordinary class member objects and functions can access objects and functions of individual template classes. For example, assume you have this class template:

```
template<class T> class vehicle
{
public:
    vehicle() { /* ... */ }    // constructor
    ~vehicle() {};            // destructor
    T kind[16];
    T* drive();
    static void roadmap();
    // ...
};
```

And you have the following declaration:

```
vehicle<char> bicycle; // instantiates the template
```

In the above example, the constructor, the constructed object, and the member function drive() can be accessed with any of the following. (This assumes the standard header file <string.h> is included in the program file.)

constructor	vehicle<char> bicycle; // constructor called automatically // object bicycle created
object bicycle	strcpy (bicycle.kind, "10 speed"); bicycle.kind[0] = '2';
function drive()	char* n = bicycle.drive();
function roadmap()	vehicle<char>::roadmap();

## Class Template Declarations and Definitions

You must declare a class template before declaring a corresponding template class. A class template definition can only appear once in any single compilation unit. You must define a class template before using a template class that requires the size of the class, or refers to members of the class.

The following example declares the class template key before defining it. The declaration of the pointer keyiptr is valid because the example does not need the size of the class. The declaration of keyi, however, causes an error.

```
template <class L> class key;    // class template declared,
                                // not defined yet
                                //
class key<int> *keyiptr;        // declaration of pointer
                                //
class key<int> keyi;            // error, cannot declare keyi
                                // without knowing size
                                //
template <class L> class key    // now class template defined
```



```
{
//      .
//      .
//      .
};
```

If you use a template class before defining the corresponding class template, the compiler issues an error. The compiler considers a class name, with the appearance of a template class name, to be a template class. In other words, angle brackets are valid in a class name only if that class is a template class.

The compiler does not compile the definition of a class template until it requires the definition of a template class. At that point, it compiles the class template definition by using the argument list of the template class to instantiate the template arguments. The compiler flags any errors in the class definition at this time. If the compiler never requires the definition of a class template, it does not compile it. In this case, the compiler will not flag any errors in the definition.

## Reference and Uniqueness

You can only define a class template once within a compilation unit. You cannot declare the class template name to refer to any other template, class, object, function, value, or type in the same scope.

## Nontype Template Arguments

A nontype template argument that is provided within a template argument list is an expression whose value can be determined at compile time. Such arguments must be constant expressions, addresses of functions, objects with external linkage, or addresses of static class members. You normally use nontype template arguments to initialize a class or to specify the sizes of class members.

For nontype integral arguments, the instance argument matches the corresponding template argument as long as the instance argument has a value and sign appropriate to the argument type.

For nontype address arguments, the type of the instance argument must be of the form `identifier` or `&identifier`. The type of the instance argument must match the template argument exactly, except that a function name is changed to a pointer to function type before matching.

The resulting values of nontype template arguments within a template argument list form part of the template class's type. If two template class names have the same template name and if their arguments have identical values, they are the same class.

In the following example, a class template is defined that requires a nontype template `int` argument as well as the type argument:

```
template<class T, int size> class myfilebuf
{
    T* filepos;
    static int array[size];
public:
    myfilebuf() { /* ... */ }
    ~myfilebuf();
    advance(); // function defined elsewhere in program
};
```

## Class Templates

In this example, the template argument size becomes a part of the template class name. It creates an object of such a template class with both the type arguments of the class and the values of any additional template arguments.

From this template, you can create an object `x` and its corresponding template class with arguments `double` and `size=200`. Use a value as the second template argument:

```
myfilebuf<double,200> x;
```

You can also create `x` by using an arithmetic expression:

```
myfilebuf<double,10*20> x;
```

The objects that are created by these expressions are identical because the template arguments evaluate identically. The value 200 in the first expression could have been represented by an expression whose result at compile time is known to be equal to 200, as shown in the second construction.

**Note:** Arguments that contain the `<` symbol or the `>` symbol must be enclosed in parentheses. This prevents it from being parsed as a template argument list delimiter when you use it as a relational operator or a nested template delimiter. For example, the arguments in the following definition are valid:

```
myfilebuf<double, (20>10)> x;          // valid
```

The following definition, however, is not valid because it interprets the greater than operator (`>`) as the closing delimiter of the template argument list:

```
myfilebuf<double, 20>10> x;          // error
```

If the template arguments do not evaluate identically, the objects that are created are of different types:

```
myfilebuf<double,200> x;                // create object x of class
                                         // myfilebuf<double,200>
myfilebuf<double,200.0> y;              // error, 200.0 is a double,
                                         // not an int
```

The instantiation of `y` fails because the value 200.0 is of type `double`, and the template argument is of type `int`.

Consider the following two objects:

```
myfilebuf<double, 128> x
myfilebuf<double, 512> y
```

These two objects belong to separate template classes, and referencing either of these objects later with `myfilebuf<double>` is an error.

A class template does not need to have a type argument if it has nontype arguments. For example, the following template is a valid class template:

```
template<int i> class C
{
    public:
        int k;
        C() { k = i; }
};
```

Declarations such as the following can instantiate this class template:

```
class C<100>;
class C<200>;
```

Again, these two declarations refer to distinct classes because the values of their nontype arguments differ.

## Explicitly Defined Template Classes

You can override the class template definition of a particular template class by providing a class definition for the type of class required. For example, the following class template creates a class for each type that it references, but that class may be inappropriate for a particular type:

```
template<class M> class portfolio
{
    double capital;
    M arr;
    // ...
};
```

Using the applicable template class name can define the type for which the template class is inappropriate. Assuming the inappropriately defined type is `stocks`, you can redefine the class `portfolio<stocks>` as follows:

```
class portfolio<stocks>
{
    double capital;
    stocks yield;
    // ...
};
```

You can define an explicit specialization of a template class before declaring the class template. In particular, you can define a template class such as `portfolio<stocks>` before defining its class template.

---

## Function Templates

A function template allows you to define a group of functions that are the same, except for the types of one or more of their arguments or objects. You must use all type arguments in a function template in the argument list, or in the class qualifier for the function name. You do not need to explicitly specify the type of a template function argument when you call the template function. In this respect, a template function differs from a template class.

Note the distinction between the terms *function template* and *template function*:

*Function template*

Is a template used to generate template functions. A function template can be only a declaration, or it can define the function.

*Template function*

Is a function generated by a function template.

## Example of a Function Template

If you want to create a function `approximate()`, which determines whether two values are within 5% of each other, you can define the following template:

## Function Templates

```
#include <math.h>
template <class T> int approximate (T first, T second)
{
    double aptemp=double(first)/double(second);
    return int(abs(aptemp-1.0) <= .05);
};
```

Assuming that you have two values of type float you want to compare. You can use the approximate function template:

```
float a=3.24, b=3.35;
if (approximate(a,b))
    cout << "a and b are pretty close" << endl;
```

The above example generates the following template function to resolve the call:

```
int approximate(float,float)
```

## Overloading Resolution for Template Functions

OS/390 C++ resolves overloaded template functions in the following order:

1. Looks for a function with an exact type match. It does not include template functions, unless you explicitly declare such functions by using a function declaration. OS/390 C++ performs trivial conversions if they produce an exact type match.
2. Looks for a function template that allows generation of a function with an exact type match. OS/390 C++ performs trivial conversions if they produce an exact type match.
3. Tries ordinary overloading resolution for functions already present. This does not include template functions, unless you have explicitly declared such functions by using a function declaration.

A call to a template function causes an error, and OS/390 C++ does no overloading if the following conditions are true:

- The only available functions for a call are template functions.
- These functions would require nontrivial conversions for the call to succeed.
- You have not explicitly declared these functions.

In the case of the approximate() function template, if the two input values are of different types, overloading resolution does not take place:

```
float a=3.24;
double b=3.35;
if (approximate(a,b)) // error, different types
{ /* ... */ }
```

The solution is to force a conversion to one of the available function types by explicitly declaring the function for the chosen type. To resolve the above example, include the following function declaration:

```
int approximate(double a, double b);
// force conversion of the float to double
```

This declaration creates a function, approximate() that expects two arguments of type double. When you call approximate(a,b), the overloading is resolved by converting variable a to type double.

For more information on argument matching and conversions, see “Trivial Conversions” on page 314, “Standard Type Conversions” on page 167, and “Integral Promotions” on page 167.

## Defining Template Functions

You can generate template functions in all compilation units that contain function template definitions. Consequently, you may want to group function template definitions into one or two compilation units.

## Explicitly Defined Template Functions

In some situations, a function template can define a group of functions in which, for one function type, the function definition would be inappropriate. For instance, consider the following function template:

```
template<class T> int approximate(T first, T second);
```

It is defined in “Example of a Function Template” on page 373, and determines whether two values are within 5% of each other. The algorithm used for this function template is appropriate for numerical values. However, for `char*` values, it indicates whether the *pointers* to two character strings are within 5% of one another. It does not indicate whether the strings themselves are approximately equal. Whether two pointers are within 5% of each other is not useful information. You can define an explicit template function for `char*` values to compare the two strings themselves, character by character.

The following explicitly defined template function compares two strings and returns a value that indicates whether more than 5% of the characters differ between the two strings:

```
#include <string.h>
int approximate(char *first, char *second)
{
    if (strcmp(first,second) == 0)
        return 1; // strings are identical

    double difct=0;
    int maxlen=0;

    if (strlen(first)>strlen(second))
        maxlen=strlen(first);

    else maxlen=strlen(second);
    for (int i=0; i<=maxlen ; ++i)
        if ( first[i] != second[i] ) difct++;
    return int((difct / maxlen) <= .05 );
}
```

Given this definition, the following function call invokes the above explicitly defined function, and no template function is generated:

```
approximate("String A","String B");
```

Explicit definition has the same effect on template overloading resolution as explicit declaration. (See “Overloading Resolution for Template Functions” on page 374 for more information.) Assume that you explicitly define a template function as follows:

```
int approximate(double a, double b) { /* ... */ }
```

Then, consider the following call:

```
double a=3.54;
float b=3.5;
approximate(a,b);
```

## Function Templates

The above call resolves to the following call:

```
approximate(double a, double b)
```

OS/390 C++ converts variable `b` to type `double`.

## Function Template Declarations and Definitions

When you define a template function explicitly within a compilation unit, the compiler uses this definition in preference to any instantiation from the function template. For example, if one compilation unit contains the code:

```
#include <iostream.h>
template <class T> T f(T i) {return i+1;}
void main()
{
    cout << f(2) << endl;
}
```

And another compilation unit contains the following:

```
int f(int i) {return i+2;}
```

When compiled and run, the program prints the number 4 to standard output. This indicates that the compiler uses the explicitly defined function to resolve the call to `f()`.

You must define each template, whether of a class or of a function, at most once within a compilation unit. The same applies to an explicitly defined template class or function. You can declare function templates and class templates many times.

You declare a template class by using its name. You declare a template function if any of the following situations apply:

- A function whose name matches a function template's name you have *declared*, and the compiler can generate an appropriate template function.
- A function whose name matches a function template's name you have *called*, and the compiler can generate an appropriate template function.
- A function whose name matches a function template's name you have called, and you have explicitly defined the template function.
- You take the address of a template function in such a way that instantiation can occur. This means that the pointer to function must supply a return type and supply argument types that can be used to instantiate the template function.

OS/390 C++ instantiates or generates a template function provided the function is not explicitly defined elsewhere in the program, if you reference the function in the following ways:

- Your program declares the function.
- It calls the function.
- It takes the the address of the function.

When your program instantiates a template function, OS/390 C++ compiles the body of the function template by using the template argument list of the template class to instantiate the template arguments. The compiler flags any errors in the function definition at this time. If the function template never generates a template function, OS/390 C++ does not compile it. In this case, the compiler will not flag any errors in the function definition.

## Differences between Class and Function Templates

The name of a template class is a compound name that consists of the template name and the full template argument list that is enclosed in angle braces. Any references to a template class must use this complete name. For example:

```
template <class T, int range> class ex
{
    T a;
    int r;
    // ...
};
//...
ex<double,20> obj1;    // valid
ex<double> obj2;      // error
ex obj3;              // error
```

C++ requires this explicit naming convention to ensure that it can generate the appropriate class.

A template function chooses the name of its function template, and the particular function to resolve a given template function call. It determines the name and the function by the type of the calling arguments. In the following example, the call `min(a,b)` is effectively a call to `min(int a, int b)`. The call, `min(af, bf)`, is effectively a call to `min(float a, float b)`:

### CBC3X15B

```
// This example illustrates a template function.

template<class T> T min(T a, T b)
{
    if (a < b)
        return a;
    else
        return b;
}

void main()
{
    int a = 0;
    int b = 2;
    float af = 3.1;
    float bf = 2.9;
    cout << "Here is the smaller int " << min(a,b) << endl;
    cout << "Here is the smaller float " << min(af, bf) << endl;
}
```

## Member Function Templates

“Function Templates” on page 373, defines a function template outside of any template class. However, functions in C++ are often member functions of a class. Consider if you want to create a class template, and a set of function templates to go with that class template. If so, you do not have to create the function templates explicitly as long as the function definitions are contained within the class template. Any member function (inline or not inline) that is declared within a class template is implicitly a function template. When you declare a template class, it implicitly generates template functions for each function that is defined in the class template.

You can define template member functions three ways:

## Member Function Templates

- Explicitly at file scope for each type used to instantiate the template class. For example:

```
template <class T> class key
{
public:
    void f(T);
};
void key<char>::f(char) { /* ... */ }
void key<int>::f(int ) { /* ... */ }

void main()
{
    int i = 9;
    key< int> keyobj;
    keyobj.f(i);
}
```

- At file scope with the template arguments. For example:

```
template <class T> class key
{
public:
    void f(T);
};
template <class T> void key <T>::f(T) { /* ... */ }

void main()
{
    int i = 9;
    key< int> keyobj;
    keyobj.f(i);
}
```

- Inlined in the class template itself. For example:

```
template <class T> class key
{
public:
    void f(T) { /* ... */ }
};

void main()
{
    int i = 9;
    key< int> keyobj;
    keyobj.f(i);
}
```

Use member function templates to instantiate any functions that are not explicitly generated. If you have both a member function template and an explicit definition, OS/390 C++ uses the explicit definition.

Do not use the template argument in a constructor name. For example:

```
template<class L> class Key
{
    Key();           // default constructor
    Key( L );        // constructor taking L by value
    Key<L>( L );     // error, <L> implicit within class template
};
```

The declaration `Key<L>(L)` is an error because the constructor does not use the template argument. Assuming that removing the offending line corrected this class template, you can define a function template for the class template's constructor:

```
// Constructor contained in function template:
template<class L>
    Key<L>::Key(int) { /* ... */ }
```



```
// valid, constructor template argument assumed template<class L>

Key<L>::Key<L>(int) { /* ... */ }
/* error, constructor template argument <L> implicit
   in class template argument */
```

A template function name does not include the template argument. The template argument does, however, appear in the template class name if a member function of a template class is defined or declared outside of the class template. Consider the following definition:

```
Key<L>::Key(int) { /* ... */ }
```

The above definition is valid because `Key<L>` (with template argument) refers to the class, while `Key(int)` { /\* ... \*/ } refers to the member function.

---

## Friends and Templates

You can declare a friend function in a class template as a single function shared by all classes created by the template. Or, you can declare it as a template function that varies from class to class within the class template. For example:

```
template<class T> class portfolio
{
    //...
    friend void taxes();
    friend void transact(T);
    friend portfolio<T>* invest(portfolio<T>*);
    friend portfolio* divest(portfolio*);          //error
    // ...
};
```

In this example, each declaration has the following characteristics:

`taxes()`

Is a single function that can access private and protected members of any template class generated by the class template. Note that `taxes()` is not a template function.

`transact(T)`

Is a function template that declares a distinct function for each class generated by the class template. The only private and protected members that functions that are generated from this template can access, are the private and protected members of their template class.

`invest(portfolio<T>*)`

Is a function template whose return and argument types are pointers to objects of type `portfolio<T>`. Each class that is generated by the class template will have a friend function of this name. And, each such function will have a pointer to an object of its own class as both its return type and its argument type.

`divest(portfolio*)`

Is an error because `portfolio*` attempts to point to a class template. A pointer to a class template is undefined and produces an error. This statement can be corrected by using the syntax of the `invest()` function template instead.

All friend functions in this example are declared but not defined. Consequently, you could create a set of function templates to define those functions that are

## Friends and Templates

implicitly template functions. (That is, all the valid functions except `taxes()`.) OS/390 C++ then uses the function templates to instantiate the template functions as required.

---

## Static Data Members and Templates

A static declaration within a class template declares a static data member for each template class generated from the template. The static declaration can be of template argument type or of any defined type.

Like member function templates, you can explicitly define a static data member of a template class at file scope for each type used to instantiate a template class. For example:

```
template <class T> class key
{
public:
    static T x;
};
int key<int>::x;
char key<char>::x;
void main()
{
    key<int>::x = 0;
}
```

You can also define a static data member of a template class using a template definition at file scope. For example:

```
template <class T> class key
{
public:
    static T x;
};
template <class T> T key<T> ::x; // template definition
void main()
{
    key<int>::x = 0;
}
```

In the following example:

```
template<class L> class Key
{
    static L k;
    static L* kptr;
    static int length;
    // ...
}
```

The definitions of static variables and objects must be instantiated at file scope. If the classes `Key<int>` and `Key<double>` are instantiated from this template, and no template definitions exist, the following static data members must be explicitly defined at file scope, or an error occurs:

```
int Key<int>::k, Key<int>::length, Key<double>::length
int* Key<int>::kptr;
double Key<double>::k;
double* Key<double>::kptr = 0;
```

---

## Chapter 17. C++ Exception Handling

This chapter describes the OS/390 C/C++ implementation of C++ exception handling. It discusses the following topics:

- “C++ Exception Handling Overview”
- “Formal and Informal Exception Handling” on page 382
- “Using Exception Handling” on page 382
- “Transferring Control” on page 384
- “Constructors and Destructors in Exception Handling” on page 391
- “Exception Specifications” on page 393
- “Special Exception Handling Functions” on page 395

### Related Information

- “Chapter 6. Expressions and Operators” on page 133
- “Chapter 14. Special C++ Member Functions” on page 325

---

## C++ Exception Handling Overview

*Exception handling* enables a function that encounters an unusual situation to throw an exception and pass control to a direct or indirect caller of that function. The caller may or may not be able to handle the exception. A handler is code that intercepts an exception. Regardless of whether or not the caller can handle an exception, it may rethrow the exception so it can be intercepted by another handler.

C++ provides three language constructs to implement exception handling:

- Try blocks
- Catch blocks
- Throw expressions

Within a function, a throw expression can flag any unusual situation. The throw expression is of type void. Your program can throw an object to pass information back to the caller. It can throw any object, including the object that caused the exception, or an object that was constructed when the exception occurred.

You should enclose a throw expression, or a call to a function that may throw an exception, within a try block. If the called function throws an exception and you have defined an exception handler to catch the type of the object that is thrown, your program executes the exception handler. In C++, a catch block implements an exception handler. One or more catch clauses must accompany a try block; otherwise the compiler flags an error.

A catch block follows immediately after a try statement or immediately after another catch block. A catch block includes a parenthesized exception declaration that contains optional qualifiers, a type, and an optional variable name. The declaration specifies the type of object that the exception handler may catch. Once the exception handler catches an exception, it executes the body of the catch block. If no handler catches an exception, OS/390 C/C++ terminates the program.

## C++ Exception Handling Overview

Exception handling is not strictly synonymous with error handling, because the implementation allows the passing of an exception whether or not an error actually occurred. You can use exception handlers for things other than handling errors. For example, you can transfer control back to the original caller of a function. You might use this feature if you wanted to process the Quit key in a program and transfer control back to the driver program when the user types Quit. To do this, you could use exception handlers to throw an object back to the driver.

---

## Formal and Informal Exception Handling

While the exception handling features of C++ offer a formal mechanism for handling exceptions (language implemented), in many situations informal exception handling (logic implemented) is more appropriate.

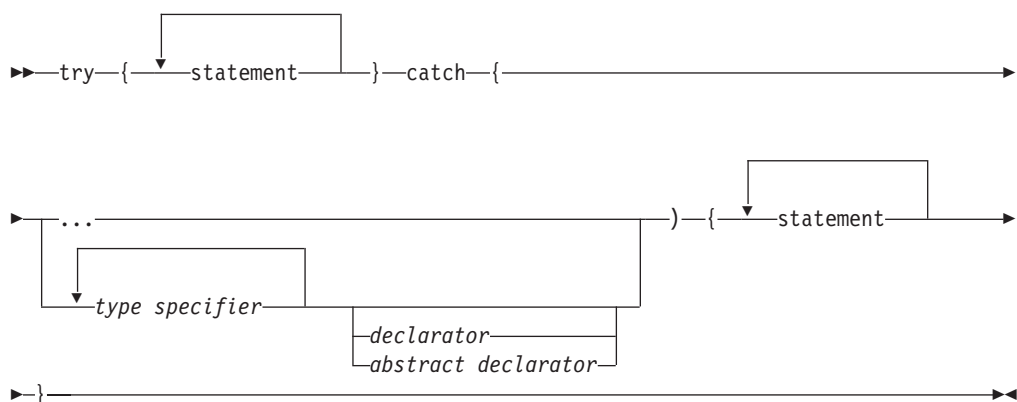
Generally, you should implement formal exception handling in libraries, classes, and functions that are likely to be accessed by several programs or programmers. It should also be used in classes and functions that are repeatedly accessed within a program but are not well-suited to handling their exceptions themselves. Formal exception handling is designed for exceptional circumstances, and it is not guaranteed to be efficient. Program performance is usually not affected when you do not invoke formal exception handling, although it can inhibit some optimizations.

Informal exception handling, in which an appropriate action is defined if an error or exception occurs, is often more suitable for handling errors. For example, you can easily and clearly handle a simple error by testing the input for validity by entering incorrect input. You can also request the input again if the original input is incorrect.

---

## Using Exception Handling

The three keywords that are designed for exception handling in C++ are try, catch, and throw.



The syntax for the keyword throw is:



To implement an exception handler, following these steps:

1. If functions will be used by many programs, code them so that error detection throws an exception. The throw expression generally throws an object. You may create the expression explicitly for exception handling, or it may be the object that causes the exception handler to detect the error. The following example throws a problem object:

```

:

int input=0;
cout << "Enter a number between 1 and 10:";
cin >> input;

if (input < 1 || input >> 10);
    throw(input); //throw the actual problem object
:
  
```

The following example throws an object for the purpose of exception handling:

```

:

int input=0;
cout << "Enter a number between 1 and 10:";
cin >> input;

if (input < 1 || input >> 10)
    throw(out_of_range_object); //throw object to tell handler
                                //what happened
  
```

2. Function calls that you anticipate might produce an exception must be enclosed in braces and preceded by the keyword try. A try statement in a caller anticipates exceptions.
3. You must code one or more catch blocks immediately following the try block. Each catch block identifies what type or class of objects it can catch:
  - a. If the object that is thrown matches the type of a catch expression, OS/390 C/C++ passes control to that catch block.
  - b. If the object that is thrown does not match the first catch block, OS/390 C/C++ searches subsequent catch blocks for a matching type.
  - c. If it cannot find a match, OS/390 C/C++ continues the search in all enclosing try blocks, and then in the code that called the current function.
  - d. If no match is found after all try blocks are searched, a call to terminate() is made.

For information on the default handlers of uncaught exceptions, see “Special Exception Handling Functions” on page 395.

#### Notes:

1. You can throw any object if you can copy and destroy it in the function from which the throw occurs.
2. You should never throw exceptions from a C language signal handler. The result is undefined, and can cause program termination.

## Using Exception Handling

A catch argument causes an error if it is a value argument and if OS/390 C/C++ cannot generate a copy of it. For example:

```
class B {
public:
    B();
    B(B&);
};
// the following catch block will cause an error
//
catch(const B x)
{
    // .
    // .
    // .
}
```

The catch block causes an error because the compiler does not know the type of the object that is thrown at compile time. It assumes that the type of the thrown object is the same as the type of the catch argument. In the above example, the compiler assumes the thrown object is type `const B`. The compiler uses a copy constructor on the thrown argument to create the catch argument. Because there is no copy constructor for class `B` that accepts `const B` as an input argument, the compiler cannot perform the construction, and an error occurs. Similarly, a throw expression causes an error if OS/390 C/C++ cannot generate a copy of the value of the expression that the handler throws.

---

## Transferring Control

C++ implements the *termination model* of exception handling. In the termination model, when your program throws an exception, control never returns to the *throw point*. The throw point is the point in program execution where the exception occurred.

C++ exception handling does not implement the *resumption model* of exception handling, which allows an exception handler to correct the exception and then return to the throw point.

When your program throws an exception, it passes control out of the throw expression, and out of the try block that anticipated the exception. It passes control to the catch block whose exception type matches the object that is thrown. The catch block handles the exception as appropriate. If the catch block ends normally, the flow of control passes over all subsequent catch blocks.

When an exception is not thrown from within a try block, the flow of control continues normally through the block. It passes over all catch blocks following the try block.

An exception handler cannot return control to the source of the error by using the return statement. A return issued in this context returns from the function that contains the catch block.

Consider if your program throws an exception, and no try block is active, or if a try block is active and the catch block exception declaration does not match the object thrown. In this case, OS/390 C/C++ issues a call to `terminate()`, which in turn calls `abort` to terminate the program. The `abort` C library function is defined in the standard header file `<stdlib.h>`.

For more information on `terminate()`, see “Special Exception Handling Functions” on page 395.

The following example illustrates the basic use of `try`, `catch`, and `throw`. The program prompts for numerical input and determines the input’s reciprocal. Before it attempts to print the reciprocal to standard output, it checks that the input value is nonzero to avoid a division by zero. If the input is zero, the program throws an exception, and the catch block catches the exception. If the input is nonzero, the reciprocal is printed to standard output.

## CBC3X16A

```
// This example illustrates the basic use of
// try, catch, and throw.

#include <iostream.h>
#include <stdlib.h>
class IsZero { /* ... */ };
void ZeroCheck( int i )
{
    if (i==0)
        throw IsZero();
}
void main()
{
    double a;

    cout << "Enter a number: ";
    cin >> a;
    try
    {
        ZeroCheck( a );
        cout << "Reciprocal is " << 1.0/a << endl;
    }
    catch ( IsZero )
    {
        cout << "Zero input is not valid" << endl;
        exit(1);
    }
    exit(0);
}
```

This example provides a simple illustration of formal exception handling. You could make it more efficient by using informal exception handling.

Another example of exception handling under C++ is shown below. The `testeh()` routine is called from the `main` function. `testeh()` first throws the integer 6, which is handled by a catch clause within `testeh()`.

