

z/OS



XL C/C++ Language Reference

z/OS



XL C/C++ Language Reference

Note!

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

Seventh Edition (September, 2007)

This edition applies to XL C/C++ in Version 1, Release 9 of z/OS (5694-A01), and to all subsequent releases and modifications until otherwise indicated in new editions. This edition replaces the z/OS V1R8 XL C/C++ Language Reference, SC09-4815-06. 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.

This book contains terminology, maintenance, and editorial changes. Technical changes or additions to the text and illustrations 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. You can also browse the books on the World Wide Web by clicking on "The Library" link on the z/OS home page. The web address for this page is <http://www.ibm.com/servers/eserver/zseries/zos/bkserv>

IBM welcomes your comments. You can send them by the Internet to the following address:

compinfo@ca.ibm.com

Include the title and order number of this book, and the page number or topic related to your comment. Be sure to include your e-mail address if you want a reply.

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 1998, 2007. All rights reserved.

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

Contents

About this document	xiii
Who should read this document	xiii
How to use this document.	xiii
How this document is organized	xiii
Conventions used in this document	xiii
z/OS XL C/C++ and related documents	xvii
Softcopy documents.	xxii
Softcopy examples	xxii
z/OS XL C/C++ on the World Wide Web	xxii
Where to find more information	xxii
Technical support	xxiv
How to send your comments	xxiv
About IBM z/OS XL C/C++	xxv
Changes for z/OS V1R9	xxv
The XL C/C++ compilers	xxvii
The C language	xxvii
The C++ language	xxvii
Common features of the z/OS XL C and XL C++ compilers	xxvii
z/OS XL C compiler-specific features	xxviii
z/OS XL C++ compiler-specific features	xxix
Class libraries.	xxix
Utilities	xxix
dbx	xxx
Language Environment element	xxx
Language Environment downward compatibility	xxxi
About prelinking, linking, and binding	xxxii
Notes on the prelinking process	xxxii
File format considerations	xxxiii
The program management binder	xxxiii
z/OS UNIX System Services	xxxiv
z/OS XL C/C++ applications with z/OS UNIX System Services C functions	xxxvi
Input and output	xxxvi
I/O interfaces	xxxvi
File types	xxxvii
Additional I/O features	xxxviii
The System Programming C facility	xxxviii
Interaction with other IBM products	xxxviii
Additional features of z/OS XL C/C++.	xli
Chapter 1. Scope and linkage	1
Scope	1
Block/local scope	2
Function scope	3
Function prototype scope	3
File/global scope.	3
Examples of scope in C	4
Class scope (C++ only)	4
Namespaces of identifiers	5
Name hiding (C++ only)	6
Program linkage	7
Internal linkage	7

External linkage	8
No linkage	9
Language linkage (C++ only)	9

Chapter 2. Lexical Elements	13
Tokens	13
Keywords	13
Identifiers	15
Literals	19
Punctuators and operators	29
Source program character set	31
Multibyte characters	32
Escape sequences	33
The Unicode standard	34
Digraph characters	35
Trigraph sequences	35
Comments	36

Chapter 3. Data objects and declarations	39
Overview of data objects and declarations	39
Overview of data objects	39
Overview of data declarations and definitions	41
Storage class specifiers	43
The auto storage class specifier	44
The static storage class specifier	44
The extern storage class specifier	46
The mutable storage class specifier (C++ only)	47
The register storage class specifier	47
Type specifiers	49
Integral types	49
Boolean types	50
Floating-point types	51
Fixed-point decimal types (C only)	53
Character types	54
The void type	54
Compatibility of arithmetic types (C only)	55
User-defined types	55
Structures and unions	55
Enumerations	61
Compatibility of structures, unions, and enumerations (C only)	64
typedef definitions	65
Type qualifiers	67
The __callback type qualifier	69
The const type qualifier	69
The __far type qualifier (C only)	70
The __ptr32 type qualifier	72
The restrict type qualifier	73
The volatile type qualifier	74
Type attributes	74
The armode noarmode type attribute (C only)	75