Next, it throws a short `int`, `k`. There are no catch clauses in `testeh()` which can handle a short `int`. The `testeh()` routine ends, and destructors are called for the objects it created. Control passes back to the `main` function. It contains a catch clause which can handle the thrown short `int`. The example calls destructors for the remaining objects as the program ends.

**CBC3X16F**

```

// This is a more extensive example of
// C++ exception handling.

#include <iostream.h>

int testeh(void);
class A {
    int i;
public:
    A(int j) { i = j; cout << "A ctor: i=" << i << '\n'; }
    ~A() { cout << "A dtor: i=" << i << '\n'; }
};
class B {
    char c;
public:
    B(char d) { c = d; cout << "B ctor: c=" << c << '\n'; }
    ~B() { cout << "B dtor: c=" << c << '\n'; }
};
A globA(1);
B globB('a');
main()
{
    int rc = 55;
    A a(1001);

    try {
        cout << "calling testeh\n";
        testeh();
    }
    catch(char c) { cout << "caught char\n"; }
    catch(short s) { cout << "caught short s = " << s << '\n'; }
    catch(int i) { cout << "caught int i = " << i << '\n'; }
    cout << "rc = " << rc << '\n';
    return(rc);
}

testeh() throw(int,short)
{
    A a(2001);
    B b('k');
    short k=12;

    try {
        cout << "testeh running\n";
        throw (6);
    }
    catch(char c) { cout << "testeh caught char\n"; }
    catch(int j) { cout << "testeh caught int j = " << j << '\n';
        try {
            cout << "testeh throwing a short\n";
            throw(k);
        }
        catch(char d) { cout << "char d caught\n"; }
    }
    cout << "this line should not be printed\n";
    return(0);
}

```



The expected output from this program is:

```
A ctor: i=1
B ctor: c=a
A ctor: i=1001
calling testeh
A ctor: i=2001
B ctor: c=k
testeh running
testeh caught int j = 6
testeh again throwing a short
B dtor: c=k
A dtor: i=2001
caught short s = 12
rc = 55
A dtor: i=1001
B dtor: c=a
A dtor: i=1
```

## Catching Exceptions

You can declare a handler to catch many types of exceptions. You declare the allowable objects that a function can catch in the parentheses that follow the catch keyword (the *catch argument*). You can catch objects of the fundamental types, base and derived class objects, references, and pointers to all of these types. You can also catch `const` and `volatile` types.

You can also use the `catch(...)` form of the handler to catch all thrown exceptions that have not been caught by a previous catch block. The ellipsis in the catch argument indicates that this handler can handle any exception that is thrown.

If an exception is caught by a `catch(...)` block, there is no direct way to access the object thrown. Information about an exception caught by `catch(...)` is very limited.

You can declare an optional variable name if you want to access the thrown object in the catch block.

A catch block can only catch accessible objects. The object that is caught must have an accessible copy constructor. For more information on access, see “Member Access” on page 304; on copy constructors, see “Copy by Initialization” on page 341.

## Matching Exceptions Thrown and Exceptions Caught

An argument in the catch argument of a handler matches an argument in the *expression* of the throw expression (throw argument) if any of the following conditions is met:

- The catch argument type matches the type of the thrown object.
- The catch argument is a public base class of the thrown class object.
- The catch specifies a pointer type, and the thrown object is a pointer type that OS/390 C/C++ can convert to the pointer type of the catch argument by standard pointer conversion. “Pointer Conversions” on page 168 describes pointer conversion.

**Note:** If the type of the thrown object is `const` or `volatile`, the catch argument must also be a `const` or `volatile` for a match to occur. However, a `const`, `volatile`, or reference type catch argument can match a nonconstant,

## Transferring Control

nonvolatile, or nonreference object type. A nonreference catch argument type matches a reference to an object of the same type.

## Order of Catching

Always place a catch block that catches a derived class before a catch block that catches the base class of that derived class (that follows a try block). Consider if a catch block for objects of a base class is followed by a catch block for objects of a derived class of that base class. Then, the latter block is flagged as an error.

A catch block of the form `catch(...)` must be the last catch block following a try block or an error occurs. This placement ensures that the `catch(...)` block does not prevent more specific catch blocks from catching exceptions intended for them.

## Nested Try Blocks

Consider when try blocks are nested and a throw occurs in a function called by an inner try block. In that case, your program transfers control outward through the nested try blocks until it finds the first catch block whose argument matches the argument of the throw expression.

For example:

```
try
{
    func1();
    try
    {
        func2();
    }
    catch (spec_err) { /* ... */ }
    func3();
}
catch (type_err) { /* ... */ }
// if no throw is issued, control resumes here.
```

If the above example throws `spec_err` within the inner try block (in this case, from `func2()`), the inner catch block catches the exception. Assuming this catch block does not transfer control, it calls `func3()`. If the example throws `spec_err` after the inner try block (for instance, by `func3()`), the handler does not catch it, and OS/390 C/C++ calls the function `terminate()`.

If the exception thrown from `func2()` in the inner try block is `type_err`, the program skips out of both try blocks to the second catch block without invoking `func3()`. No appropriate catch block exists following the inner try block.

If the entire try block in the example is in a function that has a throw list and does not include `spec_err` on its throw list, OS/390 C/C++ calls `unexpected()`. The function `unexpected()` is discussed in “Special Exception Handling Functions” on page 395.

You can also nest a try block within a catch block.

## Rethrowing an Exception

If a catch block cannot handle the particular exception it has caught, you can rethrow the exception. The rethrow expression, `throw` with no argument, causes the originally thrown object to be rethrown.

The handler has already caught the exception at the scope in which the rethrow expression occurs. Consequently, the exception is rethrown to the next dynamically enclosing try block. Therefore, catch blocks at the scope in which the rethrow expression occurred cannot handle it. Any catch blocks following the dynamically enclosing try block have an opportunity to catch the exception.

In the following example, `catch(FileIO)` catches any object of type `FileIO`, and any objects that are public base classes of the `FileIO` class. The example then checks for those exceptions it can handle. For any exception it cannot handle, it issues a rethrow expression to rethrow the exception. This allows another handler in a dynamically enclosing try block to handle the exception.

### CBC3X16B

// This example illustrates rethrowing an exception.

```
#include <iostream.h>
class FileIO
{
public:
    int notfound;
    int endfile;
    FileIO(); // initialize data members
    // the following member functions throw an exception
    // if an input error occurs
    void advance(int x);
    void clear();
    void put(int x, int y);
};
//      .
//      .
//      .
void f()
{
    FileIO fio;
    try
    {
        // call member functions of FileIO class
        fio.advance (1);
        fio.clear();
        fio.put(1,-1);
    }
    catch(FileIO fexc)
    {
        if (fexc.notfound)
            cout << "File not Found" << endl;
        else if (fexc.endfile)
            cout << "End of File" << endl;
        else
            throw;           // rethrow to outer handler
    }
    catch(...) { /* ... */ } // catch other exceptions
}
```

## Transferring Control

```
main()
{
    try
    {
        f();
    }
    catch(FileIO) { cout << "Outer Handler" << endl; }
}
```

The rethrow expression can be caught by any catch whose argument matches the argument of the exception originally thrown. Note that in this example, the `catch(...)` will not catch the rethrow expression. When the example issues a rethrow expression, it passes control out of the scope of the function `f()` into the next dynamically enclosing block.

## Using a Conditional Expression in a Throw Expression

You can use a conditional expression as a throw expression, as the following example demonstrates:

### CBC3X16C

```
// This example illustrates a conditional expression
// used as a throw expression.
```

```
#include <iostream.h>
void main() {
    int doit = 1;
    int dont = 0;
    float f = 8.9;
    int i = 7;
    int j = 6;
    try { throw(doit ? i : f); }
    catch (int x)
    {
        cout << "Caught int " << x << endl;
    }
    catch (float x)
    {
        cout << "Caught float " << x << endl;
    }
    catch (double x)
    {
        cout << "Caught double " << x << endl;
    }
    catch (...)
    {
        cout << "Caught something " << endl;
    }
}
```

This example produces the following output because `i` is of type `int`:

Caught float 7

At first glance, it looks as if the block that catches integer values should do the catch. However, the example converts `i` to a float value in the try block because it is in a conditional expression with the float value `f`. Consider replacing the try block in the example with the following try block:

```
try { throw doit ? i : j; }
```

The following output is produced:

## Constructors and Destructors in Exception Handling

Suppose your program throws an exception and passes control to a catch block that follows a try block. Then, OS/390 C/C++ calls destructors for all automatic objects constructed since the beginning of the try block that is directly associated with that catch block. Suppose your program throws an exception during construction of an object that consists of subobjects or array elements. Then, OS/390 C/C++ calls destructors only for those subobjects or array elements that are successfully constructed before the program throws the exception. OS/390 C/C++ calls a destructor for a local static object only if your program has successfully constructed the object.

For more information on constructors and destructors, see “Constructors and Destructors Overview” on page 325.

Suppose a destructor detects and then throws an exception. You can catch the exception if the caller of the destructor is contained within a try block and you have coded an appropriate catch.

Suppose a function that is called from an inner try block throws an exception that is caught by an outer try block, because the inner try block did not have an appropriate handler. All objects constructed within the outer and all inner try blocks are destroyed. If the thrown object has a destructor, OS/390 C/C++ does not call the destructor until the handler catches and handles the exception.

A throw expression throws an object, and a catch statement can catch an object. Consequently, the object that is thrown enables error-related information to be transferred from the point at which an exception is detected to the exception's handler. If you throw an object with a constructor, you can construct an object that contains information relevant to the catch expression.

In the following example, the function `divide()` throws an object of class `DivideByZero`. The constructor copies the string "Division by zero" into the char array `errmsg`. Because `DivideByZero` is a derived class of class `Matherr`, the catch block for `Matherr` catches the thrown exception. The catch block can then access information that is provided by the thrown object, (in this case, the text of an error message).

### CBC3X16D

```
// This example illustrates constructors and
// destructors in exception handling.

#include <string.h>           // needed for strcpy
#include <iostream.h>
class Matherr { public: char errmsg[30]; };
class DivideByZero : public Matherr
{
public:
    DivideByZero() {strcpy (errmsg, "Division by zero");}
};
double divide(double a, double b)
```

## Constructors and Destructors

```
{
    if (b == 0) throw DivideByZero();
    return a/b;
}

void main()
{
    double a=7,b=0;
    try {divide (a,b);}
    catch (Matherr xx)
    {
        cout << xx.errname << endl;
    }
}
```

You can use exception handling in conjunction with constructors and destructors to provide resource management. This resource management ensures that all locked resources are unlocked when your program throws an exception.

```
class data
{
public:
    void lock();        // prevent other users from
                        // changing the object
    void unlock();      // allow other users to change
                        // the object
};

void q(data&), bar(data&);
//      .
//      .
//      .
main()
{
    data important;
    important.lock();
    q(important);
    bar(important);
    important.unlock();
}
```

If `q()` or `bar()` throw an exception, `important.unlock()` will not be called and the data will stay locked. Use a helper class to write an *exception-aware* program for resource management to correct this problem.

```
class data
{
public:
    void lock();        // prevent other users from
                        // changing the object
    void unlock();      // allow other users to change
                        // the object
};

class locked_data      // helper class
{
    data& real_data;
public:
    locked_data(data& d) : real_data(d)
        {real_data.lock();}
    ~locked_data() {real_data.unlock();}
};

void q(data&), bar(data&);
//      .
//      .
//      .
main()
{
    data important;
```

```

        locked_data my_lock(important);
        q(important);
        bar(important);
    }

```

In this case, if `q()` or `bar()` throws an exception, the destructor for `my_lock` will be called, and the data will be unlocked.

## Exception Specifications

C++ provides a mechanism that limits a given function to throwing only a specified list of exceptions. An exception specification at the beginning of any function acts as a guarantee to the function's caller that the function will not directly or indirectly throw any exception not contained in the exception specification. For example, consider the following function:

```
void translate() throw(unknown_word,bad_grammar) { /* ... */ }
```

The above function explicitly states that it will not throw any exception other than `unknown_word` or `bad_grammar`. The function `translate()` must handle any exceptions thrown by functions it might call, unless those exceptions are specified in the exception specification of `translate()`. Consider if an exception is thrown by a function called by `translate()` and the exception is not handled by `translate()` or contained in the exception specification of `translate()`. Then, OS/390 C/C++ calls `unexpected()`. The function `unexpected()` is discussed in "Special Exception Handling Functions" on page 395.

There are qualifications to the rule about throwing only a specified list of exceptions. If a class `A` is included in the exception specification of a function, the function will also allow exception objects of any classes that are publicly derived from class `A`. Also, if a pointer type `B*` is included in the exception specification of a function, the function will allow exceptions of type `B*` or of pointers to any type publicly derived from `B*`.

## Exception Specification Syntax

The syntax of the exception specification is:


```

    ►► throw (  )
               ^
               |
             type

```

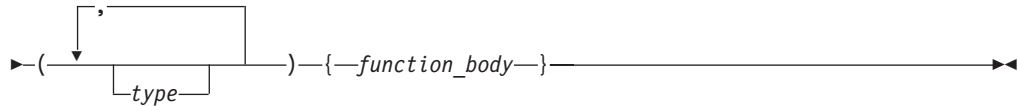
The syntax of a function definition that includes an exception specification is:

```

    ►► return_type function_name (  ) throw
                                   ^
                                   |
                                argument

```

## Exception Specifications



An exception specification is not part of a function's type. If an exception is thrown from a function that has not specified the thrown exception in its exception specification, the result is a call to the function `unexpected()`, which is discussed in "Special Exception Handling Functions" on page 395.

## Empty Exception Specifications

A function with an empty `throw()` specification guarantees that the function will not throw any exceptions.

## Functions without an Exception Specification

A function without an exception specification allows any object to be thrown from the function.

## Other Exception Specifications

The compiler does not prevent an exception specification from defining more valid exceptions than the set of exceptions the function may actually throw. Such an error is detected only at run time, and only if the unspecified exception is thrown.

In the following example, `NameTooShort` is thrown from within a function that explicitly states that it will only throw `NameTooLong`. This is a valid function, although at run time, if `NameTooShort` is thrown, a call to `unexpected()` will be made.

```
#include <string.h>           // needed for strlen
class NameTooLong {};
class NameTooShort {};

void check(char* fname) throw (NameTooLong)
{
    if ( strlen(fname)<4 ) throw NameTooShort();
}
```

If a function with an exception specification calls a subfunction with a less restrictive exception specification (one that contains more objects than the calling function's exception specification), any thrown objects from within the subfunction that are not handled by the subfunction, and that are not part of the outer function's specification list, must be handled within the outer function. If the outer function fails to handle an exception not in its exception specification, a call to `unexpected()` is made.



## Special Exception Handling Functions

Not all thrown errors can be caught and successfully dealt with by a catch block. In some situations, the best way to handle an exception is to terminate the program. Two special library functions are implemented in C++ to process exceptions not properly handled by catch blocks or exceptions thrown outside of a valid try block. These functions are `unexpected()` and `terminate()`.

### **unexpected()**

When a function with an exception specification throws an exception that is not listed in its exception specification, the function `void unexpected()` is called. Next, `unexpected()` calls a function specified by the `set_unexpected()` function. By default, `unexpected()` calls the function `terminate()`. In turn, `terminate()` calls `abort` by default, terminating the program.

Although `unexpected()` cannot return, it may throw an exception. The search for a handler starts at the call of the function whose exception specification was violated. For more information, see “`set_unexpected()` and `set_terminate()`”.

### **terminate()**

In some cases, the exception handling mechanism fails and a call to `void terminate()` is made. This `terminate()` call occurs in any of the following situations:

- When `terminate()` is explicitly called
- When no catch can be matched to a thrown object
- When the stack becomes corrupted during the exception-handling process
- When a system defined `unexpected()` is called

The `terminate()` function calls a function specified by the `set_terminate()` function. By default, `terminate` calls `abort`, which exits from the program.

A `terminate` function cannot return to its caller, either by using `return` or by throwing an exception.

### **set\_unexpected() and set\_terminate()**

When invoked, the function `unexpected()` calls the function most recently supplied as an argument to `set_unexpected()`. If `set_unexpected()` has not been called yet, `unexpected()` calls `terminate()`.

The function `terminate()`, when invoked, calls the function most recently supplied as an argument to `set_terminate()`. If `set_terminate()` has not yet been called, `terminate()` calls `abort`, which ends the program.

You can use `set_unexpected()` and `set_terminate()` to register functions you define to be called by `unexpected()` and `terminate()`. `set_unexpected()` and `set_terminate()` are included in the standard header files `<terminate.h>` and `<unexcept.h>`. Each of these functions has as its return type and its argument type a pointer to function with a void return type and no arguments. The pointer to function you supply as the argument becomes the function called by the

## Special Exception Handling Functions

corresponding special function: the argument to `set_unexpected()` becomes the function called by `unexpected()`, and the argument to `set_terminate()` becomes the function called by `terminate()`.

Both `set_unexpected()` and `set_terminate()` return a pointer to the function that was previously called by their respective special functions (`unexpected()` and `terminate()`). By saving the return values, you can restore the original special functions later so that `unexpected()` and `terminate()` will once again call `terminate()` and `abort`.

If you use `set_terminate()` to register your own function, the final action of that program should be to exit from the program. If you attempt to return from the function called by `terminate()`, `abort` is called instead and the program ends.

**Note:** Providing a call to `longjmp()` inside a user-defined `terminate` function can transfer execution control to some other desired point. When you call `longjmp`, objects existing at the time of a `setjmp` call will still exist, but some objects constructed after the call to `setjmp` might not be destructed.

The `longjmp` and `setjmp` functions are described in the *OS/390 C/C++ Run-Time Library Reference*.

## Example of Using the Exception Handling Functions

The following example shows the flow of control and special functions used in exception handling:

```
#include <terminate.h>
#include <unexpected.h>
#include <iostream.h>
class X { /* ... */ };
class Y { /* ... */ };
class A { /* ... */ };
// pfv type is pointer to function returning void
typedef void (*pfv)();
void my_terminate()
{
    cout << "Call to my terminate" << endl; }
void my_unexpected()
{
    cout << "Call to my unexpected" << endl; }
void f() throw(X,Y)      // f() is permitted to throw objects of class
                        // types X and Y only
{
    A aobj;
    throw(aobj); // error, f() throws a class A object
}
main()
{
    pfv old_term = set_terminate(my_terminate);
    pfv old_unex = set_unexpected(my_unexpected);
    try{ f(); }
    catch(X)      { /* ... */ }
    catch(Y)      { /* ... */ }
    catch (...)   { /* ... */ }

    set_unexpected(old_unex);
    try { f(); }
    catch(X)      { /* ... */ }
    catch(Y)      { /* ... */ }
    catch (...)   { /* ... */ }
}
```

At run time, this program behaves as follows:

## Special Exception Handling Functions

1. The call to `set_terminate()` assigns to `old_term` the address of the function last passed to `set_terminate()` when `set_terminate()` was previously called.
2. The call to `set_unexpected()` assigns to `old_unex` the address of the function last passed to `set_unexpected()` when `set_unexpected()` was previously called.
3. Within a try block, function `f()` is called. Because `f()` throws an unexpected exception, a call to `unexpected()` is made. `unexpected()` in turn calls `my_unexpected()`, which prints a message to standard output and returns.
4. The second call to `set_unexpected()` replaces the user-defined function `my_unexpected()` with the saved pointer to the original function (`terminate()`) called by `unexpected()`.
5. Within a second try block, function `f()` is called once more. Because `f()` throws an unexpected exception, a call to `unexpected()` is again made. `unexpected()` automatically calls `terminate()`, which calls the function `my_terminate()`.
6. `my_terminate()` displays a message. It returns, and the system calls `abort`, which terminates the program.

At run time, the following information is displayed, and the program ends:

```
Call to my_unexpected  
Call to my_terminate
```

**Note:** The catch blocks following the try block are not entered, because the exception was handled by `my_unexpected()` as an unexpected throw, not as a valid exception.

## Special Exception Handling Functions

---

## Part 4. Appendixes



---

## Appendix A. C and C++ Compatibility

The differences between ANSI/ISO C and C++ fall into three categories:

- Constructs found in C++ but not in ANSI/ISO C
- Constructs found in both C++ and ANSI/ISO C, but treated differently in the two languages
- Interactions with other products that do not support C++

---

### C++ Constructs Not Found in ANSI/ISO C

C++ contains many constructs that are not found in ANSI/ISO C:

- Single-line comments beginning with `//` (See “Comments” on page 54)
- Scope operator (See “C++ Scope Resolution Operator (`::`)” on page 137)
- Free store management using the operators `new` and `delete` (See “C++ new Operator” on page 147 and “C++ delete Operator” on page 151)
- Linkage specification for functions (See “Linkage Specifications — Linking to non-C++ Programs” on page 50)
- Reference types (See “C++ References” on page 129)
- Default arguments for functions (See “Default Arguments in C++ Functions” on page 190)
- Inline functions (See “C++ Inline Functions” on page 195)
- Classes (See “Chapter 11. C++ Classes” on page 281)
- Anonymous unions (See “Anonymous Unions in C++” on page 117)
- Overloaded operators and functions (See “Chapter 13. C++ Overloading” on page 311)
- Class templates and function templates (See “Class Templates” on page 369 and “Function Templates” on page 373)
- Exception handling (See “Chapter 17. C++ Exception Handling” on page 381)

**Note:** The OS/390 C/C++ compiler also supports anonymous unions in C, but the implementation is slightly different from C++. For more information, see “Anonymous Unions in C” on page 116.

---

### Constructs Found in Both C++ and ANSI/ISO C

Because C++ is based on ANSI/ISO C, the two languages have many constructs in common. The use of some of these shared constructs differs, as shown here.

#### Character Array Initialization

In C++, when you initialize character arrays, a trailing `'\0'` (zero of type `char`) is appended to the string initializer. You cannot initialize a character array with more initializers than there are array elements.

## Constructs Found in Both C++ and ISO/ANSI C

In ANSI/ISO C, space for the trailing `'\0'` can be omitted in this type of initialization.

The following initialization, for instance, is not valid in C++:

```
char v[3] = "asd"; // not valid in C++, valid in ANSI/ISO C
```

because four elements are required. This initialization produces an error because there is no space for the implied trailing `'\0'` (zero of type `char`).

For more information, see “Initializing Arrays” on page 102.

## Character Constants

A character constant has type `char` in C++ and `int` in ANSI/ISO C.

For more information, see “Character Constants” on page 64.

## Class and typedef Names

In C++, a class and a `typedef` cannot both use the same name to refer to a different type within the same scope (unless the `typedef` is a synonym for the class name). In C, a `typedef` name and a `struct` tag name declared in the same scope can have the same name because they have different name spaces. For example:

```
void main ()
{
    typedef double db;
    struct db; // error in C++, valid in ANSI/ISO C

    typedef struct st st; // valid ANSI/ISO C and C++
}
```

For more information on `typedef`, see “`typedef`” on page 84. For information on class types, see “Chapter 11. C++ Classes” on page 281. For information on structures, see “Structures” on page 106.

## Class and Scope Declarations

In C++, a class declaration introduces the class name into the scope where it is declared and hides any object, function, or other declaration of that name in an enclosing scope. In ANSI/ISO C, an inner scope declaration of a `struct` name does not hide an object or function of that name in an outer scope. For example:

```
double db;
void main ()
{
    struct db // hides double object db in C++
    { char* str; };
    int x = sizeof(db); // size of struct in C++
                        // size of double in ANSI/ISO C
}
```

For more information, see “Scope of Class Names” on page 286. For general information about scope, see “Scope in C++” on page 46.



## const Object Initialization

In C++, const objects must be initialized. In ANSI/ISO C, they can be left uninitialized.

For more information, see “volatile and const Qualifiers” on page 120.

## Definitions

An object declaration is a definition in C++. In ANSI/ISO C, it is a tentative definition. For example:

```
int i;
```

In C++, a global data object must be defined only once. In ANSI/ISO C, a global data object can be declared several times without using the extern keyword.

In C++, multiple definitions for a single variable cause an error. A C compilation unit can contain many identical tentative definitions for a variable.

For more information, see “Chapter 5. Declarations” on page 69.

## Definitions within Return or Argument Types

In C++, types may not be defined in return or argument types. ANSI/ISO C allows such definitions. For example, the following declarations produce errors in C++, but are valid declarations in ANSI/ISO C:

```
void print(struct X { int i;} x);      // error in C++
enum count{one, two, three} counter(); // error in C++
```

For more information, see “Function Declarations” on page 174 and “Calling Functions and Passing Arguments” on page 185.

## Enumerator Type

An enumerator has the same type as its enumeration in C++. In ANSI/ISO C, an enumeration has type int.

For more information on enumerators, see “Enumerations” on page 90.

## Enumeration Type

The assignment to an object of enumeration type with a value that is not of that enumeration type produces an error in C++. In ANSI/ISO C, an object of enumeration type can be assigned values of any integral type.

For more information, see “Enumerations” on page 90.

## Function Declarations

In C++, all declarations of a function must match the unique definition of a function. ANSI/ISO C has no such restriction.

For more information, see “Function Declarations” on page 174.

## Functions with an Empty Argument List

Consider the following function declaration:

```
int f();
```

In C++, this function declaration means that the function takes no arguments. In ANSI/ISO C, it could take any number of arguments, of any type.

For more information, see “Function Declarations” on page 174.

## Global Constant Linkage

In C++, an object declared `const` has internal linkage, unless it has previously been given external linkage. In ANSI/ISO C, it has external linkage.

For more information, see “Program Linkage” on page 37.

## Jump Statements

C++ does not allow you to jump over declarations containing initializations. ANSI/ISO C does allow you to use jump statements for this purpose.

For more information, see “Initializers” on page 127.

## Keywords

C++ contains some additional keywords not found in ANSI/ISO C. C programs that use these keywords as identifiers are not valid C++ programs:

*Table 13. C++ Keywords*

<code>asm</code>	<code>inline</code>	<code>public</code>	<code>virtual</code>
<code>catch</code>	<code>new</code>	<code>template</code>	<code>wchar_t</code>
<code>class</code>	<code>operator</code>	<code>this</code>	
<code>delete</code>	<code>friend</code>	<code>private</code>	
<code>protected</code>	<code>throw</code>	<code>try</code>	

For more information, see “Keywords” on page 57.

## main() Recursion

In C++, `main()` cannot be called recursively and cannot have its address taken. ANSI/ISO C allows recursive calls and allows pointers to hold the address of `main()`.

For more information, see “The `main()` Function” on page 184.

## Names of Nested Classes

In C++, the name of a nested class is local to its enclosing class. In ANSI/ISO C, the name of the nested structure belongs to the same scope as the name of the outermost enclosing structure.

For more information, see “Nested Classes” on page 287.

## Pointers to void

C++ allows void pointers to be assigned only to other void pointers. In ANSI/ISO C, a pointer to void can be assigned to a pointer of any other type without an explicit cast.

For more information, see “void Type” on page 99 and “Pointers” on page 94.

## Prototype Declarations

C++ requires full prototype declarations. ANSI/ISO C allows nonprototyped functions.

For more information, see “Function Declarator” on page 180.

## Return without Declared Value

In C++, a return (either explicit or implicit) from `main()` that is declared to return a value results in an error if no value is returned. A return (either explicit or implicit) from all other functions that is declared to return a value *must* return a value. In ANSI/ISO C, a function that is declared to return a value can return with no value, with unspecified results.

For more information, see “Function Return Values” on page 192.

## \_\_STDC\_\_ Macro

The predefined macro variable `__STDC__` is not defined for C++. It has the integer value 0 when it is used a `#if` statement, indicating that the C++ language is not a proper superset of C, and that the compiler does not conform to ANSI/ISO C. In ANSI/ISO C, `__STDC__` has the integer value 1.

For more information on macros, see “Predefined Macro Names” on page 229.

## typedefs in Class Declarations

In C++, a typedef name may not be redefined in a class declaration after being used in the declaration. ANSI/ISO C allows such a declaration. For example:

```
void main ()
{
    typedef double db;
    struct st
    {
        db x;
        double db; // error in C++, valid in ANSI/ISO C
    };
}
```

For more information, see “typedef” on page 84.

### Interactions with Other Products

You cannot write a C++ program that includes interfaces to Cross-System Product (CSP). However, you can write a C program to access CSP and call the C program from a C++ program.

In general, application libraries that provide C interfaces may not support applications written in C++ if their header files do not conform to C++ syntax.

Not all OS/390 C pragmas are supported by OS/390 C++. If you have any OS/390 C pragmas in C source to be compiled with C++, you should add conditional compilation directives around the pragmas that are not recognized for C++.

For example, the following directive is not supported in OS/390 C++, and would generate an error message because the C++ compiler interprets the K as an integer constants suffix:

```
#pragma runopts(stack(100K))
```

To avoid the error, place conditional compilation directives around the unsupported pragma:

```
#ifndef __cplusplus
#   pragma runopts(stack(100K))
#endif
```

---

## Appendix B. Common Usage C Language Level

The X/Open Portability Guide (XPG) Issue 3 describes a C language definition referred to as Common Usage C. This language definition is roughly equivalent to K&R C, and differs from the ANSI/ISO C language definition. It is based on various C implementations that predate the ANSI/ISO standard.

Common Usage C is supported with the `LANGVLV1(COMMONC)` compiler option or the `#pragma langlvl(commonc)` directive. These cause the compiler to accept C source code containing Common Usage C constructs.

Many of the Common Usage C constructs are already supported by `#pragma langlvl(extended)`. The following language elements are different from those accepted by `#pragma langlvl(extended)`.

- Standard integral promotions preserve sign. For example, unsigned char or unsigned short are promoted to unsigned int. This is functionally equivalent to specifying the `UPCONV` compiler option.
- Trigraphs are not processed in string or character literals. For example, consider the following source line:

```
??=define STR "??= not processed"
```

The above line gets preprocessed to:

```
#define STR "??= not processed"
```

- The `sizeof` operator is permitted on bitfields. The result is the size of an unsigned int (4).
- Bitfields other than type `int` are permitted. The compiler issues a warning and changes the type to unsigned int.
- Macro parameters found within single or double quotation marks are expanded. For example, consider the following source lines:

```
#define STR(AAA) "String is: AAA"  
#define ST STR(BBB)
```

The above lines are preprocessed to:

```
#define STR(AAA) "String is: AAA"  
#define ST "String is: BBB"
```

- Macros can be redefined without first being undefined (that is, without an intervening `#undef`). An informational message is issued saying that the second definition is used.
- The empty comment (`/**/`) in a function-like macro is equivalent to the ANSI/ISO token concatenation operator `##`.

The `LANGVLV1` compiler option is described in the *OS/390 C/C++ User's Guide*. `#pragma langlvl` is described in "langlvl" on page 259.



---

## Appendix C. Conforming to POSIX 1003.1

The implementation resulting from the combination of OS/390 UNIX and the OS/390 Language Environment conforms to the ISO/IEC 9945-1:1990/IEEE POSIX 1003.1-1990 standard. POSIX stands for Portable Operating System Interface.

See the *OpenEdition POSIX.1 Conformance Document for POSIX on MVS/ESA: IEEE Standard 1003.1-1990*, GC23-3011, for a description of how the OS/390 UNIX implementation meets the criteria.





---

## Appendix D. Conforming to ANSI/ISO Standards

This appendix describes changes made to the OS/390 C/C++ Compiler and Library for conformance to the *American National Standard for Information Systems - Programming Language C* standard. It also describes implementation-defined behavior of the OS/390 C/C++ compiler which is not defined by ANSI/ISO.

---

### Implementation-Defined Behavior

The following sections describe how the OS/390 C compiler defines some of the implementation-defined behavior from the ANSI/ISO C Standard.

- “Identifiers”
- “Characters” on page 412
- “String Conversion” on page 413
- “Integers” on page 413
- “Floating-Point” on page 413
- “Arrays and Pointers” on page 414
- “Registers” on page 414
- “Structures, Unions, Enumerations, Bit Fields” on page 414
- “Declarators” on page 415
- “Statements” on page 415
- “Preprocessing Directives” on page 415
- “Library Functions” on page 416
- “Error Handling” on page 416
- “Signals” on page 417
- “Translation Limits” on page 417
- “Streams, Records, and Files” on page 418
- “Memory Management” on page 419
- “Environment” on page 419
- “Localization” on page 420
- “Time” on page 420

### Identifiers

The number of significant characters in an identifier with no external linkage:

- 1024

The number of significant characters in an identifier with external linkage:

- 1024 with the compile-time option `LONGNAME` specified
- 8 otherwise

The C++ compiler truncates external identifiers without C++ linkage after 8 characters if the `NOLONGNAME` compiler option or `#pragma` is in effect.

## Implementation-Defined Behavior

Case sensitivity of external identifiers:

- The linkage editor accepts all external names up to 8 characters, and may not be case sensitive. The binder accepts all external names up to 1024 characters, and is optionally case sensitive. The linkage editor accepts all external names up to 8 characters, and may not be case sensitive, depending on whether you use the NOLONGNAME compiler option or #pragma. When using the OS/390 C compiler with the NOLONGNAME option, all external names are truncated to 8 characters. As an aid to portability, identifiers that differ only in case after truncation are flagged as an error.

## Characters

Source and execution characters which are not specified by the ANSI/ISO standard:

- The caret (^) character in ASCII (bitwise exclusive OR symbol) or the equivalent not (~) character in EBCDIC.
- The vertical broken line (|) character in ASCII which may be represented by the vertical line (I) character on EBCDIC systems.

Shift states used for the encoding of multibyte characters:

- The shift states are indicated with the SHIFTOUT (hex value x0E) characters and SHIFTIN (hex value x0F). Refer to the *OS/390 C/C++ Run-Time Library Reference* for more information on wide character strings.

The number of bits that represent a character:

- 8 bits

The mapping of members of the source character set (characters and strings) to the execution character set:

- The same code page is used for the source and execution character set.

The value of an integer character constant that contains a character/escape sequence not represented in the basic execution character set:

- A warning is issued for an unknown character/escape sequence and the char is assigned the character following the back slash.

The value of a wide character constant that contains a character/escape sequence not represented in the extended execution character set:

- A warning is issued for the unknown character/escape sequence and the wchar\_t is assigned the wide character following the back slash.

The value of an integer character constant that contains more than one character:

- The lowest four bytes represent the character constant.

The value of a wide character constant that contains more than one multibyte character:

- The lowest four bytes of the multibyte characters are converted to represent the wide character constant.

Equivalent type of char: signed char, unsigned char, or user-defined:

- The default for char is unsigned

Is each sequence of white-space characters (excluding the new-line) retained or replaced by one space character?

- Any spaces or comments in your source program will be interpreted as one space.

## String Conversion

Additional implementation-defined sequence forms that can be accepted by `strtod()`, `strtol()` and `strtoul()` functions in other than the C locale:

- None

## Integers

Table 14. *Integers*

Type	Amount of Storage	Range (in <code>limits.h</code> )
signed short	2 bytes	-32,768 to 32,767
unsigned short	2 bytes	0 to 65,535
signed int	4 bytes	-2,147,483,647 minus 1 to 2,147,483,647
unsigned int	4 bytes	0 to 4,294,967,295
signed long	4 bytes	-2,147,483,647 minus 1 to 2,147,483,647
unsigned long	4 bytes	0 to 4,294,967,295
signed long long	8 bytes	-9,223,372,036,854,775,807 minus 1 to 9,223,372,036,854,775,807
unsigned long long	4 bytes	0 to 18,446,744,073,709,551,615

The result of converting an integer to a signed char:

- The lowest 1 byte of the integer is used to represent the char. See the *OS/390 C/C++ Run-Time Library Reference* for more information on data conversions.

The result of converting an integer to a shorter signed integer:

- The lowest 2 bytes of the integer are used to represent the short int.

The result of converting an unsigned integer to a signed integer of equal length, if the value cannot be represented:

- The bit pattern is preserved and the sign bit has no significance.

The result of bitwise operations (`|`, `&`, `^`) on signed int:

- The representation is treated as a bit pattern and 2's complement arithmetic is performed.

The sign of the remainder of integer division if either operand is negative:

- The remainder is negative if exactly one operand is negative.

The result of a right shift of a negative-valued signed integral type:

- The result is sign extended and the sign is propagated.

## Floating-Point

Table 15. *Floating Point*

Type	Amount of Storage	Range (approximate)	
		IBM S/390 Hexadecimal Format	IEEE Binary Format
float	4 bytes	$5.5 \times 10^{-79}$ to $7.2 \times 10^{75}$	$1.2 \times 10^{-38}$ to $3.4 \times 10^{38}$
double	8 bytes	$5.5 \times 10^{-79}$ to $7.2 \times 10^{75}$	$2.2 \times 10^{-308}$ to $1.8 \times 10^{308}$
long double	16 bytes	$5.5 \times 10^{-79}$ to $7.2 \times 10^{75}$	$3.4 \times 10^{-4932}$ to $1.2 \times 10^{4932}$

## Implementation-Defined Behavior

The following is the direction of truncation (or rounding) when you convert an integer number to an IBM S/390 hexadecimal floating-point number, or to an IEEE binary floating-point number:

- IBM S/390 hexadecimal format:

When the floating-point cannot exactly represent the original value, the value is truncated.

When a floating-point number is converted to a narrower floating-point number, the floating-point number is truncated.

- IEEE binary format:

The rounding direction is determined by the `ROUND` compiler option. The `ROUND` option only affects the rounding of floating-point values that the OS/390 C/C++ compiler can evaluate at compile time. It has no effect on rounding at run time.

## Arrays and Pointers

The type of `size_t`:

- `unsigned int`

The type of `ptrdiff_t`:

- `int`

The result of casting a pointer to an integer:

- The bit patterns are preserved.

The result of casting an integer to a pointer:

- The bit patterns are preserved.

## Registers

The effect of the register storage class specifier on the storage of objects in registers:

- If there is a register available, the object is stored in a register.

## Structures, Unions, Enumerations, Bit Fields

The result when a member of a union object is accessed using a member of a different type:

- The result is undefined.

The alignment/padding of structure members:

- If the structure is not packed, then padding is added to align the structure members on their natural boundaries. If the structure is packed, no padding is added. Refer to “C/C++ Data Mapping” on page 129 for more information on C data mapping.

The padding at the end of structure/union:

- Padding is added to end the structure on its natural boundary. The alignment of the struct or union is that of its strictest member. Refer to “C/C++ Data Mapping” on page 129 for more information on C data mapping.

The type of an `int` bit field (signed `int`, unsigned `int`, user defined):

- The default is unsigned.

The order of allocation of bit fields within an int:

- Bit fields are allocated from low memory to high memory. For example, 0x12345678 would be stored with byte 0 containing 0x12, and byte 3 containing 0x78.

The rule for bit fields crossing a storage unit boundary:

- Bit fields can cross storage unit boundaries.

The integral type that represents the values of an enumeration type:

- Enumerations can have the type char, short, or long and be either signed or unsigned depending on their smallest and largest values.

## Declarators

The maximum number of declarators (pointer, array, function) that can modify an arithmetic, structure, or union type:

- The only constraint is the availability of system resources.

## Statements

The maximum number of case values in a switch statement:

- Because the case values must be integers and cannot be duplicated, the limit is INT\_MAX.

## Preprocessing Directives

Does the value of a single-character constant in a constant expression that controls conditional inclusion match the value of the character constant in the execution character set?

- Yes

Can such a constant have a negative value?

- Yes

The method of searching include source files (< >):

- See the *OS/390 C/C++ User's Guide*.

Is the search for quoted source file names supported ("...")?

- User include files can be specified in double quotes. See the *OS/390 C/C++ User's Guide*.

The mapping between the name specified in the include directive and the external source file name:

- See the *OS/390 C/C++ User's Guide*.

The behavior of each pragma directive:

- See "Pragma Directives (#pragma)" on page 243.

The definitions of \_\_DATE\_\_ and \_\_TIME\_\_ when date and time of translation is not available:

- For OS/390 C/C++, the date and time of translation are always available.

## Implementation-Defined Behavior

### Library Functions

The definition of NULL macro:

- NULL is defined to be a `((void *)0)`.

The format of diagnostic printed by the assert macro, and the termination behavior (abort behavior):

- When assert is executed, if the expression is false, the diagnostic message written by the assert macro has the format:

Assertion failed: *expression*, file *filename*, line *line-number*

Set of characters tested by the isxxxx functions:

- To create a table of the characters set up by the ctype functions use the program in the following example.

#### CBC3RABG

*/\* this example prints out ctest characters \*/*

```
#include <stdio.h>
#include <ctype.h>

int main(void)
{
    int ch;

    for (ch = 0; ch <= 0xff; ch++)
    {
        printf("#04X ", ch);
        printf("%3s ", isalnum(ch) ? "AN" : " ");
        printf("%2s ", isalpha(ch) ? "A" : " ");
        printf("%2s", iscntrl(ch) ? "C" : " ");
        printf("%2s", isdigit(ch) ? "D" : " ");
        printf("%2s", isgraph(ch) ? "G" : " ");
        printf("%2s", islower(ch) ? "L" : " ");
        printf("%c", isprint(ch) ? ch : ' ');
        printf("%3s", ispunct(ch) ? "PU" : " ");
        printf("%2s", isspace(ch) ? "S" : " ");
        printf("%3s", isprint(ch) ? "PR" : " ");
        printf("%2s", isupper(ch) ? "U" : " ");
        printf("%2s", isxdigit(ch) ? "X" : " ");

        putchar('\n');
    }
}
```

The result of calling `fmod()` function with the second argument zero (return zero, domain error):

- `fmod()` returns a 0.

### Error Handling

The format of the message generated by the `perror()` and `strerror()` functions:

- See the *OS/390 Language Environment Debugging Guide and Run-Time Messages* for the messages emitted for `perror()` and `strerror()`.

**Note:** `errno` is not emitted with the message.

How diagnostic messages are recognized:

## Implementation-Defined Behavior

- Refer to the *OS/390 C/C++ User's Guide* and the *OS/390 Language Environment Debugging Guide and Run-Time Messages* for the lists of OS/390 C/C++ messages provided.

The different classes of messages:

- In general, messages are classified as shown by the following table.

Type of Message	Numeric Severity Level	Return Code
Information	00	0
Warning	10	4
Error	30	12
Severe error	> 30	16

How the level of diagnostics can be controlled:

- Use the compile-time option FLAG to control the level of diagnostics. There is also a compile-time option CHECKOUT which provides programming style diagnostics to aid you in determining possible programming errors.

## Signals

The set of signals for the `signal()` function:

- See the *OS/390 C/C++ User's Guide*.

The parameters and the usage of each signal recognized by the `signal()` function:

- See the *OS/390 C/C++ Programming Guide*.

The default handling and the handling at program start-up for each signal recognized by `signal()` function:

- SIG\_DFL is the default signal. See the *OS/390 C/C++ Programming Guide* for more information on signal handling.

The signal blocking performed if the equivalent of `signal(sig, SIG_DFL)` is not executed at the beginning of signal handler:

- See the *OS/390 C/C++ Programming Guide*.

Is the default handling reset if a SIGKILL is received by a signal handler?

- Whenever you enter the signal handler, SIG\_DFL becomes the default.

## Translation Limits

System-determined means that the limit is determined by your system resources.

*Table 16. Translation Limits*

Nesting levels of:

• Compound statements	• System-determined
• Iteration control	• System-determined
• Selection control	• System-determined
• Conditional inclusion	• System-determined
• Parenthesized declarators	• System-determined
• Parenthesized expression	• System-determined
Number of pointer, array and function declarators modifying an arithmetic a structure, a union, and incomplete type declaration	• System-determined

## Implementation-Defined Behavior

Table 16. Translation Limits (continued)

Significant initial characters in:

- |   |        |
|---|--------|
| • Internal identifiers                              | • 1024 |
| • Macro names                                       | • 1024 |
| • C external identifiers ( <i>without</i> LONGNAME) | • 8    |
| • C external identifiers ( <i>with</i> LONGNAME)    | • 1024 |
| • C++ external identifiers                          | • 1024 |

Number of:

- |   |                     |
|---|---------------------|
| • External identifiers in a translation unit                      | • System-determined |
| • Identifiers with block scope in one block                       | • System-determined |
| • Macro identifiers simultaneously declared in a translation unit | • System-determined |
| • Parameters in one function definition                           | • System-determined |
| • Arguments in a function call                                    | • System-determined |
| • Parameters in a macro definition                                | • System-determined |
| • Parameters in a macro invocation                                | • System-determined |
| • Characters in a logical source line                             | • 32760 under MVS   |
| • Characters in a string literal                                  | • 32K minus 1       |
| • Bytes in an object  | • LONG_MAX (See 1)  |
| • Nested include files  | • SHRT_MAX          |
| • Enumeration constants in an enumeration                         | • System-determined |
| • Levels in nested structure or union                             | • System-determined |

**Note:**

- 1      LONG\_MAX is the limit for automatic variables only. For all other variables, the limit is 16 Megabytes.

---

## Streams, Records, and Files

Does the last line of a text stream require a terminating new-line character?

- No, the last new-line character is defaulted.

Do space characters, that are written out to a text stream immediately before a new-line character, appear when read?

- White space characters written to fixed record format text streams before a new-line do not appear when read. However, white space characters written to variable record format text streams before a new-line character appear when read.

The number of null characters that can be appended to the end of the binary stream:

- No limit

Where is the file position indicator of an append-mode stream initially positioned?

- The file position indicator is positioned at the end of the file.

Does a write on a text stream cause the associated file to be truncated?

- Yes

Does a file of zero length exist?

- Yes

The rules for composing a valid file name:

- See the *OS/390 C/C++ Programming Guide*.

Can the same file be simultaneously opened multiple times?



## Implementation-Defined Behavior

- For reading, the file can be opened multiple times; for writing/appending, the file can be opened once. Once a file is opened for reading, it cannot be opened for writing.

The effect of the `remove()` function on an open file:

- `remove()` fails.

The effect of the `rename()` function on file to a name that exists prior to the function call:

- The `rename()` fails.

Are temporary files removed if the program terminates abnormally?

- Yes

The effect of calling the `tmpnam()` function more than `TMP_MAX` times:

- `tmpnam()` fails and returns `NULL`.

The output of `%p` conversion in the `fprintf()` function:

- It is equivalent to `%X`.

The input of `%p` conversion in the `fscanf()` function:

- The value is treated as an integer.

The interpretation of a `-` character that is neither the first nor the last in the scanlist for `%[` conversion in the `fscanf()` function:

- The sequence of characters on either side of the `-` are used as delimiters. For example, `%[a-f]` will read in characters between `'a'` and `'f'`.

The value of `errno` on failure of `fgetpos()` and `ftell()` functions:

- This depends on the failure. For a list of the messages associated with `errno`, see the *OS/390 Language Environment Debugging Guide and Run-Time Messages*.

## Memory Management

The behavior of `calloc()`, `malloc()` and `realloc()` functions if the size requested is zero:

- Nothing is performed for `calloc()` and `malloc()`; `realloc()` frees the storage.

## Environment

The arguments of `main` function:

- You can pass arguments to `main` through `argv` and `argc`.

What happens with open files when the `abort()` function is called?

- The files are closed.

What is returned to the host environment when the `abort()` function is called?

- The return code of 2000 is returned.

The form of successful termination when the `exit` function is called with argument zero or `EXIT_SUCCESS`:

- All files are closed, all storage is released and the return code of 0 is returned.

The form of unsuccessful termination when the `exit` function is called with argument `EXIT_FAILURE`:

## Implementation-Defined Behavior

- All files are closed, all storage is released and the return code of EXIT\_FAILURE is returned.

What status is returned by the exit function if the argument is other than zero, EXIT\_FAILURE and EXIT\_SUCCESS?

- The return code 4096 is returned.

The set of environmental names:

- There are no environmental names.

The method of altering the environment list obtained by a call to the getenv() function:

- See how to execute a command in the *OS/390 C/C++ Run-Time Library Reference*.

The format and a mode of execution of a string on a call to the system() function:

- See the *OS/390 C/C++ Run-Time Library Reference*.

## Localization

The environment specified by the "" locale on a setlocale() call:

- EDC\$SAAC

The supported locales:

- See the *OS/390 C/C++ Programming Guide*.

## Time

The local time zone and Daylight Saving Time:

- This is specified in the locale.

The era for the clock() function:

- The era starts when the program is started by either a call from the operating system, or a call to system(). Under TSO, the era starts when you log on to the system. To measure the time spent in a program, call the clock() function at the start of the program, and subtract its return value from the value returned by subsequent calls to clock().

---

## Glossary

This glossary defines terms and abbreviations that are used in this book. Included are terms and definitions from the following sources:

- *American National Standard Dictionary for Information Systems*, ANSI/ISO X3.172-1990, copyright 1990 by the American National Standards Institute (ANSI/ISO). Copies may be purchased from the American National Standards Institute, 1430 Broadway, New York, New York 10018. Such definitions are indicated by the symbol *ANSI/ISO* after the definition.
- *IBM Dictionary of Computing*, SC20-1699. These definitions are indicated by the registered trademark *IBM* after the definition.
- *X/Open CAE Specification, Commands and Utilities, Issue 4*, July, 1992. These definitions are indicated by the symbol *X/Open* after the definition.
- *ISO/IEC 9945-1:1990/IEEE POSIX 1003.1-1990*. These definitions are indicated by the symbol *ISO.1* after the definition.
- *The Information Technology Vocabulary*, developed by Subcommittee 1, Joint Technical Committee 1, of the International Organization for Standardization and the International Electrotechnical Commission (ISO/IEC JTC1/SC1). Definitions of published parts of this vocabulary are identified by the symbol *ISO-JTC1* after the definition; definitions taken from draft international standards, committee drafts, and working papers being developed by ISO/IEC JTC1/SC1 are identified by the symbol *ISO Draft* after the definition, indicating that final agreement has not yet been reached among the participating National Bodies of SC1.

## A

**abstract class.** (1) A class with at least one pure virtual function that is used as a base class for other classes. The abstract class represents a concept; classes derived from it represent implementations of the concept. You cannot have a direct object of an abstract class. See also *base class*. (2) A class that allows polymorphism. There can be no objects of an abstract class; they are only used to derive new classes.

**abstract code unit.** See *ACU*.

**abstract data type.** A mathematical model that includes a structure for storing data and operations that can be performed on that data. Common abstract data types include sets, trees, and heaps.

**abstraction (data).** A data type with a private representation and a public set of operations (functions or operators) which restrict access to that data type to that set of operations. The C++ language uses the concept of classes to implement data abstraction.

**access.** An attribute that determines whether or not a class member is accessible in an expression or declaration.

**access declaration.** A declaration used to restore access to members of a base class.

**access mode.** (1) A technique that is used to obtain a particular logical record from, or to place a particular logical record into, a file assigned to a mass storage device. *ANSI/ISO*. (2) The manner in which files are referred to by a computer. Access can be sequential (records are referred to one after another in the order in which they appear on the file), access can be random (the individual records can be referred to in a nonsequential manner), or access can be dynamic (records can be accessed sequentially or randomly, depending on the form of the input/output request). *IBM*. (3) A particular form of access permitted to a file. *X/Open*.

**access resolution.** The process by which the accessibility of a particular class member is determined.

**access specifier.** One of the C++ keywords: *public*, *private*, and *protected*, used to define the access to a member.

**ACU (abstract code unit).** A measurement used by the OS/390 C/C++ compiler for judging the size of a function. The number of ACUs that comprise a function is proportional to its size and complexity.

**addressing mode.** See *AMODE*.

**address space.** (1) The range of addresses available to a computer program. *ANSI/ISO*. (2) The complete range of addresses that are available to a programmer. See also *virtual address space*. (3) The area of virtual storage available for a particular job. (4) The memory locations that can be referenced by a process. *X/Open*. *ISO.1*.

**aggregate.** (1) An array or a structure. (2) A compile-time option to show the layout of a structure or union in the listing. (3) An array or a class object with no private or protected members, no constructors, no base classes, and no virtual functions. (4) In

programming languages, a structured collection of data items that form a data type. *ISO-JTC1*.

**alert.** (1) A message sent to a management services focal point in a network to identify a problem or an impending problem. *IBM*. (2) To cause the user's terminal to give some audible or visual indication that an error or some other event has occurred. When the standard output is directed to a terminal device, the method for alerting the terminal user is unspecified. When the standard output is not directed to a terminal device, the alert is accomplished by writing the alert character to standard output (unless the utility description indicates that the use of standard output produces undefined results in this case). *X/Open*.

**alert character.** A character that in the output stream should cause a terminal to alert its user via a visual or audible notification. The alert character is the character designated by a '\a' in the C and C++ languages. It is unspecified whether this character is the exact sequence transmitted to an output device by the system to accomplish the alert function. *X/Open*.

This character is named <alert> in the portable character set.

**alias.** (1) An alternate label; for example, a label and one or more aliases may be used to refer to the same data element or point in a computer program. *ANSI/ISO*. (2) An alternate name for a member of a partitioned data set. *IBM*. (3) An alternate name used for a network. Synonymous with nickname. *IBM*.

**alias name.** (1) A word consisting solely of underscores, digits, and alphabets from the portable file name character set, and any of the following characters: ! % , @. Implementations may allow other characters within alias names as an extension. *X/Open*. (2) An alternate name. *IBM*. (3) A name that is defined in one network to represent a logical unit name in another interconnected network. The alias name does not have to be the same as the real name; if these names are not the same; translation is required. *IBM*.

**alignment.** The storing of data in relation to certain machine-dependent boundaries. *IBM*.

**alternate code point.** A syntactic code point that permits a substitute code point to be used. For example, the left brace (l) can be represented by X'B0' and also by X'C0'.

**American National Standard Code for Information Interchange (ASCII).** The standard code, using a coded character set consisting of 7-bit coded characters (8 bits including parity check), that is used for information interchange among data processing systems, data communication systems, and associated equipment. The ASCII set consists of control characters and graphic characters. *IBM*.

**Note:** IBM has defined an extension to ASCII code (characters 128–255).

**American National Standards Institute (ANSI/ISO).** An organization consisting of producers, consumers, and general interest groups, that establishes the procedures by which accredited organizations create and maintain voluntary industry standards in the United States. *ANSI/ISO*.

**AMODE (addressing mode).** In MVS, a program attribute that refers to the address length that a program is prepared to handle upon entry. In MVS, addresses may be 24 or 31 bits in length. *IBM*.

**angle brackets.** The characters < (left angle bracket) and > (right angle bracket). When used in the phrase "enclosed in angle brackets", the symbol < immediately precedes the object to be enclosed, and > immediately follows it. When describing these characters in the portable character set, the names <less-than-sign> and <greater-than-sign> are used. *X/Open*.

**anonymous union.** A union that is declared within a structure or class and does not have a name. It must not be followed by a declarator.

**ANSI/ISO.** See *American National Standards Institute*.

**API (application program interface).** A functional interface supplied by the operating system or by a separately orderable licensed program that allows an application program written in a high-level language to use specific data or functions of the operating system or the licensed program. *IBM*.

**application.** (1) The use to which an information processing system is put; for example, a payroll application, an airline reservation application, a network application. *IBM*. (2) A collection of software components used to perform specific types of user-oriented work on a computer. *IBM*.

**application generator.** An application development tool that creates applications, application components (panels, data, databases, logic, interfaces to system services), or complete application systems from design specifications.

**application program.** A program written for or by a user that applies to the user's work, such as a program that does inventory control or payroll. *IBM*.

**archive libraries.** The archive library file, when created for application program object files, has a special symbol table for members that are object files.

**argument.** (1) A parameter passed between a calling program and a called program. *IBM*. (2) In a function call, an expression that represents a value that the calling function passes to the function specified in the call. Also called *parameter*. (3) In the shell, a parameter passed to a utility as the equivalent of a single string in

the *argv* array created by one of the *exec* functions. An argument is one of the options, option-arguments, or operands following the command name. *X/Open*.

**argument declaration.** See *parameter declaration*.

**arithmetic object.** (1) An integral object, a bit field, or floating-point object. (2) A real object or objects having the type float, double, or long double.

**array.** In programming languages, an aggregate that consists of data objects with identical attributes, each of which may be uniquely referenced by subscripting. *IBM*.

**array element.** A data item in an array. *IBM*.

**ASCII.** See *American National Standard Code for Information Interchange*.

**Assembler H.** An IBM licensed program. Translates symbolic assembler language into binary machine language.

**assembler language.** A source language that includes symbolic language statements in which there is a one-to-one correspondence with the instruction formats and data formats of the computer. *IBM*.

**assembler user exit.** In the OS/390 Language Environment a routine to tailor the characteristics of an enclave prior to its establishment.

**assignment expression.** An expression that assigns the value of the right operand expression to the left operand variable and has as its value the value of the right operand. *IBM*.

**atexit list.** A list of actions specified in the OS/390 C/C++ *atexit()* function that occur at normal program termination.

**auto storage class specifier.** A specifier that enables the programmer to define a variable with automatic storage; its scope restricted to the current block.

**automatic call library.** Contains modules that are used as secondary input to the prelinker or the binder to resolve external symbols left undefined after all the primary input has been processed.

The automatic call library can contain:

- Object modules, with or without binder control statements
- Load modules
- OS/390 C/C++ run-time routines (SCEELKED)

**automatic library call.** The process in which control sections are processed by the binder or loader to resolve references to members of partitioned data sets. *IBM*.

**automatic storage.** Storage that is allocated on entry to a routine or block and is freed on the subsequent return. Sometimes referred to as *stack storage* or *dynamic storage*.

## B

**background process.** (1) A process that does not require operator intervention but can be run by the computer while the workstation is used to do other work. *IBM*. (2) A mode of program execution in which the shell does not wait for program completion before prompting the user for another command. *IBM*. (3) A process that is a member of a background process group. *X/Open*. *ISO.1*.

**background process group.** Any process group, other than a foreground process group, that is a member of a session that has established a connection with a controlling terminal. *X/Open*. *ISO.1*.

**backslash.** The character \. This character is named <backslash> in the portable character set.

**base class.** A class from which other classes are derived. A base class may itself be derived from another base class. See also *abstract class*.

**based on.** The use of existing classes for implementing new classes.

**binary expression.** An expression containing two operands and one operator.

**binary stream.** (1) An ordered sequence of untranslated characters. (2) A sequence of characters that corresponds on a one-to-one basis with the characters in the file. No character translation is performed on binary streams. *IBM*.

**bind.** To combine one or more control sections or program modules into a single program module, resolving references between them, or to assign virtual storage addresses to external symbols.

**binder.** The DFSMS/MVS program that processes the output of language translators and compilers into an executable program (load module or program object). It replaces the linkage editor and batch loader in the MVS/ESA or OS/390 operating system.

**bit field.** A member of a structure or union that contains a specified number of bits. *IBM*.

**bitwise operator.** An operator that manipulates the value of an object at the bit level.

**blank character.** (1) A graphic representation of the space character. *ANSI/ISO*. (2) A character that represents an empty position in a graphic character string. *ISO Draft*. (3) One of the characters that belong to the *blank* character class as defined via the



LC\_CTYPE category in the current locale. In the POSIX locale, a blank character is either a tab or a space character. *X/Open*.

**block.** (1) In programming languages, a compound statement that coincides with the scope of at least one of the declarations contained within it. A block may also specify storage allocation or segment programs for other purposes. *ISO-JTC1*. (2) A string of data elements recorded or transmitted as a unit. The elements may be characters, words or physical records. *ISO Draft*. (3) The unit of data transmitted to and from a device. Each block contains one record, part of a record, or several records.

**block statement.** In the C or C++ languages, a group of data definitions, declarations, and statements appearing between a left brace and a right brace that are processed as a unit. The block statement is considered to be a single C or C++ statement. *IBM*.

**boundary alignment.** The position in main storage of a fixed-length field, such as a halfword or doubleword, on a byte-level boundary for that unit of information. *IBM*.

**braces.** The characters { (left brace) and } (right brace), also known as *curly braces*. When used in the phrase “enclosed in (curly) braces” the symbol { immediately precedes the object to be enclosed, and } immediately follows it. When describing these characters in the portable character set, the names <left-brace> and <right-brace> are used. *X/Open*.

**brackets.** The characters [ (left bracket) and ] (right bracket), also known as *square brackets*. When used in the phrase *enclosed in (square) brackets* the symbol [ immediately precedes the object to be enclosed, and ] immediately follows it. When describing these characters in the portable character set, the names <left-bracket> and <right-bracket> are used. *X/Open*.

**break statement.** A C or C++ control statement that contains the keyword “break” and a semicolon. *IBM*. 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.

**built-in.** (1) A function that the compiler will automatically inline instead of making the function call, unless the programmer specifies not to inline. (2) In programming languages, pertaining to a language object that is declared by the definition of the programming language; for example, the built-in function SIN in PL/I, the predefined data type INTEGER in FORTRAN. *ISO-JTC1*. Synonymous with *predefined*. *IBM*.

**byte-oriented stream.** See *orientation of a stream*.

## C

**C library.** A system library that contains common C language subroutines for file access, string operators, character operations, memory allocation, and other functions. *IBM*.

**C or C++ language statement.** A C or C++ language statement contains zero or more expressions. A block statement begins with a { (left brace) symbol, ends with a } (right brace) symbol, and contains any number of statements.

All C or C++ language statements, except block statements, end with a ; (semicolon) symbol.

**c89 utility.** A utility used to compile and bind an OS/390 UNIX application program from the OS/390 shell.

**C++ class library.** A collection of C++ classes.

**C++ library.** A system library that contains common C++ language subroutines for file access, memory allocation, and other functions.

**callable services.** A set of services that can be invoked by a OS/390 Language Environment-conforming high level language using the conventional OS/390 Language Environment-defined call interface, and usable by all programs sharing the OS/390 Language Environment conventions.

Use of these services helps to decrease an application's dependence on the specific form and content of the services delivered by any single operating system.

**call chain.** A trace of all active routines and subroutines.

**caller.** A routine that calls another routine.

**cancelability point.** A specific point within the current thread that is enabled to solicit cancel requests. This is accomplished using the `pthread_testintr()` function.

**carriage-return character.** A character that in the output stream indicates that printing should start at the beginning of the same physical line in which the carriage-return character occurred. The carriage-return is the character designated by '\r' in the C and C++ languages. It is unspecified whether this character is the exact sequence transmitted to an output device by the system to accomplish the movement to the beginning of the line. *X/Open*.

**case clause.** In a C or C++ 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.

**cataloged procedures.** A set of control statements placed in a library and retrievable by name. *IBM.*

**catch block.** A block associated with a try block that receives control when an exception matching its argument is thrown.

**char specifier.** A char is a built-in data type. In the C++ language, char, signed char, and unsigned char are all distinct data types.

**character.** (1) A letter, digit, or other symbol that is used as part of the organization, control, or representation of data. A character is often in the form of a spatial arrangement of adjacent or connected strokes. *ANSI/ISO.* (2) A sequence of one or more bytes representing a single graphic symbol or control code. This term corresponds to the ISO C standard term *multibyte character* (multibyte character), where a single-byte character is a special case of the multibyte character. Unlike the usage in the ISO C standard, *character* here has no necessary relationship with storage space, and *byte* is used when storage space is discussed. *X/Open. ISO.1.*

**character array.** An array of type char. *X/Open.*

**character class.** A named set of characters sharing an attribute associated with the name of the class. The classes and the characters that they contain are dependent on the value of the LC\_CTYPE category in the current locale. *X/Open.*

**character constant.** (1) A constant with a character value. *IBM.* (2) A string of any of the characters that can be represented, usually enclosed in apostrophes. *IBM.* (3) In some languages, a character enclosed in apostrophes. *IBM.*

**character set.** (1) A finite set of different characters that is complete for a given purpose; for example, the character set in ISO Standard 646, 7-bit Coded Character Set for Information Processing Interchange. *ISO Draft.* (2) All the valid characters for a programming language or for a computer system. *IBM.* (3) A group of characters used for a specific reason; for example, the set of characters a printer can print. *IBM.* (4) See also *portable character set.*

**character special file.** (1) A special file that provides access to an input or output device. The character interface is used for devices that do not use block I/O. *IBM.* (2) A file that refers to a device. One specific type of character special file is a terminal device file. *X/Open. ISO.1.*

**character string.** A contiguous sequence of characters terminated by and including the first null byte. *X/Open.*

**child.** A node that is subordinate to another node in a tree structure. Only the root node is not a child.

**child enclave.** The *nested enclave* created as a result of certain commands being issued from a *parent enclave*.

**CICS (Customer Information Control System).** Pertaining to an IBM licensed program that enables transactions entered at remote terminals to be processed concurrently by user-written application programs. It includes facilities for building, using, and maintaining databases. *IBM.*

**CICS destination control table.** See *DCT.*

**CICS translator.** A routine that accepts as input an application containing EXEC CICS commands and produces as output an equivalent application in which each CICS command has been translated into the language of the source.

**class.** (1) A C++ aggregate that may contain functions, types, and user-defined operators in addition to data. Classes may be defined hierarchically, allowing one class to be derived from another, and may restrict access to its members. (2) A user-defined data type. A class data type can contain both data representations (data members) and functions (member functions).

**class key.** One of the C++ keywords: class, struct and union.

**class library.** A collection of classes.

**class member operator.** An operator used to access class members through class objects or pointers to class objects. The class member operators are:

. -> .\* ->\*

**class name.** A unique identifier of a class type that becomes a reserved word within its scope.

**class scope.** An indication that a name of a class can be used only in a member function of that class.

**class tag.** Synonym for *class name*.

**class template.** A blueprint describing how a set of related classes can be constructed.

**client program.** A program that uses a class. The program is said to be a *client* of the class.

**CLIST.** A programming language that typically executes a list of TSO commands.

**CLLE (COBOL Load List Entry).** Entry in the load list containing the name of the program and the load address.

**COBCOM.** Control block containing information about a COBOL partition.

**COBOL (common business-oriented language).** A high-level language, based on English, that is primarily used for business applications.

**COBOL Load List Entry.** See *CLLE*.

**COBVEC.** COBOL vector table containing the address of the library routines.

**coded character set.** (1) A set of graphic characters and their code point assignments. The set may contain fewer characters than the total number of possible characters: some code points may be unassigned. *IBM*. (2) A coded set whose elements are single characters; for example, all characters of an alphabet. *ISO Draft*. (3) Loosely, a code. *ANSI/ISO*.

**code element set.** (1) The result of applying a code to all elements of a coded set, for example, all the three-letter international representations of airport names. *ISO Draft*. (2) The result of applying rules that map a numeric code value to each element of a character set. An element of a character set may be related to more than one numeric code value but the reverse is not true. However, for state-dependent encodings the relationship between numeric code values to elements of a character set may be further controlled by state information. The character set may contain fewer elements than the total number of possible numeric code values; that is, some code values may be unassigned. *X/Open*. (3) Synonym for codeset.

**code page.** (1) An assignment of graphic characters and control function meanings to all code points; for example, assignment of characters and meanings to 256 code points for an 8-bit code, assignment of characters and meanings to 128 code points for a 7-bit code. (2) A particular assignment of hexadecimal identifiers to graphic characters.

**code point.** (1) A 1-byte code representing one of 256 potential characters. (2) An identifier in an alert description that represents a short unit of text. The code point is replaced with the text by an alert display program.

**codeset.** Synonym for code element set. *IBM*.

**collating element.** The smallest entity used to determine the logical ordering of character or wide-character strings. A collating element consists of either a single character, or two or more characters collating as a single entity. The value of the *LC\_COLLATE* category in the current locale determines the current set of collating elements. *X/Open*.

**collating sequence.** (1) A specified arrangement used in sequencing. *ISO-JTC1*. *ANSI/ISO*. (2) An ordering assigned to a set of items, such that any two sets in that assigned order can be collated. *ANSI/ISO*. (3) The

relative ordering of collating elements as determined by the setting of the *LC\_COLLATE* category in the current locale. The character order, as defined for the *LC\_COLLATE* category in the current locale, defines the relative order of all collating elements, such that each element occupies a unique position in the order. This is the order used in ranges of characters and collating elements in regular expressions and pattern matching. In addition, the definition of the collating weights of characters and collating elements uses collating elements to represent their respective positions within the collation sequence.

**collation.** The logical ordering of character or wide-character strings according to defined precedence rules. These rules identify a collation sequence between the collating elements, and such additional rules that can be used to order strings consisting of multiple collating elements. *X/Open*.

**collection.** (1) An abstract class without any ordering, element properties, or key properties. All abstract classes are derived from collection. (2) In a general sense, an implementation of an abstract data type for storing elements.

**Collection Class Library.** A set of classes that provide basic functions for collections, and can be used as base classes.

**column position.** A unit of horizontal measure related to characters in a line.

It is assumed that each character in a character set has an intrinsic column width independent of any output device. Each printable character in the portable character set has a column width of one. The standard utilities, when used as described in this document set, assume that all characters have integral column widths. The column width of a character is not necessarily related to the internal representation of the character (numbers of bits or bytes).

The column position of a character in a line is defined as one plus the sum of the column widths of the preceding characters in the line. Column positions are numbered starting from 1. *X/Open*.

**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.

**command.** A request to perform an operation or run a program. When parameters, arguments, flags, or other operands are associated with a command, the resulting character string is a single command.

**command processor parameter list (CPPL).** The format of a TSO parameter list. When a TSO terminal monitor application attaches a command processor,



register 1 contains a pointer to the CPPL, containing addresses required by the command processor.

**COMMAREA.** A communication area made available to applications running under CICS.

**Common Business-Oriented Language.** See *COBOL*.

**common expression elimination.** Duplicated expressions are eliminated by using the result of the previous expression. This includes intermediate expressions within expressions.

**compilation unit.** (1) A portion of a computer program sufficiently complete to be compiled correctly. *IBM*. (2) A single compiled file and all its associated include files. (3) An independently compilable sequence of high-level language statements. Each high-level language product has different rules for what makes up a compilation unit.

**complete class name.** The complete qualification of a nested class name including all enclosing class names.

**Complex Mathematics library.** A C++ class library that provides the facilities to manipulate complex numbers and perform standard mathematical operations on them.

**computational independence.** No data modified by either a main task program or a parallel function is examined or modified by a parallel function that might be running simultaneously.

**concrete class.** A class that implements an abstract data type but does not allow polymorphism.

**condition.** (1) A relational expression that can be evaluated to a value of either true or false. *IBM*. (2) An exception that has been enabled, or recognized, by the OS/390 Language Environment and thus is eligible to activate user and language condition handlers. Any alteration to the normal programmed flow of an application. Conditions can be detected by the hardware/operating system and result in an interrupt. They can also be detected by language-specific generated code or language library code.

**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).

**condition handler.** A user-written condition handler or language-specific condition handler (such as a PL/I ON-unit or OS/390 C/C++ `signal()` function call) invoked by the OS/390 C/C++ *condition manager* to respond to conditions.

**condition manager.** Manages conditions in the common execution environment by invoking various user-written and language-specific *condition handlers*.

**condition token.** In the OS/390 Language Environment, a data type consisting of 12 bytes (96 bits). The condition token contains structured fields that indicate various aspects of a condition including the severity, the associated message number, and information that is specific to a given instance of the condition.

**const.** (1) An attribute of a data object that declares the object cannot be changed. (2) A keyword that allows you to define a variable whose value does not change.

**constant.** (1) In programming languages, a language object that takes only one specific value. *ISO-JTC1*. (2) A data item with a value that does not change. *IBM*.

**constant expression.** An expression having a value that can be determined during compilation and that does not change during the running of the program. *IBM*.

**constant propagation.** An optimization technique where constants used in an expression are combined and new ones are generated. Mode conversions are done to allow some intrinsic functions to be evaluated at compile time.

**constructed reentrancy.** The attribute of applications that contain external data and require additional processing to make them reentrant. Contrast with *natural reentrancy*.

**constructor.** A special C++ class member function that has the same name as the class and is used to create an object of that class.

**control character.** (1) A character whose occurrence in a particular context specifies a control function. *ISO Draft*. (2) Synonymous with nonprinting character. *IBM*. (3) A character, other than a graphic character, that affects the recording, processing, transmission, or interpretation of text. *X/Open*.

**control statement.** (1) In programming languages, a statement that is used to alter the continuous sequential execution of statements; a control statement may be a conditional statement, such as IF, or an imperative statement, such as STOP. *ISO Draft*. (2) A statement that changes the path of execution.

**controlling process.** The session leader that establishes the connection to the controlling terminal. If the terminal ceases to be a controlling terminal for this session, the session leader ceases to be the controlling process. *X/Open*. *ISO.1*.

**controlling terminal.** A terminal that is associated with a session. Each session may have at most one

controlling terminal associated with it, and a controlling terminal is associated with exactly one session. Certain input sequences from the controlling terminal cause signals to be sent to all processes in the process group associated with the controlling terminal. *X/Open. ISO.1.*

**conversion.** (1) In programming languages, the transformation between values that represent the same data item but belong to different data types. Information may be lost because of conversion since accuracy of data representation varies among different data types. *ISO-JTC1.* (2) The process of changing from one method of data processing to another or from one data processing system to another. *IBM.* (3) The process of changing from one form of representation to another; for example to change from decimal representation to binary representation. *IBM.* (4) A change in the type of a value. For example, when you add values having different data types, the compiler converts both values to a common form before adding the values.

**conversion descriptor.** A per-process unique value used to identify an open codeset conversion. *X/Open.*

**conversion function.** A member function that specifies a conversion from its class type to another type.

**coordinated universal time (UTC).** Synonym for Greenwich Mean Time (GMT). See *GMT.*

**copy constructor.** A constructor that copies a class object of the same class type.

**Cross System Product.** See *CSP.*

**CSP (Cross System Product).** A set of licensed programs designed to permit the user to develop and run applications using independently defined maps (display and printer formats), data items (records, working storage, files, and single items), and processes (logic). The Cross System Product set consists of two parts: Cross System Product/Application Development (CSP/AD) and Cross System Product/Application Execution (CSP/AE). *IBM.*

**current working directory.** (1) A directory, associated with a process, that is used in path-name resolution for path names that do not begin with a slash. *X/Open. ISO.1.* (2) In the OS/2 operating system, the first directory in which the operating system looks for programs and files and stores temporary files and output. *IBM.* (3) In the OS/390 UNIX environment, a directory that is active and that can be displayed. Relative path name resolution begins in the current directory. *IBM.*

**cursor.** A reference to an element at a specific position in a data structure.

**Customer Information Control System.** See *CICS.*

## D

**data abstraction.** A data type with a private representation and a public set of operations (functions or operators) which restrict access to that data type to that set of operations. The C++ language uses the concept of classes to implement data abstraction.

**DATABASE 2.** Pertaining to an IBM relational database.

**data definition (DD).** (1) In the C and C++ languages, a definition that describes a data object, reserves storage for a data object, and can provide an initial value for a data object. A data definition appears outside a function or at the beginning of a block statement. *IBM.* (2) A program statement that describes the features of, specifies relationships of, or establishes context of, data. *ANSI/ISO.* (3) A statement that is stored in the environment and that externally identifies a file and the attributes with which it should be opened.

**data definition name.** See *ddname.*

**data definition statement.** See *DD statement.*

**data member.** The smallest possible piece of complete data. Elements are composed of data members.

**data object.** (1) A storage area used to hold a value. (2) Anything that exists in storage and on which operations can be performed, such as files, programs, classes, or arrays. (3) In a program, an element of data structure, such as a file, array, or operand, that is needed for the execution of a program and that is named or otherwise specified by the allowable character set of the language in which a program is coded. *IBM.*

**data set.** Under MVS, a named collection of related data records that is stored and retrieved by an assigned name.

**data stream.** A continuous stream of data elements being transmitted, or intended for transmission, in character or binary-digit form, using a defined format. *IBM.*

**data structure.** The internal data representation of an implementation.

**data type.** The properties and internal representation that characterize data.

**Data Window Services (DWS).** Services provided as part of the Callable Services Library that allow manipulation of data objects such as VSAM linear data sets and temporary data objects known as *TEMPSPACE.*

**DBCS (double-byte character set).** A set of characters in which each character is represented by 2 bytes.

Languages such as Japanese, Chinese, and Korean, which contain more symbols than can be represented by 256 code points, require double-byte character sets.

Because each character requires 2 bytes, the typing, display, and printing of DBCS characters requires hardware and programs that support DBCS. *IBM*.

**DCT (destination control table).** A table that contains an entry for each extrapartition, intrapartition, and indirect destination. Extrapartition entries address data sets external to the CICS region. Intrapartition destination entries contain the information required to locate the queue in the intrapartition data set. Indirect destination entries contain the information required to locate the queue in the intrapartition data set.

**ddname (data definition name).** (1) The logical name of a file within an application. The ddname provides the means for the logical file to be connected to the physical file. (2) The part of the data definition before the equal sign. It is the name used in a call to `fopen` or `freopen` to refer to the data definition stored in the environment.

**DD statement (data definition statement).** (1) In MVS, serves as the connection between the logical name of a file and the physical name of the file. (2) A job control statement that defines a file to the operating system, and is a request to the operating system for the allocation of input/output resources.

**dead code elimination.** A process that eliminates code that exists for calculations that are not necessary. Code may be designated as dead by other optimization techniques.

**dead store elimination.** A process that eliminates unnecessary storage use in code. A store is deemed unnecessary if the value stored is never referenced again in the code.

**decimal constant.** (1) A numerical data type used in standard arithmetic operations. (2) A number containing any of the digits 0 through 9. *IBM*.

**decimal overflow.** A condition that occurs when one or more nonzero digits are lost because the destination field in a decimal operation is too short to contain the results.

**declaration.** (1) In the C and C++ languages, a description that makes an external object or function available to a function or a block statement. *IBM*. (2) 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 argument.** An argument that is declared with a default value in a function prototype or declaration. If a call to the function omits this argument, the default

value is used. Arguments with default values must be the trailing arguments in a function prototype argument list.

**default clause.** In the C or C++ languages, within 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. *IBM*.

**default constructor.** A constructor that takes no arguments, or, if it takes arguments, all its arguments have default values.

**default initialization.** The initial value assigned to a data object by the compiler if no initial value is specified by the programmer.

**default locale.** (1) The C locale, which is always used when no selection of locale is performed. (2) A system default locale, named by locale-related environmental variables.

**define directive.** A preprocessor statement that directs the preprocessor to replace an identifier or macro invocation with special code.

**define statement.** A preprocessor statement that causes the preprocessor to replace an identifier or macro call with specified code. *IBM*.

**definition.** (1) A data description that reserves storage and may provide an initial value. (2) A declaration that allocates storage, and may initialize a data object or specify the body of a function.

**degree.** The number of children of a node.

**delete.** (1) A C++ keyword that identifies a free storage deallocation operator. (2) A C++ operator used to destroy objects created by `new`.

**demangling.** The conversion of mangled names back to their original source code names. During C++ compilation, identifiers such as function and static class member names are mangled (encoded) with type and scoping information to ensure type-safe linkage. These mangled names appear in the object file and the final executable file. Demangling (decoding) converts these names back to their original names to make program debugging easier. See also *mangling*.

**denormal.** Pertaining to a number with a value so close to 0 that its exponent cannot be represented normally. The exponent can be represented in a special way at the possible cost of a loss of significance.

**deque.** A queue that can have elements added and removed at both ends. A double-ended queue.

**dequeue.** An operation that removes the first element of a queue.

**dereference.** In the C and C++ languages, the application of the unary operator \* to a pointer to access the object the pointer points to. Also known as *indirection*.

**derivation.** In the C++ language, to derive a class, called a derived class, from an existing class, called a base class.

**derived class.** A class that inherits from a base class. All members of the base class become members of the derived class. You can add additional data members and member functions to the derived class. A derived class object can be manipulated as if it is a base class object. The derived class can override virtual functions of the base class.

**descriptor.** PL/I control block that holds information such as string lengths, array subscript bounds, and area sizes, and is passed from one PL/I routine to another during run time.

**destination control table.** See *DCT*.

**destructor.** A special member function that has the same name as its class, preceded by a tilde (~), and that "cleans up" after an object of that class, for example, freeing storage that was allocated when the object was created. A destructor has no arguments and no return type.

**detach state attribute.** An attribute associated with a thread attribute object. This attribute has two possible values:

- |   |   |
|---|---|
| 0 | Undetached. An undetached thread keeps its resources after termination of the thread. |
| 1 | Detached. A detached thread has its resources freed by the system after termination.  |

**device.** A computer peripheral or an object that appears to the application as such. *X/Open. ISO.1.*

**difference.** For two sets A and B, the difference (A-B) is the set of all elements in A but not in B. For bags, there is an additional rule for duplicates: If bag P contains an element *m* times and bag Q contains the same element *n* times, then, if *m* > *n*, the difference contains that element *m-n* times. If *m* ≤ *n*, the difference contains that element zero times.

**digraph.** A combination of two keystrokes used to represent unavailable characters in a C++ source program. Digraphs are read as tokens during the preprocessor phase.

**directory.** A type of file containing the names and controlling information for other files or other directories. *IBM.*

**Direct-to-SOM (DTS).** (1) Term applied to the method by which the OS/390 C++ compiler converts existing

C++ classes to SOM classes. (2) Term applied to a class that has been converted to SOM by the OS/390 C++ compiler.

**disabled signal.** Synonym for *enabled signal*.

**display.** To direct the output to the user's terminal. If the output is not directed to the terminal, the results are undefined. *X/Open.*

**do statement.** In the C and C++ compilers, a looping statement that contains the keyword "do", followed by a statement (the action), the keyword "while", and an expression in parentheses (the condition). *IBM.*

**dot.** The file name consisting of a single dot character (.). *X/Open. ISO.1.*

**double-byte character set.** See *DBCS*.

**double-precision.** Pertaining to the use of two computer words to represent a number in accordance with the required precision. *ISO-JTC1. ANSI/ISO.*

**double-quote.** The character ", also known as *quotation mark*. *X/Open.*

This character is named <quotation-mark> in the portable character set.

**doubleword.** A contiguous sequence of bits or characters that comprises two computer words and is capable of being addressed as a unit. *IBM.*

**dynamic.** Pertaining to an operation that occurs at the time it is needed rather than at a predetermined or fixed time. *IBM.*

**dynamic allocation.** Assignment of system resources to a program when the program is executed rather than when it is loaded into main storage. *IBM.*

**dynamic binding.** The act of resolving references to external variables and functions at run time.

**dynamic link library (DLL).** A file containing executable code and data bound to a program at run time. The code and data in a dynamic link library can be shared by several applications simultaneously. Compiling code with the DLL option does not mean that the produced executable will be a DLL. To create a DLL, use #pragma export or the EXPORTALL compiler option.

**DSA (dynamic storage area).** An area of storage obtained during the running of an application that consists of a register save area and an area for automatic data, such as program variables. DSAs are generally allocated within Language Environment-managed stack segments. DSAs are added to the stack when a routine is entered and removed upon exit in a last in, first out (LIFO) manner. In Language Environment, a DSA is known as a stack frame.



**dynamic storage.** Synonym for *automatic storage*.

**dynamic storage area.** See DSA

## E

**EBCDIC.** See *extended binary-coded decimal interchange code*.

**effective group ID.** An attribute of a process that is used in determining various permissions, including file access permissions. This value is subject to change during the process lifetime, as described in the *exec* family of functions and `setgid()`. *X/Open. ISO.1.*

**effective user ID.** (1) The user ID associated with the last authenticated user or the last `setuid()` program. It is equal to either the real or the saved user ID. (2) The current user ID, but not necessarily the user's login ID; for example, a user logged in under a login ID may change to another user's ID. The ID to which the user changes becomes the effective user ID until the user switches back to the original login ID. All discretionary access decisions are based on the effective user ID. *IBM.* (3) An attribute of a process that is used in determining various permissions, including file access permissions. This value is subject to change during the process lifetime, as described in *exec* and `setuid()`. *X/Open. ISO.1.*

**elaborated type specifier.** A specifier typically used in an incomplete class declaration to qualify types that are otherwise hidden.

**element.** The component of an array, subrange, enumeration, or set.

**element equality.** A relation that determines if two elements are equal.

**element occurrence.** A single instance of an element in a collection. In a unique collection, element occurrence is synonymous with element value.

**element value.** All the instances of an element with a particular value in a collection. In a nonunique collection, an element value may have more than one occurrence. In a unique collection, element value is synonymous with element occurrence.

**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 started when the if condition evaluates to a value of zero (false). *IBM.*

**empty line.** A line consisting of only a new-line character. *X/Open.*

**empty string.** (1) A string whose first byte is a null byte. Synonymous with null string. *X/Open.* (2) A character array whose first element is a null character. *ISO.1.*

**enabled signal.** The occurrence of an enabled signal results in the default system response or the execution of an established signal handler. If disabled, the occurrence of the signal is ignored.

**encapsulation.** Hiding the internal representation of data objects and implementation details of functions from the client program. This enables the end user to focus on the use of data objects and functions without having to know about their representation or implementation.

**enclave.** In the Language Environment for MVS and VM, an independent collection of routines, one of which is designated as the main routine. An enclave is roughly analogous to a program or run unit.

**enqueue.** An operation that adds an element as the last element to a queue.

**entry point.** In assembler language, the address or label of the first instruction that is executed when a routine is entered for execution.

**enumeration constant.** In the C or C++ language, an identifier, with an associated integer value, defined in an enumerator. An enumeration constant may be used anywhere an integer constant is allowed. *IBM.*

**enumeration data type.** (1) In the Fortran, C, and C++ language, a data type that represents a set of values that a user defines. *IBM.* (2) A type that represents integers and a set of enumeration constants. Each enumeration constant has an associated integer value.

**enumeration tag.** In the C and C++ language, the identifier that names an enumeration data type. *IBM.*

**enumeration type.** An enumeration type defines a set of enumeration constants. In the C++ language, an enumeration type is a distinct data type that is not an integral type.

**enumerator.** In the C and C++ language, an enumeration constant and its associated value. *IBM.*

**equivalence class.** (1) A grouping of characters that are considered equal for the purpose of collation; for example, many languages place an uppercase character in the same equivalence class as its lowercase form, but some languages distinguish between accented and unaccented character forms for the purpose of collation. *IBM.* (2) A set of collating elements with the same primary collation weight.

Elements in an equivalence class are typically elements that naturally group together, such as all accented letters based on the same base letter.

The collation order of elements within an equivalence class is determined by the weights assigned on any subsequent levels after the primary weight. *X/Open.*

**escape sequence.** (1) A representation of a character. An escape sequence contains the `\` symbol followed by one of the characters: `a`, `b`, `f`, `n`, `r`, `t`, `v`, `'`, `"`, `x`, `\`, or followed by one or more octal or hexadecimal digits. (2) A sequence of characters that represent, for example, nonprinting characters, or the exact code point value to be used to represent variant and nonvariant characters regardless of code page. (3) In the C and C++ language, an escape character followed by one or more characters. The escape character indicates that a different code, or a different coded character set, is used to interpret the characters that follow. Any member of the character set used at runtime can be represented using an escape sequence. (4) A character that is preceded by a backslash character and is interpreted to have a special meaning to the operating system. (5) A sequence sent to a terminal to perform actions such as moving the cursor, changing from normal to reverse video, and clearing the screen. Synonymous with multibyte control. *IBM.*

**exception.** (1) 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 (also called throwing the exception). (2) In programming languages, an abnormal situation that may arise during execution, that may cause a deviation from the normal execution sequence, and for which facilities exist in a programming language to define, raise, recognize, ignore, and handle it; for example, (ON-) condition in PL/I, exception in ADA. *ISO-JTC1.*

**executable.** A load module or program object which has yet to be loaded into memory for execution.

**executable file.** A regular file acceptable as a new process image file by the equivalent of the *exec* family of functions, and thus usable as one form of a utility. The standard utilities described as compilers can produce executable files, but other unspecified methods of producing executable files may also be provided. The internal format of an executable file is unspecified, but a conforming application cannot assume an executable file is a text file. *X/Open.*

**exception handler.** (1) Exception handlers are catch blocks in C++ applications. Catch blocks catch exceptions when they are thrown from a function enclosed in a try block. Try blocks, catch blocks, and throw expressions are the constructs used to implement formal exception handling in C++ applications. (2) A set of routines used to detect deadlock conditions or to process abnormal condition processing. An exception handler allows the normal running of processes to be interrupted and resumed. *IBM.*

**executable file.** A regular file acceptable as a new process image file by the equivalent of the *exec* family of functions, and thus usable as one form of a utility. The standard utilities described as compilers can produce executable files, but other unspecified methods of producing executable files may also be provided.

The internal format of an executable file is unspecified, but a conforming application cannot assume an executable file is a text file. *X/Open.*

**executable program.** A program that has been link-edited and therefore can be run in a processor. *IBM.*

**extended binary-coded data interchange code (EBCDIC).** A coded character set of 256 8-bit characters. *IBM.*

**extension.** (1) An element or function not included in the standard language. (2) File name extension.

**external data definition.** A description of a variable appearing outside a function. It causes the system to allocate storage for that variable and makes that variable accessible to all functions that follow the definition and are located in the same file as the definition. *IBM.*

**extern storage class specifier.** A specifier that enables the programmer to declare objects and functions that several source files can use.

## F

**feature test macro (FTM).** A macro (`#define`) used to determine whether a particular set of features will be included from a header. *X/Open. ISO.1.*

**FIFO special file.** A type of file with the property that data written to such a file is read on a first-in-first-out basis. Other characteristics of FIFOs are described in `open()`, `read()`, `write()`, and `lseek()`. *X/Open. ISO.1.*

**file access permissions.** The standard file access control mechanism uses the file permission bits. The bits are set at the time of file creation by functions such as `open()`, `creat()`, `mkdir()`, and `mkfifo()` and can be changed by `chmod()`. The bits are read by `stat()` or `fstat()`. *X/Open.*

**file descriptor.** (1) A small positive integer that the system uses instead of the file name to identify an open file. *IBM.* (2) A per-process unique, non-negative integer used to identify an open file for the purpose of file access. *ISO.1.*

The value of a file descriptor is from zero to `{OPEN_MAX}`—which is defined in `<limits.h>`. A process can have no more than `{OPEN_MAX}` file descriptors open simultaneously. File descriptors may also be used to implement directory streams. *X/Open.*

**file mode.** An object containing the *file mode bits* and file type of a file, as described in `<sys/stat.h>`. *X/Open.*

**file mode bits.** A file's file permission bits, set-user-ID-on-execution bit (`S_ISUID`) and set-group-ID-on-execution bit (`S_ISGID`). *X/Open.*

**file permission bits.** Information about a file that is used, along with other information, to determine if a process has read, write, or execute/search permission to a file. The bits are divided into three parts: owner, group, and other. Each part is used with the corresponding file class of process. These bits are contained in the file mode, as described in *<sys/stat.h>*. The detailed usage of the file permission bits is described in *file access permissions. X/Open. ISO.1.*

**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.

**filter.** A command whose operation consists of reading data from standard input or a list of input files and writing data to standard output. Typically, its function is to perform some transformation on the data stream. *X/Open.*

**first element.** The element visited first in an iteration over a collection. Each collection has its own definition for first element. For example, the first element of a sorted set is the element with the smallest value.

**flat collection.** A collection that has no hierarchical structure.

**float constant.** (1) A constant representing a nonintegral number. (2) 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). *IBM.*

**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.

**foreground process.** (1) A process that must run to completion before another command is issued. The foreground process is in the foreground process group, which is the group that receives the signals generated by a terminal. *IBM.* (2) A process that is a member of a foreground process group. *X/Open. ISO.1.*

**foreground process group.** (1) The group that receives the signals generated by a terminal. *IBM.* (2) A process group whose member processes have certain privileges, denied to processes in background process groups, when accessing their controlling terminal. Each session that has established a connection with a controlling terminal has exactly one process group of the session as the foreground process group of that controlling terminal. *X/Open. ISO.1.*

**foreground process group ID.** The process group ID of the foreground process group. *X/Open. ISO.1.*

**form-feed character.** A character in the output stream that indicates that printing should start on the next

page of an output device. The formfeed is the character designated by '\f' in the C and C++ language. If the formfeed is not the first character of an output line, the result is unspecified. It is unspecified whether this character is the exact sequence transmitted to an output device by the system to accomplish the movement to the next page. *X/Open.*

**forward declaration.** A declaration of a class or function made earlier in a compilation unit, so that the declared class or function can be used before it has been defined.

**freestanding application.** (1) An application that is created to run without the run-time environment or library with which it was developed. (2) An OS/390 C/C++ application that does not use the services of the dynamic OS/390 C/C++ run-time library or of the Language Environment. Under OS/390 C support, this ability is a feature of the System Programming C support.

**free store.** Dynamically allocated memory. New and delete are used to allocate and deallocate free store.

**friend class.** A class in which all the member functions are granted access to the private and protected members of another class. It is named in the declaration of another class and uses the keyword *friend* as a prefix to the class. For example, the following source code makes all the functions and data in class *you* friends of class *me*:

```
class me {  
    friend class you;  
    // ...  
};
```

**friend function.** A function that is granted access to the private and protected parts of a class. It is named in the declaration of the other class with the prefix *friend*.

**function.** A named group of statements that can be called and evaluated and can return a value to the calling statement. *IBM.*

**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 values. *IBM.*

**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. *IBM.*

**function definition.** The complete description of a function. A function definition contains an optional storage class specifier, an optional type specifier, a function declarator, optional 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 (;). The declaration is required by the compiler at the time that the function is declared, so that the compiler can check the type.

**function scope.** Labels that are declared in a function have function scope and can be used anywhere in that function.

**function template.** Provides a blueprint describing how a set of related individual functions can be constructed.

## G

**Generalization.** Refers to a class, function, or static data member which derives its definition from a template. An instantiation of a template function would be a generalization.

**generic class.** Synonym for *class templates*.

**global.** Pertaining to information available to more than one program or subroutine. *IBM*.

**global scope.** Synonym for *file scope*.

**global variable.** A symbol defined in one program module that is used in other independently compiled program modules.

**GMT (Greenwich Mean Time).** The solar time at the meridian of Greenwich, formerly used as the prime basis of standard time throughout the world. GMT has been superseded by coordinated universal time (UTC).

**graphic character.** (1) A visual representation of a character, other than a control character, that is normally produced by writing, printing, or displaying. *ISO Draft*. (2) A character that can be displayed or printed. *IBM*.

**Graphical Data Display Manager (GDDM).** Pertaining to an IBM licensed program that provides a group of routines that allows pictures to be defined and displayed procedurally through function routines that correspond to graphic primitives. *IBM*.

**Greenwich Mean Time.** See GMT.

**group ID.** (1) A non-negative integer that is used to identify a group of system users. Each system user is a member of at least one group. When the identity of a group is associated with a process, a group ID value is referred to as a real group ID, an effective group ID, one of the supplementary group IDs or a saved set-group-ID. *X/Open*. (2) A non-negative integer, which can be contained in an object of type *gid\_t*, that is used to identify a group of system users. *ISO.1*.

## H

**halfword.** A contiguous sequence of bits or characters that constitutes half a computer word and can be addressed as a unit. *IBM*.

**hash function.** A function that determines which category, or bucket, to put an element in. A hash function is needed when implementing a hash table.

**hash table.** (1) A data structure that divides all elements into (preferably) equal-sized categories, or buckets, to allow quick access to the elements. The hash function determines which bucket an element belongs in. (2) A table of information that is accessed by way of a shortened search key (that hash value). Using a hash table minimizes average search time.

**header file.** A text file that contains declarations used by a group of functions, programs, or users.

**heap storage.** An area of storage used for allocation of storage whose lifetime is not related to the execution of the current routine. The heap consists of the initial heap segment and zero or more increments.

**hexadecimal constant.** A constant, usually starting with special characters, that contains only hexadecimal digits. Three examples for the hexadecimal constant with value 0 would be '\x00', '0x0', or '0X00'.

**hiperspace memory file.** An IBM file used under MVS to deal with memory files as large as 2 gigabytes. *IBM*.

**hooks.** Instructions inserted into a program by a compiler at compile-time. Using hooks, you can set break-points to instruct the Debug Tool to gain control of the program at selected points during its execution.

**hybrid code.** Program statements that have not been internationalized with respect to code page, especially where data constants contain variant characters. Such statements can be found in applications written in older implementations of MVS, which required syntax statements to be written using code page IBM-1047 exclusively. Such applications cannot be converted from one code page to another using *iconv()*.

## I

**I18N.** Abbreviation for *internationalization*.

**identifier.** (1) One or more characters used to identify or name a data element and possibly to indicate certain properties of that data element. *ANSI/ISO*. (2) In programming languages, a token that names a data object such as a variable, an array, a record, a subprogram, or a function. *ANSI/ISO*. (3) A sequence of letters, digits, and underscores used to identify a data object or function. *IBM*.



**if statement.** A conditional statement that contains the keyword `if`, followed by an expression in parentheses (the condition), a statement (the action), and an optional `else` clause (the alternative action). *IBM.*

**ILC (interlanguage call).** A function call made by one language to a function coded in another language. Interlanguage calls are used to communicate between programs written in different languages.

**ILC (interlanguage communication).** The ability of routines written in different programming languages to communicate. ILC support enables the application writer to readily build applications from component routines written in a variety of languages.

**implementation-defined behavior.** Application behavior that is not defined by the standards. The implementing compiler and library defines this behavior when a program contains correct program constructs or uses correct data. Programs that rely on implementation-defined behavior may behave differently on different C or C++ implementations. Refer to the OS/390 C/C++ books that are listed in "IBM OS/390 C/C++ and Related Publications" on page 4 for information about implementation-defined behavior in the OS/390 C/C++ environment. Contrast with *unspecified behavior* and *undefined behavior*.

**IMS (Information Management System).** Pertaining to an IBM database/data communication (DB/DC) system that can manage complex databases and networks. *IBM.*

**include directive.** A preprocessor directive that causes the preprocessor to replace the statement with the contents of a specified file.

**include file.** See *header file*.

**include statement.** In the C and C++ languages, a preprocessor statement that causes the preprocessor to replace the statement with the contents of a specified file. *IBM.*

**incomplete class declaration.** A class declaration that does not define any members of a class. Until a class is fully declared, or defined, you can only use the class name where the size of the class is not required. Typically an incomplete class declaration is used as a forward declaration.

**incomplete type.** A type that has no value or meaning when it is first declared. There are three incomplete types: `void`, arrays of unknown size and structures and unions of unspecified content. A `void` type can never be completed. Arrays of unknown size and structures or unions of unspecified content can be completed in further declarations.

**indirection.** (1) A mechanism for connecting objects by storing, in one object, a reference to another object. (2)

In the C and C++ languages, the application of the unary operator `*` to a pointer to access the object to which the pointer points.

**indirection class.** Synonym for *reference class*.

**inheritance.** A technique that allows the use of an existing class as the base for creating other classes.

**initial heap.** The OS/390 C/C++ heap controlled by the HEAP runtime option and designated by a `heap_id` of 0. The initial heap contains dynamically allocated user data.

**initializer.** An expression used to initialize data objects. The C++ language, supports the following types of initializers:

- An expression followed by an assignment operator that is used to initialize fundamental data type objects or class objects that contain copy constructors.
- A parenthesized expression list that is used to initialize base classes and members that use constructors.

Both the C and C++ languages support an expression enclosed in braces ( `{ }` ), that used to initialize aggregates.

**inlined function.** A function whose actual code replaces a function call. A function that is both declared and defined in a class definition is an example of an inline function. Another example is one which you explicitly declared inline by using the keyword `inline`. Both member and nonmember functions can be inlined.

**input stream.** A sequence of control statements and data submitted to a system from an input unit. Synonymous with *input job stream*, *job input stream*. *IBM.*

**instance.** An object-oriented programming term synonymous with *object*. An instance is a particular instantiation of a data type. It is simply a region of storage that contains a value or group of values. For example, if a class `box` is previously defined, two instances of a class `box` could be instantiated with the declaration: `box box1, box2;`

**instantiate.** To create or generate a particular instance or object of a data type. For example, an instance `box1` of class `box` could be instantiated with the declaration: `box box1;`

**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.

**instruction scheduling.** An optimization technique that reorders instructions in code to minimize execution time.

**integer constant.** A decimal, octal, or hexadecimal constant.

**integral object.** A character object, an object having an enumeration type, an object having variations of the type `int`, or an object that is a bit field.

**Interactive System Productivity Facility.** See *ISPF*.

**interlanguage call.** See *ILC* (*interlanguage call*).

**interlanguage communication.** See *ILC* (*interlanguage communication*).

**internationalization.** The capability of a computer program to adapt to the requirements of different native languages, local customs, and coded character sets. *X/Open*.

Synonymous with *I18N*.

**interoperability.** The capability to communicate, execute programs, or transfer data among various functional units in a way that requires the user to have little or no knowledge of the unique characteristics of those units.

**Interprocedural Analysis.** See *IPA*.

**interprocess communication.** (1) The exchange of information between processes or threads through semaphores, queues, and shared memory. (2) The process by which programs communicate data to each other to synchronize their activities. Semaphores, signals, and internal message queues are common methods of inter-process communication.

**I/O Stream library.** A class library that provides the facilities to deal with many varieties of input and output.

**IPA (Interprocedural Analysis).** A process for performing optimizations across compilation units.

**ISPF (Interactive System Productivity Facility).** An IBM licensed program that serves as a full-screen editor and dialogue manager. Used for writing application programs, it provides a means of generating standard screen panels and interactive dialogues between the application programmer and terminal user. (*ISPF*)

**iteration.** The process of repeatedly applying a function to a series of elements in a collection until some condition is satisfied.

## J

**JCL (job control language).** A control language used to identify a job to an operating system and to describe the job's requirement. *IBM*.

**job control.** A facility that allows users to selectively stop (suspend) the execution of a process and continue (resume) their execution at a later point.

The user typically employs this facility via the interactive interface jointly supplied by the terminal I/O driver and a command interpreter. *X/Open. ISO.1*.

## K

**keyword.** (1) A predefined word reserved for the C and C++ languages, that may not be used as an identifier. (2) A symbol that identifies a parameter in JCL.

**kind attribute.** An attribute for a mutex attribute object. This attribute's value determines whether the mutex can be locked once or more than once for a thread and whether state changes to the mutex will be reported to the debug interface.

## L

**label.** An identifier within or attached to a set of data elements. *ISO Draft*.

**Language Environment.** Abbreviated form of IBM Language Environment for MVS and VM. Pertaining to an IBM software product that provides a common runtime environment and runtime services to applications compiled by Language Environment-conforming compilers.

**last element.** The element visited last in an iteration over a collection. Each collection has its own definition for last element. For example, the last element of a sorted set is the element with the largest value.

**late binding.** Allowing the system to determine the specific class of the object and invoke the appropriate function implementations at run time. Late binding or dynamic binding hides the differences between a group of related classes from the application program.

**leaves.** Nodes without children. Synonymous with terminals.

**lexically.** Relating to the left-to-right order of units.

**library.** (1) A collection of functions, calls, subroutines, or other data. *IBM*. (2) A set of object modules that can be specified in a link command.

**linkage editor.** Synonym for linker. The linkage editor has been replaced by the *binder* for the MVS/ESA or OS/390 operating systems. See *binder*.

**Linkage.** Refers to the binding between a reference and a definition. A function has internal linkage if the function is defined inline as part of the class, is declared with the inline keyword, or is a nonmember function declared with the static keyword. All other functions have external linkage.

**linker.** A computer program for creating load modules from one or more object modules by resolving cross references among the modules and, if necessary, adjusting addresses. *IBM.*

**link pack area (LPA).** In MVS, an area of storage containing re-enterable routines from system libraries. Their presence in main storage saves loading time.

**literal.** (1) In programming languages, a lexical unit that directly represents a value; for example, 14 represents the integer fourteen, "APRIL" represents the string of characters APRIL, 3.0005E2 represents the number 300.05. *ISO-JTC1.* (2) A symbol or a quantity in a source program that is itself data, rather than a reference to data. *IBM.* (3) A character string whose value is given by the characters themselves; for example, the numeric literal 7 has the value 7, and the character literal CHARACTERS has the value CHARACTERS. *IBM.*

**loader.** A routine, commonly a computer program, that reads data into main storage. *ANSI/ISO.*

**load module.** All or part of a computer program in a form suitable for loading into main storage for execution. A load module is usually the output of a linkage editor. *ISO Draft.*

**local.** (1) In programming languages, pertaining to the relationship between a language object and a block such that the language object has a scope contained in that block. *ISO-JTC1.* (2) Pertaining to that which is defined and used only in one subdivision of a computer program. *ANSI/ISO.*

**local customs.** The conventions of a geographical area or territory for such things as date, time, and currency formats. *X/Open.*

**locale.** The definition of the subset of a user's environment that depends on language and cultural conventions. *X/Open.*

**localization.** The process of establishing information within a computer system specific to the operation of particular native languages, local customs, and coded character sets. *X/Open.*

**local scope.** A name declared in a block has scope within the block, and can therefore only be used in that block.

**Long name.** An external name C++ name in an object module, or and external name in an object module created by the C compiler when the LONGNAME option is used. Long names are up to 1024 characters long and may contain both upper-case and lower-case characters.

**lvalue.** An expression that represents a data object that can be both examined and altered.

## M

**macro.** An identifier followed by arguments (may be a parenthesized list of arguments) that the preprocessor replaces with the replacement code located in a preprocessor #define directive.

**macro call.** Synonym for *macro*.

**macro instruction.** Synonym for *macro*.

**main function.** An external function with the identifier main that is the first user function—aside from exit routines and C++ static object constructors—to get control when program execution begins. Each C and C++ program must have exactly one function named main.

**makefile.** A text file containing a list of your application's parts. The make utility uses makefiles to maintain application parts and dependencies.

**make utility.** Maintains all of the parts and dependencies for your application. The make utility uses a makefile to keep the parts of your program synchronized. If one part of your application changes, the make utility updates all other files that depend on the changed part. This utility is available under the OS/390 shell and by default, uses the c89 utility to recompile and bind your application.

**mangling.** The encoding during compilation of identifiers such as function and variable names to include type and scope information. These mangled names ensure type-safe linkage. See also *demangling*.

**manipulator.** A value that can be inserted into streams or extracted from streams to affect or query the behavior of the stream.

**member.** A data object or function in a structure, union, or class. Members can also be classes, enumerations, bit fields, and type names.

**member function.** (1) An operator or function that is declared as a member of a class. A member function has access to the private and protected data members and member functions of objects of its class. Member functions are also called methods. (2) A function that performs operations on a class.

**method.** In the C++ language, a synonym for *member function*.

**migrate.** To move to a changed operating environment, usually to a new release or version of a system. *IBM.*

**module.** A program unit that usually performs a particular function or related functions, and that is distinct and identifiable with respect to compiling, combining with other units, and loading.

**multibyte character.** A mixture of single-byte characters from a single-byte character set and double-byte characters from a double-byte character set.

**multicharacter collating element.** A sequence of two or more characters that collate as an entity. For example, in some coded character sets, an accented character is represented by a non-spacing accent, followed by the letter. Other examples are the Spanish elements *ch* and *ll*. *X/Open*.

**multiple inheritance.** An object-oriented programming technique implemented in the C++ language through derivation, in which the derived class inherits members from more than one base class.

**multitasking.** A mode of operation that allows concurrent performance, or interleaved execution of two or more tasks. *ISO/JTC1. ANSI/ISO*.

**mutex.** A flag used by a semaphore to protect shared resources. The mutex is locked and unlocked by threads in a program. A mutex can only be locked by one thread at a time and can only be unlocked by the same thread that locked it. The current owner of a mutex is the thread that it is currently locked by. An unlocked mutex has no current owner.

**mutex attribute object.** Allows the user to manage the characteristics of mutexes in their application by defining a set of values to be used for the mutex during its creation. A mutex attribute object allows the user to create many mutexes with the same set of characteristics without redefining the same set of characteristics for each mutex created.

**mutex object.** Used to identify a mutex.

## N

**name space.** A category used to group similar types of identifiers.

**named pipe.** A FIFO file. Named pipes allow transfer of data between processes in a FIFO manner and synchronization of process execution. Allows processes to communicate even though they do not know what processes are on the other end of the pipe.

**natural reentrancy.** A program that contains no writable static and requires no additional processing to make it reentrant is considered naturally reentrant.

**nested class.** A class defined within the scope of another class.

**nested enclave.** A new enclave created by an existing enclave. The nested enclave that is created must be a new main routine within the process. See also *child enclave* and *parent enclave*.

**newline character.** A character that in the output stream indicates that printing should start at the

beginning of the next line. The newline character is designated by '\n' in the C and C++ language. It is unspecified whether this character is the exact sequence transmitted to an output device by the system to accomplish the movement to the next line. *X/Open*.

**nickname.** Synonym for alias.

**nonprinting character.** See *control character*.

**null character (NUL).** The ASCII or EBCDIC character '\0' with the hex value 00, all bits turned off. It is used to represent the absence of a printed or displayed character. This character is named <NUL> in the portable character set.

**null pointer.** The value that is obtained by converting the number 0 into a pointer; for example, (void \*) 0. The C and C++ languages guarantee that this value will not match that of any legitimate pointer, so it is used by many functions that return pointers to indicate an error. *X/Open*.

**null statement.** A C or C++ statement that consists solely of a semicolon.

**null string.** (1) A string whose first byte is a null byte. Synonymous with *empty string*. *X/Open*. (2) A character array whose first element is a null character. *ISO.1*.

**null value.** A parameter position for which no value is specified. *IBM*.

**null wide-character code.** A wide-character code with all bits set to zero. *X/Open*.

**number sign.** The character #, also known as *pound sign* and *hash sign*. This character is named <number-sign> in the portable character set.

## O

**object.** (1) A region of storage. An object is created when a variable is defined. An object is destroyed when it goes out of scope. (See also *instance*.) (2) In object-oriented design or programming, an abstraction consisting of data and the operations associated with that data. See also *class*. *IBM*. (3) An instance of a class.

**object code.** Machine-executable instructions, usually generated by a compiler from source code written in a higher level language (such as the C++ language). For programs that must be linked, object code consists of relocatable machine code.

**object module.** (1) All or part of an object program sufficiently complete for linking. Assemblers and compilers usually produce object modules. *ISO Draft*. (2) A set of instructions in machine language produced by a compiler from a source program. *IBM*.

**object-oriented programming.** A programming approach based on the concepts of data abstraction and



inheritance. Unlike procedural programming techniques, object-oriented programming concentrates not on how something is accomplished, but on what data objects comprise the problem and how they are manipulated.

**octal constant.** The digit 0 (zero) followed by any digits 0 through 7.

**open file.** A file that is currently associated with a file descriptor. *X/Open. ISO.1.*

**operand.** An entity on which an operation is performed. *ISO-JTC1. ANSI/ISO.*

**operating system (OS).** Software that controls functions such as resource allocation, scheduling, input/output control, and data management.

**operator function.** An overloaded operator that is either a member of a class or that takes at least one argument that is a class type or a reference to a class type.

**operator precedence.** In programming languages, an order relation defining the sequence of the application of operators within an expression. *ISO-JTC1.*

**orientation of a stream.** After application of an input or output function to a stream, it becomes either byte-oriented or wide-oriented. A byte-oriented stream is a stream that had a byte input or output function applied to it when it had no orientation. A wide-oriented stream is a stream that had a wide character input or output function applied to it when it had no orientation. A stream has no orientation when it has been associated with an external file but has not had any operations performed on it.

**OS/390 UNIX System Services (OS/390 UNIX).** An element of the OS/390 operating system, (formerly known as OpenEdition). OS/390 UNIX includes a POSIX system Application Programming Interface for the C language, a shell and utilities component, and a dbx debugger. All the components conform to IEEE POSIX standards (ISO 9945-1: 1990/IEEE POSIX 1003.1-1990, IEEE POSIX 1003.1a, IEEE POSIX 1003.2, and IEEE POSIX 1003.4a).

**overflow.** (1) A condition that occurs when a portion of the result of an operation exceeds the capacity of the intended unit of storage. (2) That portion of an operation that exceeds the capacity of the intended unit of storage. *IBM.*

**overlay.** The technique of repeatedly using the same areas of internal storage during different stages of a program. *ANSI/ISO.*

**overloading.** An object-oriented programming technique that allows you to redefine functions and most standard C++ operators when the functions and operators are used with class types.

## P

**parameter.** (1) In the C and C++ languages, an object declared as part of a function declaration or definition that acquires a value on entry to the function, or an identifier following the macro name in a function-like macro definition. *X/Open.* (2) Data passed between programs or procedures. *IBM.*

**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.

**parent enclave.** The enclave that issues a call to system services or language constructs to create a nested or child enclave. See also *child enclave* and *nested enclave*.

**parent process.** (1) The program that originates the creation of other processes by means of spawn or exec function calls. See also *child process*. (2) A process that creates other processes.

**parent process ID.** (1) An attribute of a new process identifying the parent of the process. The parent process ID of a process is the process ID of its creator, for the lifetime of the creator. After the creator's lifetime has ended, the parent process ID is the process ID of an implementation-dependent system process. *X/Open.* (2) An attribute of a new process after it is created by a currently active process. *ISO.1.*

**partitioned concatenation.** Specifying multiple PDSs or PDSEs under one ddname. The concatenated data sets act as one big PDS or PDSE and access can be made to any member with a unique name. An attempted access to a member whose name occurs more than once in the concatenated data sets, returns the first member with that name found in the entire concatenation.

**partitioned data set (PDS).** A data set in direct access storage that is divided into partitions, called members, each of which can contain a program, part of a program, or data. *IBM.*

**partitioned data set extended (PDSE).** Similar to *partitioned data set*, but with extended capabilities.

**path name.** (1) A string that is used to identify a file. A path name consists of, at most, [PATH\_MAX] bytes, including the terminating null character. It has an optional beginning slash, followed by zero or more file names separated by slashes. If the path name refers to a directory, it may also have one or more trailing slashes. Multiple successive slashes are treated as one slash. A path name that begins with two successive slashes may be interpreted in an implementation-dependent manner, although more than two leading slashes are treated as a single slash. The interpretation

of the path name is described in *path name resolution*.  
**ISO.1.** (2) A file name specifying all directories leading to the file.

**path name resolution.** Path name resolution is performed for a process to resolve a path name to a particular file in a file hierarchy. There may be multiple path names that resolve to the same file. *X/Open*.

**pattern.** A sequence of characters used either with regular expression notation or for path name expansion, as a means of selecting various characters strings or path names, respectively. The syntaxes of the two patterns are similar, but not identical. *X/Open*.

**PCH (precompiled header).** One or more headers that have already been compiled.

**period.** The character (.). The term *period* is contrasted against *dot*, which is used to describe a specific directory entry. This character is named `<period>` in the portable character set.

**permissions.** Codes that determine how a file can be used by any users who work on the system. See also *file access permissions*. *IBM*.

**persistent environment.** A program can explicitly establish a persistent environment, direct functions to it, and explicitly terminate it.

**pointer.** In the C and C++ languages, a variable that holds the address of a data object or a function. *IBM*.

**pointer class.** A class that implements pointers.

**pointer to member.** An operator used to access the address of non-static members of a class.

**polymorphism.** The technique of taking an abstract view of an object or function and using any concrete objects or arguments that are derived from this abstract view.

**portable character set.** The set of characters specified in POSIX 1003.2, section 2.4:

```
<NUL>
<alert>
<backspace>
<tab>
<newline>
<vertical-tab>
<form-feed>
<carriage-return>
<space>
<exclamation-mark>    !
<quotation-mark>      "
<number-sign>         #
<dollar-sign>          $
<percent-sign>         %
<ampersand>            &
<apostrophe>          '
<left-parenthesis>    (
<right-parenthesis>  )
```

```
<asterisk>             *
<plus-sign>            +
<comma>                ,
<hyphen>               -
<hyphen-minus>        -
<period>               .
<slash>                /
<zero>                 0
<one>                  1
<two>                  2
<three>                3
<four>                 4
<five>                 5
<six>                  6
<seven>                7
<eight>                8
<nine>                 9
<colon>                :
<semicolon>           ;
<less-than-sign>      <
<equals-sign>         =
<greater-than-sign>   >
<question-mark>       ?
<commercial-at>      @

<A>                    A
<B>                    B
<C>                    C
<D>                    D
<E>                    E
<F>                    F
<G>                    G
<H>                    H
<I>                    I
<J>                    J
<K>                    K
<L>                    L
<M>                    M
<N>                    N
<O>                    O
<P>                    P
<Q>                    Q
<R>                    R
<S>                    S
<T>                    T
<U>                    U
<V>                    V
<W>                    W
<X>                    X
<Y>                    Y
<Z>                    Z

<left-square-bracket> [
<backslash>           \
<reverse-solidus>     \
<right-square-bracket> ]
<circumflex>          ^
<circumflex-accent>   ^
<underscore>          _
<low-line>             ~
<grave-accent>        `

<a>                    a
<b>                    b
<c>                    c
<d>                    d
<e>                    e
<f>                    f
<g>                    g
```

<h>	h
<i>	i
<j>	j
<k>	k
<l>	l
<m>	m
<n>	n
<o>	o
<p>	p
<q>	q
<r>	r
<s>	s
<t>	t
<u>	u
<v>	v
<w>	w
<x>	x
<y>	y
<z>	z
<left-brace>	{
<left-curly-bracket>	{
<vertical-line>	
<right-brace>	}
<right-curly-bracket>	}
<tilde>	~

**portable file name character set.** The set of characters from which portable file names are constructed. For a file name to be portable across implementations conforming to the ISO POSIX-1 standard and to ISO/IEC 9945, it must consist only of the following characters:

```
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 last three characters are the period, underscore, and hyphen characters, respectively. The hyphen must not be used as the first character of a portable file name. Upper- and lower-case letters retain their unique identities between conforming implementations. In the case of a portable path name, the slash character may also be used. *X/Open. ISO.1.*

**portability.** The ability of a programming language to compile successfully on different operating systems without requiring changes to the source code.

**positional parameter.** A parameter that must appear in a specified location relative to other positional parameters. *IBM.*

**precedence.** The priority system for grouping different types of operators with their operands.

**precompiled header.** See *PCH*.

**predefined macros.** Frequently used routines provided by an application or language for the programmer.

**preinitialization.** A process by which an environment or library is initialized once and can then be used repeatedly to avoid the inefficiency of initializing the environment or library each time it is needed.

**prelinker.** A utility provided with OS/390 Language Environment that you can use to process application programs that require DLL support, or contain either constructed reentrancy or external symbol names that are longer than 8 characters. You require the prelinker, or its equivalent function which is provided by the binder, to process all C++ applications, or C applications that are compiled with the RENT, DLL, LONGNAME or IPA options. As of Version 2 Release 4, the prelinker was superseded by the binder. See also *binder*.

**preprocessor.** A phase of the compiler that examines the source program for preprocessor statements that are then executed, resulting in the alteration of the source program.

**preprocessor statement.** In the C and C++ languages, a statement that begins with the symbol # and is interpreted by the preprocessor during compilation. *IBM.*

**primary expression.** (1) An identifier, parenthesized expression, function call, array element specification, structure member specification, or union member specification. *IBM.* (2) Literals, names, and names qualified by the :: (scope resolution) operator.

**printable character.** One of the characters included in the print character classification of the LC\_CTYPE category in the current locale. *X/Open.*

**private.** Pertaining to a class member that is only accessible to member functions and friends of that class.

**process.** (1) An instance of an executing application and the resources it uses. (2) An address space and single thread of control that executes within that address space, and its required system resources. A process is created by another process issuing the fork() function. The process that issues the fork() function is known as the parent process, and the new process created by the fork() function is known as the child process. *X/Open. ISO.1.*

**process group.** A collection of processes that permits the signaling of related processes. Each process in the system is a member of a process group that is identified by the process group ID. A newly created process joins the process group of its creator. *IBM. X/Open. ISO.1.*

**process group ID.** The unique identifier representing a process group during its lifetime. A process group ID is a positive integer. (Under ISO only, it is a positive integer *that can be contained in a pid\_t.*) A process group ID will not be reused by the system until the process group lifetime ends. *X/Open. ISO.1.*

**process group lifetime.** A period of time that begins when a process group is created and ends when the last remaining process in the group leaves the group,

because either it is the end of the last process' lifetime or the last remaining process is calling the `setsid()` or `setpgid()` functions. *X/Open. ISO.1.*

**process ID.** The unique identifier representing a process. A process ID is a positive integer. (Under ISO only, it is a positive integer *that can be contained in a pid\_t.*) A process ID will not be reused by the system until the process lifetime ends. In addition, if there exists a process group whose process group ID is equal to that process ID, the process ID will not be reused by the system until the process group lifetime ends. A process that is not a system process will not have a process ID of 1. *X/Open. ISO.1.*

**process lifetime.** The period of time that begins when a process is created and ends when the process ID is returned to the system. After a process is created with a `fork()` function, it is considered active. Its thread of control and address space exist until it terminates. It then enters an inactive state where certain resources may be returned to the system, although some resources, such as the process ID, are still in use. When another process executes a `wait()` or `waitpid()` function for an inactive process, the remaining resources are returned to the system. The last resource to be returned to the system is the process ID. At this time, the lifetime of the process ends. *X/Open. ISO.1.*

**program object.** All or part of a computer program in a from suitable for loading into main storage for execution. A program object is the output of the OS/390 Binder and is a newer more flexible format (e.g. longer external names) than a load module.

**protected.** Pertaining to a class member that is only accessible to member functions and friends of that class, or to member functions and friends of classes derived from that class.

**prototype.** A function declaration or definition that includes both the return type of the function and the types of its parameters. See *function prototype*.

**public.** Pertaining to a class member that is accessible to all functions.

**pure virtual function.** A virtual function that has a function definition of `= 0;`. See also *abstract classes*.

## Q

**qualified class name.** Any class name or class name qualified with one or more `::` (scope resolution) operators.

**qualified name.** Used to qualify a nonclass type name such as a member by its class name.

**qualified type name.** Used to reduce complex class name syntax by using typedefs to represent qualified class names.

**Query Management Facility (QMF).** Pertaining to an IBM query and report writing facility that enables a variety of tasks such as data entry, query building, administration, and report analysis. *IBM.*

**queue.** A sequence with restricted access in which elements can only be added at the back end (or bottom) and removed from the front end (or top). A queue is characterized by first-in, first-out behavior and chronological order.

**quotation marks.** The characters `"` and `'`, also known as *double-quote* and *single-quote* respectively. *X/Open.*

## R

**radix character.** The character that separates the integer part of a number from the fractional part. *X/Open.*

**real group ID.** The attribute of a process that, at the time of process creating, identifies the group of the user who created the process. This value is subject to change during the process lifetime, as describe in `setgid()`. *X/Open. ISO.1.*

**real user ID.** The attribute of a process that, at the time of process creation, identifies the user who created the process. This value is subject to change during the process lifetime, as described in `setuid()`. *X/Open. ISO.1.*

**reason code.** A code that identifies the reason for a detected error. *IBM.*

**reassociation.** An optimization technique that rearranges the sequence of calculations in a subscript expression producing more candidates for common expression elimination.

**redirection.** In the shell, a method of associating files with the input or output of commands. *X/Open.*

**reentrant.** The attribute of a program or routine that allows the same copy of a program or routine to be used concurrently by two or more tasks.

**reference class.** A class that links a concrete class to an abstract class. Reference classes make polymorphism possible with the Collection Classes. Synonymous with *indirection class*.

**refresh.** To ensure that the information on the user's terminal screen is up-to-date. *X/Open.*

**register storage class specifier.** A specifier that indicates to the compiler within a block scope data definition, or a parameter declaration, that the object being described will be heavily used.

**register variable.** A variable defined with the register storage class specifier. Register variables have automatic storage.



**regular expression.** (1) A mechanism to select specific strings from a set of character strings. (2) A set of characters, meta-characters, and operators that define a string or group of strings in a search pattern. (3) A string containing wildcard characters and operations that define a set of one or more possible strings.

**regular file.** A file that is a randomly accessible sequence of bytes, with no further structure imposed by the system. *X/Open. ISO.1.*

**relation.** An unordered flat collection class that uses keys, allows for duplicate elements, and has element equality.

**relative path name.** The name of a directory or file expressed as a sequence of directories followed by a file name, beginning from the current directory. See *path name resolution. IBM.*

**reserved word.** (1) In programming languages, a keyword that may not be used as an identifier. *ISO-JTC1.* (2) A word used in a source program to describe an action to be taken by the program or compiler. It must not appear in the program as a user-defined name or a system name. *IBM.*

**RMODE (residency mode).** In MVS, a program attribute that refers to where a module is prepared to run. RMODE can be 24 or ANY. ANY refers to the fact that the module can be loaded either above or below the 16M line. RMODE 24 means the module expects to be loaded below the 16M line.

**runtime library.** A compiled collection of functions whose members can be referred to by an application program during runtime execution. Typically used to refer to a dynamic library that is provided in object code, such that references to the library are resolved during the linking step. The runtime library itself is not statically bound into the application modules.

## S

**saved set-group-ID.** An attribute of a process that allows some flexibility in the assignment of the effective group ID attribute, as described in the `exec()` family of functions and `setgid()`. *X/Open. ISO.1.*

**saved set-user-ID.** An attribute of a process that allows some flexibility in the assignment of the effective user ID attribute, as described in `exec()` and `setuid()`. *X/Open. ISO.1.*

**scalar.** An arithmetic object, or a pointer to an object of any type.

**scope.** (1) That part of a source program in which a variable is visible. (2) That part of a source program in which an object is defined and recognized.

**scope operator (::).** An operator that defines the scope for the argument on the right. If the left argument is blank, the scope is global; if the left argument is a class name, the scope is within that class. Synonymous with *scope resolution operator.*

**scope resolution operator (::).** Synonym for *scope operator.*

**semaphore.** An object used by multi-threaded applications for signalling purposes and for controlling access to serially reusable resources. Processes can be locked to a resource with semaphores if the processes follow certain programming conventions.

**sequence.** A sequentially ordered flat collection.

**sequential concatenation.** Multiple sequential data sets or partitioned data-set members are treated as one long sequential data set. In the case of sequential data sets, you can access or update the data sets in order. In the case of partitioned data-set members, you can access or update the members in order. Repositioning is possible if all of the data sets in the concatenation support repositioning.

**sequential data set.** A data set whose records are organized on the basis of their successive physical positions, such as on magnetic tape. *IBM.*

**session.** A collection of process groups established for job control purposes. Each process group is a member of a session. A process is a member of the session of which its process group is a member. A newly created process joins the session of its creator. A process can alter its session membership; see `setsid()`. There can be multiple process groups in the same session. *X/Open. ISO.1.*

**shell.** A program that interprets sequences of text input as commands. It may operate on an input stream or it may interactively prompt and read commands from a terminal. *X/Open.*

| This feature is provided as part of the OS/390 Shell  
| and Utilities feature licensed program.

**Short name.** An external non-C++ name in an object module produced by compiling with the `NOLONGNAME` option. Such a name is up to 8 characters long and single case.

**signal.** (1) A condition that may or may not be reported during program execution. For example, `SIGFPE` is the signal used to represent erroneous arithmetic operations such as a division by zero. (2) A mechanism by which a process may be notified of, or affected by, an event occurring in the system. Examples of such events include hardware exceptions and specific actions by processes. The term *signal* is also used to refer to the event itself. *X/Open. ISO.1.* (3) A method of interprocess communication that simulates software interrupts. *IBM.*

**signal handler.** A function to be called when the signal is reported.

**single-byte character set (SBCS).** A set of characters in which each character is represented by a one-byte code. *IBM.*

**single-precision.** Pertaining to the use of one computer word to represent a number in accordance with the required precision. *ISO-JTC1. ANSI/ISO.*

**single-quote.** The character `'`, also known as *apostrophe*. This character is named `<quotation-mark>` in the portable character set.

**slash.** The character `/`, also known as *solidus*. This character is named `<slash>` in the portable character set.

**socket.** (1) A unique host identifier created by the concatenation of a port identifier with a transmission control protocol/Internet protocol (TCP/IP) address. (2) A port identifier. (3) A 16-bit port-identifier. (4) A port on a specific host; a communications end point that is accessible through a protocol family's addressing mechanism. A socket is identified by a socket address. *IBM.*

**sorted map.** A sorted flat collection with key and element equality.

**sorted relation.** A sorted flat collection that uses keys, has element equality, and allows duplicate elements.

**sorted set.** A sorted flat collection with element equality.

**source module.** A file that contains source statements for such items as high-level language programs and data description specifications. *IBM.*

**source program.** A set of instructions written in a programming language that must be translated to machine language before the program can be run. *IBM.*

**space character.** The character defined in the portable character set as `<space>`. The space character is a member of the space character class of the current locale, but represents the single character, and not all of the possible members of the class. *X/Open.*

**spanned record.** A logical record contained in more than one block. *IBM.*

**specialization.** A user-supplied definition which replaces a corresponding template instantiation.

**specifiers.** Used in declarations to indicate storage class, fundamental data type and other properties of the object or function being declared.

**spill area.** A storage area used to save the contents of registers. *IBM.*

**SQL (Structured Query Language).** A language designed to create, access, update and free data tables.

**square brackets.** The characters `[` (left bracket) and `]` (right bracket). Also see *brackets*.

**stack frame.** The physical representation of the activation of a routine. The stack frame is allocated and freed on a LIFO (last in, first out) basis. A stack is a collection of one or more stack segments consisting of an initial stack segment and zero or more increments.

**stack storage.** Synonym for *automatic storage*.

**standard error.** An output stream usually intended to be used for diagnostic messages. *X/Open.*

**standard input.** (1) An input stream usually intended to be used for primary data input. *X/Open.* (2) The primary source of data entered into a command. Standard input comes from the keyboard unless redirection or piping is used, in which case standard input can be from a file or the output from another command. *IBM.*

**standard output.** (1) An output stream usually intended to be used for primary data output. *X/Open.* (2) The primary destination of data coming from a command. Standard output goes to the display unless redirection or piping is used, in which case standard output can go to a file or to another command. *IBM.*

**statement.** An instruction that ends with the character `;` (semicolon) or several instructions that are surrounded by the characters `{` and `}`.

**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.

**static binding.** The act of resolving references to external variables and functions before run time.

**storage class specifier.** One of the terms used to specify a storage class, such as *auto*, *register*, *static*, or *extern*.

**stream.** (1) A continuous stream of data elements being transmitted, or intended for transmission, in character or binary-digit form, using a defined format. (2) A file access object that allows access to an ordered sequence of characters, as described by the ISO C standard. Such objects can be created by the `fdopen()` or `fopen()` functions, and are associated with a file descriptor. A stream provides the additional services of user-selectable buffering and formatted input and output. *X/Open.*

**string.** A contiguous sequence of bytes terminated by and including the first null byte. *X/Open*.

**string constant.** Zero or more characters enclosed in double quotation marks.

**string literal.** Zero or more characters enclosed in double quotation marks.

**striped data set.** A special data set organization that spreads a data set over a specified number of volumes so that I/O parallelism can be exploited. Record  $n$  in a striped data set is found on a volume separate from the volume containing record  $n - p$ , where  $n > p$ .

**struct.** An aggregate of elements having arbitrary types.

**structure.** A construct (a class data type) that contains an ordered group of data objects. Unlike an array, the data objects within a structure can have varied data types. A structure can be used in all places a class is used. The initial projection is public.

**structure tag.** The identifier that names a structure data type.

**Structured Query Language.** See *SQL*.

**stub routine.** A routine, within a runtime library, that contains the minimum lines of code required to locate a given routine at run time.

**subprogram.** In the IPA Link version of the Inline Report listing section, an equivalent term for 'function'.

**subscript.** One or more expressions, each enclosed in brackets, that follow an array name. A subscript refers to an element in an array.

**subsystem.** A secondary or subordinate system, usually capable of operating independently of or asynchronously with, a controlling system. *ISO Draft*.

**subtree.** A tree structure created by arbitrarily denoting a node to be the root node in a tree. A subtree is always part of a whole tree.

**superset.** Given two sets A and B, A is a superset of B if and only if all elements of B are also elements of A. That is, A is a superset of B if B is a subset of A.

**support.** In system development, to provide the necessary resources for the correct operation of a functional unit. *IBM*.

**switch expression.** The controlling expression of a switch statement.

**switch statement.** A C or C++ language statement that causes control to be transferred to one of several statements depending on the value of an expression.

**system default.** A default value defined in the system profile. *IBM*.

**System Object Model (SOM).** Defines an IBM interface between programs, or between libraries and programs, so that an object's interface is separated from its implementation. SOM allows classes of objects to be defined in one programming language and used in another, and it allows libraries of such classes to be updated without requiring client code to be recompiled. *IBM*.

**system process.** (1) An implementation-dependent object, other than a process executing an application, that has a process ID. *X/Open*. (2) An object, other than a process executing an application, that is defined by the system, and has a process ID. *ISO.1*.

## T

**tab character.** A character that in the output stream indicates that printing or displaying should start at the next horizontal tabulation position on the current line. The tab is the character designated by '\t' in the C language. If the current position is at or past the last defined horizontal tabulation position, the behavior is unspecified. It is unspecified whether the character is the exact sequence transmitted to an output device by the system to accomplish the tabulation. *X/Open*.

This character is named <tab> in the portable character set.

**task library.** A class library that provides the facilities to write programs that are made up of tasks.

**template.** A family of classes or functions with variable types.

**template class.** A class instance generated by a class template.

**Template Declaration.** A prototype of a template which can optionally include a template definition.

**Template Definition.** A blueprint the compiler uses to generate a template instantiation.

**template function.** A function generated by a function template.

**Template Instantiation.** Compiler generated code for a class or function using the referenced types and the corresponding class or function template definition.

**terminals.** Synonym for *leaves*.

**text file.** A file that contains characters organized into one or more lines. The lines must not contain NUL characters and none can exceed {LINE\_MAX}—which is defined in *limits.h*—bytes in length, including the

new-line character. The term *text file* does not prevent the inclusion of control or other unprintable characters (other than NUL). *X/Open*.

**thread.** The smallest unit of operation to be performed within a process. *IBM*.

**throw expression.** An argument to the C++ exception being thrown.

**tilde.** The character ~. This character is named <tilde> in the portable character set.

**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. *IBM*.

**traceback.** A section of a dump that provides information about the stack frame, the program unit address, the entry point of the routine, the statement number, and the status of the routines on the call-chain at the time the traceback was produced.

**trigraph sequence.** An alternative spelling of some characters to allow the implementation of C in character sets that do not provide a sufficient number of non-alphabetic graphics. *ANSI/ISO*.

Before preprocessing, each trigraph sequence in a string or literal is replaced by the single character that it represents.

**truncate.** To shorten a value to a specified length.

**try block.** A block in which a known C++ exception is passed to a handler.

**type conversion.** Synonym for *boundary alignment*.

**type definition.** A definition of a name for a data type. *IBM*.

**type specifier.** Used to indicate the data type of an object or function being declared.

## U

**ultimate consumer.** The target of data in an I/O 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 I/O 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. *IBM*.

**undefined behavior.** Action by the compiler and library when the program uses erroneous constructs or contains erroneous data. Permissible undefined behavior includes ignoring the situation completely

with unpredictable results. It also includes behaving in a documented manner that is characteristic of the environment, during translation or program execution, with or without issuing a diagnostic message. It can also include terminating a translation or execution, while issuing a diagnostic message. Contrast with *unspecified behavior* and *implementation-defined behavior*.

**underflow.** (1) A condition that occurs when the result of an operation is less than the smallest possible nonzero number. (2) Synonym for arithmetic underflow, monadic operation. *IBM*.

**union.** (1) In the C or C++ language, a variable that can hold any one of several data types, but only one data type at a time. *IBM*. (2) For bags, there is an additional rule for duplicates: If bag P contains an element *m* times and bag Q contains the same element *n* times, then the union of P and Q contains that element *m+n* times.

**union tag.** The identifier that names a union data type.

**unnamed pipe.** A pipe that is accessible only by the process that created the pipe and its child processes. An unnamed pipe does not have to be opened before it can be used. It is a temporary file that lasts only until the last file descriptor that uses it is closed.

**unique collection.** A collection in which the value of an element only occurs once; that is, there are no duplicate elements.

**unrecoverable error.** An error for which recovery is impossible without use of recovery techniques external to the computer program or run.

**unspecified behavior.** Action by the compiler and library when the program uses correct constructs or data, for which the standards impose no specific requirements. Such action should not cause compiler or application failure. You should not, however, write any programs to rely on such behavior as they may not be portable to other systems. Contrast with *implementation-defined behavior* and *undefined behavior*.

**user-defined data type.** (1) A mathematical model that includes a structure for storing data and operations that can be performed on that data. Common abstract data types include sets, trees, and heaps. (2) See also *abstract data type*.

**user ID.** A nonnegative integer that is used to identify a system user. (Under ISO only, a nonnegative integer, which can be contained in an object of type *uid\_t*.) When the identity of a user is associated with a process, a user ID value is referred to as a real user ID, an effective user ID, or (under ISO only, and there optionally) a saved set-user ID. *X/Open. ISO.1*.

**user name.** A string that is used to identify a user. *ISO.1*.



**user prefix.** In an MVS environment, the user prefix is typically the user's logon user identification.

## V

**value numbering.** An optimization technique that involves local constant propagation, local expression elimination, and folding several instructions into a single instruction.

**variable.** In programming languages, a language object that may take different values, one at a time. The values of a variable are usually restricted to a certain data type. *ISO-JTC1*.

**variant character.** A character whose hexadecimal value differs between different character sets. On EBCDIC systems, such as S/390, these 13 characters are an exception to the portability of the portable character set.

<left-square-bracket>	[
<right-square-bracket>	]
<left-brace>	{
<right-brace>	}
<backslash>	\
<circumflex>	^
<tilde>	~
<exclamation-mark>	!
<number-sign>	#
<vertical-line>	
<grave-accent>	`
<dollar-sign>	\$
<commercial-at>	@

**vertical-tab character.** A character that in the output stream indicates that printing should start at the next vertical tabulation position. The vertical-tab is the character designated by '\v' in the C or C++ languages. If the current position is at or past the last defined vertical tabulation position, the behavior is unspecified. It is unspecified whether this character is the exact sequence transmitted to an output device by the system to accomplish the tabulation. *X/Open*. This character is named <vertical-tab> in the portable character set.

**virtual address space.** (1) In virtual storage systems, the virtual storage assigned to a batched or terminal job, a system task, or a task initiated by a command. (2) In VSE, a subdivision of the virtual address area available to the user for the allocation of private, non-shared partitions.

**virtual function.** A function of a class that is declared with the keyword *virtual*. The implementation that is executed when you make a call to a virtual function depends on the type of the object for which it is called, which is determined at run time.

**Virtual Storage Access Method (VSAM).** An access method for direct or sequential processing of fixed and variable length records on direct access devices. The records in a VSAM data set or file can be organized in

logical sequence by a key field (key sequence), in the physical sequence in which they are written on the data set or file (entry-sequence), or by relative-record number.

**visible.** Visibility of identifiers is based on scoping rules and is independent of *access*.

**volatile attribute.** (1) In the C or C++ language, the keyword *volatile*, used in a definition, declaration, or cast. It causes the compiler to place the value of the data object in storage and to reload this value at each reference to the data object. *IBM*. (2) An attribute of a data object that indicates the object is changeable. Any expression referring to a volatile object is evaluated immediately (for example, assignments).

## W

**while statement.** A looping statement that contains the keyword *while* followed by an expression in parentheses (the condition) and a statement (the action). *IBM*.

**white space.** (1) Space characters, tab characters, form-feed characters, and new-line characters. (2) A sequence of one or more characters that belong to the space character class as defined via the LC\_CTYPE category in the current locale. In the POSIX locale, white space consists of one or more blank characters (space and tab characters), new-line characters, carriage-return characters, form-feed characters, and vertical-tab characters. *X/Open*.

**wide-character.** A character whose range of values can represent distinct codes for all members of the largest extended character set specified among the supporting locales.

**wide-character code.** An integral value corresponding to a single graphic symbol or control code. *X/Open*.

**wide-character string.** A contiguous sequence of wide-character codes terminated by and including the first null wide-character code. *X/Open*.

**wide-oriented stream.** See *orientation of a stream*.

**working directory.** Synonym for *current working directory*.

**writable static area.** See *WSA*.

**write.** (1) To output characters to a file, such as standard output or standard error. Unless otherwise stated, standard output is the default output destination for all uses of the term *write*. *X/Open*. (2) To make a permanent or transient recording of data in a storage device or on a data medium. *ISO-JTC1*. *ANSI/ISO*.

**WSA (writable static area).** An area of memory in the program that is modifiable during program execution.

Typically, this area contains global variables and function and variable descriptors for DLLs.

---

## Bibliography

This bibliography lists the publications for IBM products that are related to the OS/390 C/C++ product. It includes publications covering the application programming task. The bibliography is not a comprehensive list of the publications for these products, however, it should be adequate for most OS/390 C/C++ users. Refer to the *OS/390 Information Roadmap*, GC28-1727, for a complete list of publications belonging to the OS/390 product.

Related publications not listed in this section can be found on the IBM Online Library Omnibus Edition: MVS Collection CD-ROM (SK2T-0710), the *IBM Online Library Omnibus Edition: OS/390 Collection* CD-ROM (SK2T-6700), or on a tape available with OS/390.

---

### OS/390

- *OS/390 Printing Softcopy BOOKs*, S544-5354
- *OS/390 Introduction and Release Guide*, GC28-1725
- *OS/390 Planning for Installation*, GC28-1726
- *OS/390 Summary of Message Changes*, GC28-1499
- *OS/390 Information Roadmap*, GC28-1727

---

### VS COBOL II Release 4

- *General Information*, GC26-4042
- *Migration Guide for MVS and CMS*, GC26-3151
- *Installation and Customization for MVS*, SC26-4048
- *Application Programming Guide for MVS and CMS*, SC26-4045
- *Application Programming Language Reference*, GC26-4047
- *Application Programming Reference Summary*, SX26-3721
- *Application Programming Debugging*, SC26-4049
- *Application Programming Diagnosis Guide*, LY27-9523
- *Application Programming Diagnosis Reference*, LY27-9522

---

### COBOL FOR MVS & VM Release 2

- *Compiler and Run-Time Migration Guide*, GC26-4764
- *Programming Guide*, SC26-4767
- *Language Reference*, SC26-4769
- *Diagnosis Guide*, SC26-3138
- *Licensed Program Specifications*, GC26-4761
- *Installation and Customization under MVS*, SC26-4766

---

## **COBOL for OS/390 & VM Version 2 Release 1**

- *Compiler and Run-Time Migration Guide*, GC26-4764
- *Programming Guide*, SC26-9049
- *Language Reference*, SC26-9046
- *Diagnosis Guide*, GC26-9047
- *Licensed Program Specifications*, GC26-9044
- *Installation and Customization under OS/390*, GC26-9045
- *Program Directory for VM*
- *Fact Sheet*, GC26-9048

---

## **PL/I for MVS & VM Release 1 Modification 1**

- *Language Reference*, SC26-3114
- *Compiler and Run-Time Migration Guide*, SC26-3118
- *Programming Guide*, SC26-3113
- *Compile-Time Messages and Codes*, SC26-3229
- *Reference Summary*, SX26-3821
- *Diagnosis Guide*, SC26-3149
- *Installation and Customization under MVS*, SC26-3119
- *Licensed Program Specifications*, GC26-3116

---

## **OS PL/I Version 2 Release 3**

- *Programming Guide*, SC26-4307
- *Programming: Language Reference*, SC26-4308
- *Programming: Messages and Codes*, SC26-4309

---

## **VS FORTRAN Version 2 Release 6**

- *Programming Reference*, SC26-4221
- *Programming Guide*, SC26-4222

---

## **CICS/ESA Version 4 Release 1**

- *Application Programming Reference*, SC33-1170
- *Application Programming Guide*, SC33-1169
- *Installation Guide*, SC33-1163
- *System Definition Guide*, SC33-1164
- *Resource Definition Guide*, SC33-1166
- *Messages and Codes*, SC33-1177

---

## **CICS Transaction Server for OS/390 Release 2**

- *Application Programming Guide*, SC33-1687
- *Application Programming Reference*, SC33-1688
- *System Programming Reference*, SC33-1689
- *Distributed Transaction Programming Guide*, SC33-1691
- *Front End Programming Interface User's Guide*, SC33-1692



---

## **DB2 Version 3 Release 1**

- *SQL Reference*, SC26-4890
- *Reference Summary*, SX26-3801
- *Command and Utility Reference*, SC26-4891
- *Application Programming and SQL Guide*, SC26-4889

---

## **DB2 Version 4 Release 1**

- *SQL Reference*, SC26-3270
- *Reference Summary*, SX26-3829
- *Command Reference*, SC26-3267
- *Application Programming and SQL Guide*, SC26-3266
- *Utility Guide and Reference*, SC26-3395

---

## **DB2 Version 5 Release 1**

- *Administration Guide*, SC26-8957
- *Application Programming and SQL Guide*, SC26-8958
- *Call Level Interface Guide and Reference*, SC26-8959
- *Command Reference*, SC26-8960
- *Data Sharing: Planning and Administration*, SC26-8961
- *Installation Guide*, GC26-8970
- *Messages and Codes*, GC26-8979
- *SQL Reference*, SC26-8966
- *Reference for Remote DRDA Requesters and Servers*, SC26-8964
- *Utility Guide and Reference*, SC26-8967

---

## **IMS/ESA Version 4 Release 1**

- *Application Programming: Design Guide*, SC26-3066
- *Application Programming: DL/I Calls*, SC26-3062
- *Application Programming: Data Communication*, SC26-3058
- *Application Programming: EXEC DL/I Commands*, SC26-3063

---

## **IMS/ESA Version 5 Release 1**

- *Application Programming: Design Guide*, SC26-8016
- *Application Programming: Transaction Manager*, SC26-8017
- *Application Programming: Database Manager*, SC26-8015
- *Application Programming: EXEC DL/I Commands for CICS and IMS*, SC26-8018

---

## **IMS/ESA Version 6 Release 1**

- *Application Programming: Design Guide*, SC26-8728
- *Application Programming: Transaction Manager*, SC26-8729
- *Application Programming: Database Manager*, SC26-8727
- *Application Programming: EXEC DL/I Commands for CICS and IMS*, SC26-8726

---

## QMF Version 3 Release 2

- *Introducing QMF*, GC26-4713
- *Using QMF*, SC26-8078
- *Developing QMF Applications*, SC26-4722
- *Reference*, SC26-4716
- *Managing QMF for MVS*, SC26-8218
- *Reference*, SC26-4716
- *Messages and Codes*, SC26-4834
- *Installing on MVS*, SC26-4719

---

## VSAM

- *MVS/ESA VSAM Catalog Administration: Access Method Services Reference*, SC26-4501
- *MVS/ESA VSAM Administration: Macro Instruction Reference*, SC26-4517
- *MVS/ESA VSAM Administration Guide for MVS/DFP*, SC26-4518
- *MVS/ESA Integrated Catalog Administration: Access Method Services Reference*, SC26-4500
- *DFSMS/MVS Access Method Services for VSAM*, SC26-4905
- *MVS/DFP Access Method Services for VSAM Catalogs*, SC26-4570
- *MVS/Extended Architecture VSAM Catalog Administration: Access Method Services Reference (Data Facility Product, Version 2)*, GC26-4136

---

# INDEX

## Special Characters

[ ] array subscript operators 140  
-- decrement operator 143  
?: conditional operators 160  
/= compound assignment operator 164  
\*= compound assignment operator 164  
&= compound assignment operator 164  
+= compound assignment operator 164  
\  
continuation character 65, 220  
== equal to operator 156  
\  
escape character 67  
++ increment operator 142  
&& logical AND operator 158  
!= not equal to operator 156  
?? trigraphs 53  
+ addition operator 153  
& address operator 144  
\_ANSI\_ macro 231  
\_BFP\_ macro 231  
& bitwise AND operator 157  
\_cdecl 123  
, comma operator 165  
/ division operator 153  
" double quotation mark 65  
\_EXTENDED\_ macro 231  
\* indirection operator 145  
! logical negation operator 144  
\* multiplication operator 152  
# preprocessor directive character 220  
# preprocessor operator 225  
= simple assignment operator 162  
~ bitwise negation operator 144  
^ (caret)  
    locale 52  
^ = compound assignment operator 164  
^ bitwise exclusive OR operator 157  
>= greater than or equal to operator 155  
>>= compound assignment operator 164  
>> right-shift operator 154  
> greater than operator 155  
<= less than or equal to operator 155  
<<= compound assignment operator 164  
<< left-shift operator 154  
< less than operator 155  
\_LONG\_LONG macro 236  
| (vertical bar)  
    locale 52

## Numerics

370 macro 236

## A

abort function 395  
abstract classes 363  
access  
    base classes 350  
    constructors 325  
    declarations 291, 351

access (*continued*)  
    derived class 349  
    effective 354  
    exception handling 387  
    friends 309  
    inherited member 349  
    members 304  
    multiple 357  
    private 350  
    protected members 350  
    public 350  
    resolution 353  
    specifiers 305, 348, 350  
    summary example 355  
    virtual function 362  
accessibility 304, 353, 357  
additive operators  
    addition + 153  
    subtraction - 154  
address operator & 144  
aggregate classes  
    constructors and destructors 325  
    description 282  
aggregate operands 135  
ALIAS compile option 262  
alignment rules  
    changing with #pragma pack 267  
allocation expressions 147  
ambiguities  
    base classes 356, 358  
    conversions 169  
    resolving 205  
    virtual functions 361  
AND operator (bitwise) & 157  
AND operator (logical) && 158  
anonymous  
    unions 117  
    unions in C 116  
ANSI  
    implementation-defined behavior 411  
ANSI flagging 259  
ANSI/ISO 411  
ANSI macro 231  
argc (argument count) 184  
    argument to main 184  
    in different environments 40  
argument count (argc) 184  
argument-matching conversions 313  
argument vector (argv) 184  
arguments  
    best-matching 313  
    command line 35  
    const 314  
    default 190, 326  
    default initializers in templates 366  
    matching 312  
    pass by reference 188  
    passing 40  
    template  
        matching 374, 376  
        nontype 371

arguments (*continued*)  
    template (*continued*)  
        type 369  
    to main function 35  
    trivial conversions 314, 374  
    virtual functions 360  
    volatile 314  
arguments in a function call 185  
arguments to main  
    envp (environment pointer) 184  
argv (argument vector) 184  
    argument to main 184  
    in different environments 40  
    restriction 40  
arithmetic  
    conversions 170  
    operands 135  
arrays  
    ANSI/ISO conformance 414  
    class members 292  
    declaring 100  
    operands 135  
    subscripting operator  
        overloading 320  
    subscripting operators 140  
arrow operator 141  
ASCII character codes 67  
assignment  
    memberwise 319  
    operators  
        copying classes 341  
        default for classes 340  
        overloading 319  
assignment expression  
    compound 164  
    description 162  
    simple 162  
associativity of operators 133  
auto storage class specifier 73

## B

base classes  
    abstract 363  
    access 349, 350  
    ambiguities 356, 358  
    description 346  
    direct 345, 356  
    indirect 345, 356  
    initialization 336, 338  
    multiple 356  
    multiple access 357  
    pointers to 348, 353, 360  
    virtual 357, 359  
base list 346, 348, 356  
base specifier 348  
best-matching arguments 313  
BFP macro 231  
binary expression 152  
binary operators  
    addition operator + 153

- binary operators (*continued*)
    - bitwise AND operator & 157
    - bitwise inclusive OR operator | 158
    - bitwise OR operator ^ 157
    - description 152
    - division operator / 153
    - equality operator == 156
    - greater than operator > 155
    - inequality operator != 156
    - left-shift operator << 154
    - less than operator < 155
    - logical AND operator && 158
    - logical OR operator || 159
    - multiplication operator \* 152
    - overloading 318
    - remainder operator % 153
    - right-shift operator >> 154
    - subtraction operator (-) 154
  - binding
    - dynamic 346
    - static 346
    - virtual functions 359
  - bit fields 110, 292
    - ANSI/ISO conformance 414
  - bitwise operators
    - AND & 157
    - exclusive OR ^ 157
    - inclusive OR | 158
    - left-shift << 154
    - negation operator ~ 144
    - right-shift >> 154
  - block
    - scope 46
    - visibility 47
  - block scope
    - in C 35
  - block scope data declarations
    - auto 73
    - description 70
    - register 81
    - static 82
  - block statement 198
  - boundaries, data 129
  - brackets [ ] 140
  - break statement 200
- C**
- C and C++ differences 401
  - call, function 185
  - call scope 353
  - carriage return escape sequence \r 67
  - case label 213
  - cast
    - argument-matching conversions 313
    - expressions 145
    - function style 328
  - catch
    - argument matching 387
    - exceptions 387
    - keyword 382
    - no match 395
  - char constant 64
  - char type specifier 86
  - character
    - constant 64
    - constants 64
  - character (*continued*)
    - data types 86
    - set 51
    - string constant 65
  - characters
    - escape sequence 67
  - chars pragma 247
  - checkout pragma 248
  - class key 282
  - class member access operators
    - argument-matching conversions 314
    - description 141
    - overloading 320
  - class member lists 291
  - class members 291
  - class names
    - description 283
    - scope 286
  - class object 72
  - class operands 135
  - class scope 47
  - class templates
    - declarations 370
    - definitions 370
    - description 369
    - friends 379
    - instantiation 369
    - member functions 377
    - static members 380
  - classes 343, 345 (*continued*)
    - abstract 363
    - aggregate 282, 325
    - class templates 369
    - class-type members 292
    - constructors 325
    - conversions 334
    - copying
      - by assignment 341
      - by initialization 341
      - restrictions 340
    - declarations 282
    - derivation 346
    - destructors 325
    - friends 306
    - incomplete declarations 287, 292, 346
    - inheritance 343
    - initialization 336
    - local 288
    - member access 304
    - member functions 293
    - member lists 291
    - member scope 295
    - members 291
    - nested 287, 308, 352
    - objects 72
    - overloading
      - functions 311
      - operators 315, 317
    - overview 281
    - packing
      - using #pragma pack 267
    - scope 286
    - special member functions 325
    - static members 300
    - this pointer 298
    - virtual 359
    - virtual base 357
  - classes 343, 345 (*continued*)
    - virtual member functions 359
  - COBOL linkage 261
  - CODESET macro 232
  - comma operator 165
  - command-line arguments
    - main function 35
    - passing 40
  - comment pragma 248
  - comments 54
  - COMMONC macro 232
  - COMPAT macro 232
  - compatible types 161
  - COMPILER\_VER macro 232
  - complete class name 348
  - compound statement 198
  - conditional compilation
    - description 237
    - elif preprocessor directive 238
    - if preprocessor directive 238
    - ifdef preprocessor directive 239
    - ifndef preprocessor directive 239
  - conditional expression ?: 160
  - conditional statements
    - if 209
    - switch 213
  - conforming to ANSI/ISO 411
  - conforming to POSIX 409
  - const
    - arguments 314
    - qualifier 120
  - constant
    - character 64
    - member functions 293
    - string 65
  - constant expression 138
  - constants
    - character 64
    - description 60
    - enumeration 91
    - escape sequence 67
    - escape sequences 67
    - fixed-point decimal 63
    - floating-point 62
    - integer 60
    - string 65
  - construction order
    - of class objects 327
    - of derived class objects 339
  - constructors
    - construction order 327, 339
    - conversion by 335
    - copy 327, 340, 341
    - default 326, 338
    - derived class objects 339
    - description 326
    - exception handling 391
    - explicitly constructing objects 328
    - initialization by 336
    - initializer 338
    - overview 325
    - templates 370, 378
    - temporary objects 333
    - virtual 325
  - continuation character 65, 220
  - continue statement 202

- control statements
  - break 200
  - continue 202
  - goto 208
  - return 211
- conversion functions 335
- conversions
  - ambiguous 169
  - arguments 313
  - arithmetic 170
  - cast 145
  - derived class 359
  - sequence 313
  - standard 167
  - trivial 314
  - user-defined
    - by constructor 335
    - conversion functions 335
- convlit pragma 249
- copy constructors 327, 340, 341
- copy restrictions 340
- copying
  - class objects 340
- copying class objects 340
- csect pragma 250

## D

- data
  - abstraction 42
  - hiding 43
- data members
  - description 292
  - scope 295
  - static 302
- DATE macro 230
- deallocation expression 151
- decimal
  - data type operators 147
- decimal constant 61
- decimal data type
  - operators 147
- declarations
  - access 351
  - class
    - description 282
    - incomplete 287, 292, 346
    - syntax 282
  - class member 291
  - class templates 370
  - description 69
  - file scope 70, 71
  - friend 306, 309
  - function 174
    - matching 312
    - resolving ambiguities 205
  - function templates 376
  - in source files 32
  - matching 312
  - parameter 181
  - pointers to members 297
  - resolving ambiguous statements 205
  - template functions 376
- declarators
  - array 100
  - character 86
  - description 119

- declarators (*continued*)
  - floating-point 87
  - integer 90
  - member 291
  - pointer 95
  - restrictions 415
  - union 115
- decrement operator -- 321
- decrement operator (--) 143
- default
  - arguments 325, 326
  - assignment operator 340
  - constructors 326, 338
  - copy constructors 341
  - initializers in templates 366
  - member access 305
- default clause 213
- default label 213
- define pragma 251
- define preprocessor directive 221
- defined, preprocessor operator 238
- defined unary operator 238
- definitions
  - class templates 370
  - function templates 376
  - in source files 32
  - macro 221
  - member function 293
  - template classes 370
  - template functions 376
- delete operator
  - description 151
  - free store 330
  - overloading 322
- dereferencing operator 145
- derived classes
  - access 349
  - access declarations 351
  - base list 346, 348
  - catch block 388
  - construction order 339
  - description 346
  - initialization 336
  - pointers to 348, 353, 360
  - syntax 348
- destructors
  - description 328
  - destruction order 329
  - exception handling 391
  - overview 325
  - thrown objects 329
  - virtual 325, 329
- diagnostic messages 416
- differences between C and C++ 401
- digitsof operator 147
- digraph sequences 53
- direct base class 345, 356
- directives 32
- disjoint pragma 251
- division operator (/) 153
- DLL macro 232
- DLLs (Dynamic Link Libraries)
  - #pragma export 253
- do statement 203
- dominant names 348
- dot operator 141

- double byte character set (DBCS)
  - comments 55
- double precision
  - constants 62
  - variables 87
- double type specifier 87
- dynamic binding
  - in object-oriented programming 44
  - virtual functions 346

## E

- EBCDIC character codes 67
- effective access 354
- elaborated type specifier 286
- elif preprocessor directive 238
- ellipsis
  - argument-matching conversions 313
  - function call operator 320
  - in overloaded operator 317
  - type checking 182
  - user-defined conversions 313
- else clause 209
- else preprocessor directive 240
- empty argument list 182
- encapsulation 43
- end of string 65
- endif preprocessor directive 240
- enum 90
- enumeration
  - ANSI/ISO conformance 414
  - enum data types 90
  - enumeration constant 91
- enumeration operands 135
- enumerator 91
- environment
  - implementation-defined
    - behavior 419
  - pragma 252
- environment pointer argument
  - (envp) 184
- envp (environment pointer
  - argument) 184
- equal to operator == 156
- equality operators
  - equal to == 156
  - not equal to != 156
- error
  - message classes 416
- error handling
  - ANSI/ISO conformance 416
- error message classes 416
- error messages 242
- error preprocessor directive 228
- escape character \ 67
- escape sequence 67, 412
- evaluation, expression 133
- examples
  - cbc3raa 201
  - cbc3raa1 199
  - cbc3raa2 200
  - cbc3raa3 202
  - cbc3raa4 203
  - cbc3raa6 209
  - cbc3raa7 217
  - cbc3raa8 224
  - cbc3raa9 224

## examples (continued)

- cbc3raaa 33
- cbc3raab 34
- cbc3raaf 74
- cbc3raag 75
- cbc3raai 80
- cbc3raak 83
- cbc3raam 100
- cbc3raan 93
- cbc3raao 105
- cbc3raap 106
- cbc3raaq 98
- cbc3raas 112
- cbc3raat 180
- cbc3raau 183
- cbc3raav 176
- cbc3raaw 177
- cbc3raax 187
- cbc3raay 188
- cbc3rabc 241
- cbc3rabd 242
- cbc3rabe 256
- cbc3rabg 416
- cbc3rabi 215
- cbc3rah1 79
- cbc3rah2 79
- cbc3rah3 79
- cbc3raj1 83
- cbc3raj2 83
- cbc3rmax 34
- cbc3x02d 46
- cbc3x02f 48
- cbc3x02g 48
- cbc3x02h 49
- cbc3x02i 49
- cbc3x02j 50
- cbc3x02k 66
- cbc3x02l 68
- cbc3x06a 189
- cbc3x06b 190
- cbc3x06c 140
- cbc3x07e 204
- cbc3x08a 236
- cbc3x08b 236
- cbc3x08c 237
- cbc3x10a 305
- cbc3x10b 284
- cbc3x10c 282
- cbc3x10d 282
- cbc3x10e 286
- cbc3x11a 295
- cbc3x11b 297
- cbc3x11c 298
- cbc3x11d 299
- cbc3x11e 301
- cbc3x11f 302
- cbc3x11g 303
- cbc3x11h 304
- cbc3x11i 306
- cbc3x11j 307
- cbc3x12a 311
- cbc3x12b 315
- cbc3x12c 317
- cbc3x12d 321
- cbc3x12e 322
- cbc3x13a 337
- cbc3x14a 347

## examples (continued)

- cbc3x14b 347
- cbc3x14c 348
- cbc3x14d 352
- cbc3x14e 353
- cbc3x14f 355
- cbc3x14g 358
- cbc3x15a 367
- cbc3x15b 377
- cbc3x16a 385
- cbc3x16b 389
- cbc3x16c 390
- cbc3x16d 391
- cbc3x16f 386
- machine-readable 9
- naming of 9
- softcopy 9
- exception handling
  - access 387
  - catching exceptions 387
  - constructors 391
  - destructors 391
  - exception specifications
    - empty 394
    - unexpected 394
  - order of catching 388
  - overview 381
  - resumption model 384
  - rethrowing exceptions 389
  - special functions 395
  - syntax 382
  - termination model 384
  - throw 329
  - transferring control 384
- exception specification syntax 393
- exclusive OR operator (bitwise) ^ 157
- explicit definitions
  - member function templates 377
  - template classes 373, 380
  - template functions 375
- explicit initialization 336
- explicit type conversions 145
- exponent 63
- export1 125
- exporting functions
  - with #pragma export 253
- expressions
  - allocation 147
  - assignment 162
  - binary 152
  - cast 145
  - comma 165
  - conditional 160
  - constant 138
  - deallocation 151
  - description 133
  - evaluation of 133
  - list 336
  - lvalue 136
  - parenthesized 137
  - pointer-to-member 160
  - primary 136
  - resolving ambiguous statements 205
  - statement 205
  - throw 152
  - unary 142
- EXTENDED macro 231

- extern declaration 76
- extern storage class specifier 75
- external
  - linkage 38
  - names
    - length of 59
    - long name support 59
    - mapping 58
  - static 78
- external declaration 34
- external linkage 38
- extraction operator 48

## F

- FETCHABLE preprocessor directive 260
- field, bit 110
- file inclusion 228
- FILE macro 230
- file scope 47, 377
  - in C 36
- file scope data declarations
  - description 71
  - extern 75
  - static 82
- files
  - implementation defined
    - behavior 418
- FILETAG macro 232
- filetag pragma 253
- fixed-point decimal
  - constant 63
  - data type 88
- float type specifier 87, 90
- floating point
  - constant 62
- floating-point
  - conversions 168
  - range 413
  - storage 413
- floating-point variables
  - double 87
  - float 87
  - long double 87
- for statement 206
- formal exception handling 382
- FORTRAN linkage 261
- free store
  - delete operator 151
  - description 330
  - new operator 147
- friends
  - access 309
  - description 306
  - member functions 293
  - nested classes 308
  - scope 308
  - templates 379
  - virtual functions 361
- function call operator 319
- function declarator 180
- function-like macro 222
- FUNCTION macro 232
- function scope 47
  - in C 36
- function style cast
  - constructing an object 328



- function templates
  - description 373
  - friends 379
  - members 377
- functions
  - argument conversions 314
  - arguments 313
  - body 182
  - calling functions 139, 185
  - conversion 335
  - declarations 174
  - declarator 180
  - default arguments 190
  - definitions 178
  - exception specifications 393
  - friend 306
  - inline 195
  - main 184
  - operator delete() 330
  - operator new() 330
  - overloading 311
  - overview 173
  - parameter 185
  - pointers to 193
  - polymorphic 346
  - prototypes 178
  - return statements 211
  - return values 192
  - specifiers 129, 195
  - template 373
  - virtual 294, 346, 359, 361
  - void 176
- functions, main 35

## G

- global variables 75
- goto statement 208
- greater than operator > 155
- greater than or equal to operator >= 155

## H

- handler list 382
- hdrstop pragma 254
- hexadecimal constant 61
- hexadecimal numbers as escape sequences 67
- HHW370 macro 233
- hidden
  - names 137, 285, 286
  - virtual functions 360
- HOSMVS macro 233

## I

- IBMC macro 233
- IBMCPM macro 233
- identifier linkage 37
- identifier names
  - limit 59
- identifiers 56
  - ANSI/ISO conformance 411
- identifiers in OS/390 C/C++
  - external names 58
- if preprocessor directive 238

- if statement 209
- ifdef preprocessor directive 239
- ifndef preprocessor directive 239
- implementation-defined behavior 411
- implementation dependency
  - allocation of floating-point types 87
  - allocation of integral types 89
  - bit field length 110
  - class member allocation 292
  - order of argument passing 186
  - sign of char 86
  - size\_t 322, 330
- implementation pragma 255
- implicit conversions 167
- implicit declaration 174
- include preprocessor directive 228
- inclusive OR operator (bitwise) | 158
- incomplete class declarations 287, 292, 346
- increment operator ++ 142, 321
- indentation of code 220
- indirect base class 345, 356
- indirection operator \* 145
- info pragma 255
- inheritance
  - design using 345
  - graph 345, 357
  - in object-oriented programming 43
  - multiple 344, 356
  - overview 343
  - single 343
- initial expression 127
- initialization
  - by constructor
    - base classes 338
    - explicit 336
    - members 338
  - by copying 341
  - member lists 325
  - members 292
  - static data members 303
- initializers
  - array 102
  - character 86
  - constructors 336
  - description 127
  - floating 87
  - integer 90
  - structure 108
- inline
  - functions
    - constructors 325
    - description 195
    - specifiers 129
  - keyword 195
  - member functions
    - description 294
    - template 378
  - pragma 255
- inline pragma 255
- inlining functions 255
- input
  - operator 48
  - record 272
- input operator 48
- input/output overview 47
- insertion operator 48

- instantiation
  - member function templates 377
  - template classes 369, 380
  - template functions 376
- int constant 60
- int type specifier 90
- integer
  - ANSI/ISO conformance 413
  - conversions 168
  - data types 89
  - floating-point constant 62
  - integer constants 60
- integral
  - operands 135
  - promotions 167, 313
- internal linkage 38
- isolated\_call pragma 257

## K

- keywords
  - \_\_cdecl 123
  - description 57
  - export2 125

## L

- label statement 197
- langlvl pragma 259
- left-shift operator << 154
- less than operator < 155
- less than or equal to operator <= 155
- library functions 416
- limits
  - floating-point 413
  - integer 413
- line continuation
  - escape sequence, as an 67
  - preprocessor directives, in 220
  - string constants, in 65
- LINE macro 230
- line preprocessor directive 241
- linkage of identifiers 37
- linkage pragma for interlanguage calls 260
- linkage specifications 50
- linking to non-C++ programs 50
- literal 65
- local
  - classes 288
  - scope 35, 46
  - type names 289
  - variables 35, 70
- LOCALE macro 233
- localization 420
- logical AND operator && 158
- logical negation operator ! 144
- logical OR operator || 159
- long double type specifier 87
- long long
  - conversions 168
- long long type specifier 89
- long name support 59
- long type specifier 89
- LONGNAME compiler option 59
- LONGNAME macro 234

- longname pragma 261
- loop statements
  - do 203
  - for 206
  - while 216
- lvalue 136

## M

- macro
  - definition 221, 222
  - invocation 222
- main function 35, 184
- main function, parameters 35
- map pragma 262
- margins pragma 264
- matching arguments
  - description 312
  - exception handling 387
  - template functions 374, 376
- member functions
  - constant 293
  - constructors 325
  - definition 293
  - description 293
  - destructors 325
  - inline 294
  - local classes 289
  - overloading operators 316
  - special 294, 325
  - static 303
  - templates 377
  - this pointer 298, 362
  - volatile 293
- member lists 283, 291
- member of a structure 108
- members
  - access
    - default 305
    - inherited 349
    - public, private, and protected 305
  - arrays 292
  - class member access operators 141
  - class type 292
  - data 292
  - declaration 291
  - declarator 291
  - inherited 346
  - initialization 336, 338
  - initializer list 325
  - of classes 291
  - overloading class access operators 320
  - pointers to 160, 297
  - protected 350
  - scope 295
  - static 288, 300
  - virtual functions 294
- memberwise assignment 319
- memory
  - data mapping 129
  - management 419
- minus, unary operator 143
- modifying access 351
- modulo operator % 153
- multibyte characters
  - ANSI/ISO conformance 412

- multibyte characters (*continued*)
  - overview 412
- multiple
  - access 357
  - inheritance 344, 356
- multiplicative operators
  - division / 153
  - multiplication \* 152
  - remainder % 153
- MVS (Multiple Virtual System)
  - variable names 58
- MVS macro 234
- MVS variable names 58

## N

- name spaces 39
- names
  - class 283, 286
  - dominant 348
  - hidden 137, 285, 286
  - local type 289
  - types 394
- naming
  - classes 39
  - external names 58
  - long names 59
- negation operators
  - bitwise ~ 144
  - logical ! 144
- nested classes
  - access declarations 352
  - friend scope 308
  - scope 287
- nested template arguments 369
- nested try blocks 388
- nested visibility 35
- nesting level limits 417
- new-line character
  - escape sequence \n 66, 67
  - white space, as 220
- new operator
  - description 147
  - free store 330
  - overloading 322
- noinline pragma 255, 265
- NOLONGNAME compiler option 59
- nolongname pragma 261
- nomargins pragma 264
- nosequence pragma 272
- not equal to operator != 156
- null character \0 65
- null pointer 168
- NULL pointer 96
- null statement 210
- number sign (#)
  - preprocessor directive character 220
  - preprocessor operator 225

## O

- object-like macro 221
- objects
  - base class 357
  - class
    - copying 340

- objects (*continued*)
  - class (*continued*)
    - declarations 284
    - initialization 336
  - construction order 327
    - of class objects 327
    - of derived classes 339
  - constructors 326
  - conversion to 334
  - data abstraction 42
  - description 72
  - destruction order 329
  - destructors 326
  - explicitly constructing 328
  - temporary 193, 326, 330, 333
- octal constant 62
- octal numbers as escape sequences 67
- one's complement operator ~ 144
- operator 225
- operator delete() function 330
- operator new() function 330
- operators 225
  - .\* (pointer-to-member) 160
  - :: (scope resolution) 137
  - . (dot) 141
  - >\* (pointer-to-member) 160
  - > (arrow) 141
- additive
  - addition operator + 153
  - subtraction - 154
- assignment
  - compound 164
  - copying classes 341
  - default 340
  - description 162
  - overloading 319
  - simple = 162
- associativity 133
- binary 152, 318
- bitwise AND & 157
- bitwise exclusive OR ^ 157
- bitwise inclusive OR | 158
- bitwise shift
  - left-shift << 154
  - right-shift >> 154
- comma 165
- conditional ? : 160
- delete
  - description 151
  - free store 330
  - overloading 322
- digitof 147
- equality
  - equal to == 156
  - not equal to != 156
- equality operators 156
- logical AND && 158
- logical OR || 159
- multiplicative
  - division / 153
  - multiplication \* 152
  - remainder % 153
- new
  - description 147
  - free store 330
  - overloading 322



- operators 225 (*continued*)
    - overloading
      - arrow 320
      - binary 318
      - description 315
      - dot 320
      - general rules 316
      - increment 321
      - restrictions 317
      - subscripting 320
      - unary 317
    - pointer to member 297
    - pointer-to-member 160
    - precedence and associativity 133
    - precisionof 147
    - preprocessor
      - # 225
      - ## 226
    - primary 136
      - array subscripting [ ] 140
    - relational
      - greater than > 155
      - greater than or equal to >= 155
      - less than < 155
      - less than or equal to <= 155
    - relational operators 155
    - scope resolution 347, 354, 358, 361
    - unary 142
      - address operator & 144
      - bitwise negation operator ~ 144
      - decrement operator -- 143
      - increment operator (++) 142
      - indirection operator \* 145
      - logical negation operator ! 144
      - overloading 317
      - sizeof operator 146
      - unary minus operator (-) 143
      - unary plus operator (+) 143
  - optimization
    - inlining 255
  - options
    - compiler
      - overriding defaults 266
      - specifying 266
    - pragma 266
    - run-time 271
  - OR operator (logical) || 159
  - order
    - of catching exceptions 388
    - template declaration 370, 376
  - OS linkage 261
  - output
    - operator 48
  - output operator 48
  - overloading
    - functions
      - access declarations 353
      - argument matching 312
      - arguments 313
      - declaration matching 312
      - restrictions 312
    - operators
      - argument matching 312
      - assignment 319
      - class member access 320
      - decrement 321
      - delete 322, 331, 332
  - overloading (*continued*)
    - operators (*continued*)
      - description 315
      - function call 319
      - general rules 316
      - increment 321
      - member functions 316
      - new 322, 331
      - operands 316
      - restrictions 317
      - subscript 320
    - resolution for template functions 374
    - special operators 319
  - overriding virtual functions 360, 362
- ## P
- pack pragma 267
  - packed
    - assignments and comparisons 163
    - structures 112, 186
    - unions 115, 186
  - Packed qualifier 122
  - page pragma 270
  - pagesize pragma 270
  - parameter declaration list 181
  - parameter passing 185
  - parameters, to main function 35
  - parenthesis
    - for calling functions 139
    - for grouping expressions 137
  - pass by reference 188
  - passing a value 187
  - passing an address 187
  - phases of preprocessing 220
  - PL/I linkage 261
  - placement syntax 149, 331
  - plus, unary operator 143
  - pointer to member
    - conversions 169
    - declarations 297
  - pointer-to-member
    - operators 160
  - pointers
    - arrays 414
    - conversions 168
    - description 94
    - this 298
    - to functions 193
    - to members 160, 297
  - polymorphism 346, 348
    - in object-oriented programming 44
  - portability issues 411
  - POSIX 409
  - pound sign (#)
    - preprocessor directive character 220
    - preprocessor operator 225
  - pragma definitions
    - define 251
    - implementation 255
    - info 255
    - noinline 265
    - priority 270
  - pragma directives
    - csect 250
    - exporting functions and variables 253
  - pragma directives (*continued*)
    - linkage 260
    - runopts 271
    - variable 275
    - wsizof 275
  - pragmas
    - chars 247
    - checkout 248
    - comment 248
    - convlit 249
    - csect 250
    - define 251
    - disjoint 251
    - environment 252
    - filetag 253
    - hdrstop 254
    - implementation 255
    - info 255
    - inline 255
    - IPA considerations 247
    - isolated\_call 257
    - langlvl 259
    - longname 261
    - map 262
    - margins 264
    - noinline 255, 265
    - nolongname 261
    - nomargins 264
    - nosequence 272
    - options 266
    - pack 267
    - page 270
    - pagesize 270
    - priority 270
    - runopts 271
    - sequence 272
    - skip 273
    - strings 273
    - subtitle 274
    - target 274
    - title 275
    - variable 275
    - wsizof 275
  - precedence of operators 133
  - precisionof operator 147
  - predefined macros
    - 370 236
    - ANSI 231
    - BFP 231
    - CODESET 232
    - COMMONC 232
    - COMPAT 232
    - COMPILER\_VER 232
    - DATE 230
    - DLL 232
    - EXTENDED 231
    - FILE 230
    - FILETAG 232
    - FUNCTION 232
    - HHW370 233
    - HOSMVS 233
    - IBMC 233
    - IBMCP 233
    - LINE 230
    - LOCALE 233
    - LONGNAME 234
    - MVS 234

- predefined macros (*continued*)
  - SAA 231
  - SAAL2 231
  - SOM\_ENABLED 234
  - STDC 230
  - STRING\_CODE\_SET 234
  - TARGET\_LIB 235
  - TEMPINC 234
  - THW370 235
  - TIME 231
  - TIMESTAMP 235
  - TOSMVS 236
- preprocessing, phases of 220
- preprocessing directives
  - ANSI/ISO conformance 415
- preprocessor directive character 220
- preprocessor directives
  - ## operator 226
  - # operator 225
  - conditional compilation 237
  - define 221
  - else 240
  - endif 240
  - error 228
  - format of 220
  - include 228
  - line control 241
  - pragma 243
  - undef 225
- preprocessor operator
  - # 225
  - ## 226
- primary expression 136
- priority pragma 270
- private keyword 305, 350
- program, running 35
- program entry point 184
- program linkage 37
- promotions (integral) 167, 313
- protected keyword 305, 339
- protected member access 350
- prototype 178
- public derivation 350
- public keyword 305, 339, 350
- pure specifier 292, 294, 329, 361, 363

## Q

- qualified
  - class name 137
  - type name 288
- qualifiers
  - \_Packed 122
  - const 120
  - volatile 120

## R

- record
  - margins 264
  - sequence numbers 272
- recursive
  - function calls 139
- reentrancy
  - variables 275
- reentrant variables 275

- reference scope 353
- references
  - conversions 169
  - description 129
  - initialization 130
  - pass by reference 188
  - return types 193
  - temporary objects 333
- register
  - storage class specifier 81
- register storage class specifier 81
- registers
  - ANSI/ISO conformance 414
- relational operators
  - greater than > 155
  - greater than or equal to >= 155
  - less than < 155
  - less than or equal to <= 155
- remainder operator % 153
- restoring access 351
- resumption model 384
- rethrowing exceptions 389
- return statement 192, 211
- return types
  - description 192
  - references 193
- return values 176, 192
- right-shift operator >> 154
- running, starting point 35
- runopts pragma 271
- runtime
  - options 271

## S

- SAA macro 231
- SAAL2 macro 231
- scalar operands 135
- scope
  - block 35
  - call 353
  - class names 286
  - description 35, 46
  - file 36
  - friend 308
  - function 36
  - local classes 288
  - member 295
  - nested classes 287
  - reference 353
- scope resolution operator
  - ambiguous base classes 358
  - class member access 354
  - description 137
  - inheritance 347
  - virtual functions 361
- sequence pragma 272
- set\_new\_handler() library function 150
- set\_terminate() library function 395
- set\_unexpected() library function 395
- shift operators (<< and >>) 154
- shift states 412
- short type specifier 89
- signal
  - function 417
  - handler 383
- signal handler 383

- signed char type specifier 86
- signed int 89
- signed long 89
- signed long long 89
- simple assignment operator = 162
- simple I/O 47
- single inheritance 343
- sizeof operator 146
- skip pragma 273
- SOM\_ENABLEDmacro 234
- source
  - files 33
  - program 32
  - margins 264
  - variable names 58
- space character 220
- special functions
  - member functions 325
  - used in exception handling 395
- special member functions 294
- specifications
  - exception 393
  - linkage 50
- specifiers
  - access 305, 348, 350
  - base 348
  - class 282
  - declaration 291
  - inline 129, 195
  - pure 292, 294
  - virtual 129
- splice preprocessor directive ## 226
- standard conversions 167
- statements
  - ANSI/ISO conformance 415
  - block 198
  - break 200
  - continue 202
  - do 203
  - expression 205
  - for 206
  - goto 208
  - if 209
  - labels 197
  - null 210
  - overview 197
  - resolving ambiguities 205
  - return 192, 211
  - switch 213
  - while 216
- static
  - binding 44, 346
  - data members 302
  - initialization of data members 303
  - member functions 303
  - members 288, 300
  - storage class specifier 82
- STDC macro 230
- storage class specifiers
  - auto 73
  - extern 75
  - register 81
  - static 82
- storage duration 39
- storage of variables 129
- streams 418

- string
  - constants 65
  - literals 65
- STRING\_CODE\_SET macro 234
- strings
  - conversion 413
- strings pragma 273
- struct type specifier 107
- structures
  - ANSI/ISO conformance 414
  - packing
    - using `_Packed` qualifier 112
    - using `#pragma pack` 267
- subdeclarator 120
- subscript declarator
  - description 120
  - in arrays 101
- subscript operator
  - overloading 320
- subscripts 140
- subtitle pragma 274
- subtraction operator – 154
- switch statement 212
- syntax diagrams, how to read 10

## T

- tab character
  - horizontal escape sequence `\t` 67
  - vertical escape sequence `\v` 67
  - white space, as 220
- TARGET\_LIB macro 235
- target pragma 274
- TEMPINC macro 234
- template classes
  - declaration 370
  - definition 370
  - description 369
  - explicit definition 373, 380
  - instantiation 380
- template functions
  - declarations 376
  - definitions 376
  - description 373
  - explicit definition 375
  - grouping definitions of 375
  - instantiation 376
  - overloading resolution 374
- templates
  - `#pragma define` 251
  - `#pragma implementation` 255
  - argument
    - list 366, 369
    - nested list 370
    - nontype 371
  - class templates 369
  - constructors 370, 378
  - declaration 366
  - default initializers 366
  - friends 379
  - function templates 373
  - identifier 366
  - member functions 377
  - `pragma define` 251
  - `pragma implementation` 255
  - static data members 380
  - syntax 365
- temporary objects 193, 326, 330, 333
- terminate function 395
- termination model 384
- this pointer 298, 314, 362
- throw
  - argument matching 387
  - expression 152, 382, 388
  - keyword 382
  - point 384
  - rethrowing exceptions 389
- THW370 macro 235
- time 420
- TIME macro 231
- TIMESTAMP macro 235
- title pragma 275
- TMP\_MAX macro 419
- `tmpnam()` library function 419
- tokens 51, 219
- TOSMVS macro 236
- translation
  - limits 417
- translation limits 417
- trigraphs 53
- trivial conversions 314
- try
  - blocks 382
  - keyword 382
  - nested blocks 388
- type
  - data mapping 129
- type checking 182
- type conversions 167
- type declarations
  - array 100
  - character 86
  - enumerations 90
  - fixed-point decimal 88
  - floating-point 87
  - functions 178
  - integer 89
  - pointer 94
  - scalar 85
  - structure 106
  - typedef 84
  - union 113
  - void 99
- type names
  - exception specification syntax 394
  - local 289
  - scope 46
- type qualifiers
  - `_Packed` 122
  - `const` 120
  - `volatile` 120
- type specifier
  - (long) double 87
  - char 86
  - enumeration 90
  - float 87
  - int 89, 90
  - long 89
  - long long 89
  - short 89
  - union 115
  - unsigned 89
- typedef 261

- typedef specifier
  - class declaration 289
  - description 84
  - local type names 289
  - pointers to members 297
  - qualified type name 288
  - restrictions on overloaded functions 312
- types
  - aggregate classes 282
  - conversions 145
  - data abstraction 42

## U

- unary expression 142
- unary minus operator – 143
- unary operators
  - address operator `&` 144
  - bitwise negation operator `~` 144
  - decrement operator `--` 143
  - increment operator `++` 142
  - indirection operator `*` 145
  - logical negation operator `!` 144
  - minus 143
  - overloading 317
  - plus 143
  - `sizeof` operator 146
- unary plus operator `(+)` 143
- `undef` preprocessor directive 225
- underscores in identifiers 58
- `unexpected()` library function 393
- `unexpected` function 395
- union specifier 114
- unions
  - anonymous in C 116
  - anonymous in C++ 117
  - ANSI/ISO conformance 414
  - constructors and destructors 325
  - member destructors 329
  - packing
    - using `_Packed` qualifier 115
    - using `#pragma pack` 267
- unsigned char type specifier 86
- unsigned int type specifier 89
- unsigned long long type specifier 89
- unsigned long type specifier 89
- unsigned short type specifier 89
- unsigned type specifier 89
- user-defined
  - conversions 313, 334
  - types 281

## V

- variable arguments 182
- variable pragma 275
- variables
  - array 100
  - block scope data declarations 70
  - character 86
  - enumeration 90
  - file scope data declarations 71
  - floating-point 87
  - integer 89
  - local 35

- variables (*continued*)
  - names 58
  - pointer 94
  - storage of 129
  - structure 106
  - union 113
- virtual
  - base classes 357, 359
  - destructors 325
  - function specifier 129
  - functions
    - access 362
    - ambiguous calls to 361
    - description 359
    - dynamic binding 346
    - hidden 360
    - overriding 360, 362
    - pure 363
  - keyword 348
  - member functions 294
- visibility
  - block 35, 47
  - class members 304
  - description 35
  - nested 35
- void 99
- void function 176
- volatile
  - keyword 314
  - member functions 293
  - qualifier 120

## W

- wchar\_t 64
- while statement 216
- white space 54, 219, 220, 225
- wide character constant 64
- wide characters
  - ANSI/ISO conformance 412
- wsizeof pragma 275

---

## Readers' Comments — We'd Like to Hear from You

OS/390  
C/C++  
Language Reference

Publication No. SC09-2360-03

Overall, how satisfied are you with the information in this book?

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Overall satisfaction	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

How satisfied are you that the information in this book is:

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Accurate	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Complete	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Easy to find	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Easy to understand	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Well organized	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Applicable to your tasks	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

Please tell us how we can improve this book:

Thank you for your responses. May we contact you? ☐ Yes ☐ No

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

---

Name

---

Address

---

Company or Organization

---

Phone No.



Cut or Fold  
Along Line

Fold and Tape

Please do not staple

Fold and Tape

PLACE  
POSTAGE  
STAMP  
HERE

IBM Canada Ltd. Laboratory  
Information Development  
2G/345/1150/TOR  
1150 EGLINTON AVENUE EAST  
NORTH YORK ONTARIO CANADA  
M3C 1H7

Fold and Tape

Please do not staple

Fold and Tape

Cut or Fold  
Along Line





Printed in the United States of America

SC09-2360-03