Chapter 4. Declarators	77
Overview of declarators	77
Examples of declarators	78
Type names	79
Pointers	80

Pointer arithmetic	81
Type-based aliasing	82
Compatibility of pointers (C only)	83
Arrays	84
Variable length arrays (C only)	86
Compatibility of arrays (C only)	87
References (C++ only)	87
Initializers	88
Initialization and storage classes	89
Designated initializers for aggregate types (C only)	90
Initialization of structures and unions	92
Initialization of enumerations	94
Initialization of pointers	94
Initialization of arrays	95
Initialization of references (C++ only)	98
Declarator qualifiers	99
The <code>_Packed</code> qualifier (C only)	99
The <code>_Export</code> qualifier (C++ only)	100
Variable attributes	100
The aligned variable attribute	101
Chapter 5. Type conversions	103
Arithmetic conversions and promotions	103
Integral conversions	104
Boolean conversions	104
Floating-point conversions	104
Packed decimal conversions (C only)	105
Integral and floating-point promotions	106
Lvalue-to-rvalue conversions	107
Pointer conversions	107
Conversion to <code>void*</code>	108
Reference conversions (C++ only)	109
Qualification conversions (C++ only)	109
Function argument conversions	110
Chapter 6. Expressions and operators	111
Lvalues and rvalues	111
Primary expressions	112
Names	113
Literals	113
Integer constant expressions	113
Identifier expressions (C++ only)	114
Parenthesized expressions ()	115
Scope resolution operator <code>::</code> (C++ only)	116
Function call expressions	116
Member expressions	117
Dot operator <code>.</code>	117
Arrow operator <code>-></code>	117
Unary expressions	118
Increment operator <code>++</code>	118
Decrement operator <code>--</code>	119
Unary plus operator <code>+</code>	120
Unary minus operator <code>-</code>	120
Logical negation operator <code>!</code>	120
Bitwise negation operator <code>~</code>	120
Address operator <code>&</code>	121

Indirection operator *	122
The typeid operator (C++ only)	122
The sizeof operator.	123
The typeof operator.	125
The digitsof and precisionof operators (C only).	126
The __real__ and __imag__ operators (C only)	126
Binary expressions	127
Assignment operators	128
Multiplication operator *	130
Division operator /	130
Remainder operator %	131
Addition operator +	131
Subtraction operator -	131
Bitwise left and right shift operators << >>	132
Relational operators < > <= >=	132
Equality and inequality operators == !=	133
Bitwise AND operator &	135
Bitwise exclusive OR operator ^	135
Bitwise inclusive OR operator	136
Logical AND operator &&.	136
Logical OR operator .	137
Array subscripting operator []	138
Comma operator ,	139
Pointer to member operators .* ->* (C++ only)	141
Conditional expressions	141
Types in conditional C expressions	142
Types in conditional C++ expressions	142
Examples of conditional expressions	143
Cast expressions	143
Cast operator ()	144
The static_cast operator (C++ only)	145
The reinterpret_cast operator (C++ only)	146
The const_cast operator (C++ only)	148
The dynamic_cast operator (C++ only).	149
Compound literal expressions (C only).	151
new expressions (C++ only)	151
Placement syntax	153
Initialization of objects created with the new operator	154
Handling new allocation failure	154
delete expressions (C++ only)	155
throw expressions (C++ only)	156
Operator precedence and associativity.	156
Chapter 7. Statements	161
Labeled statements.	161
Expression statements	162
Resolution of ambiguous statements	162
Block statements.	163
Example of blocks	163
Selection statements	164
The if statement	164
The switch statement	166
Iteration statements.	170
The while statement	170
The do statement	171
The for statement	171

	Jump statements	173
	The break statement	173
	The continue statement	174
	The return statement	175
	The goto statement	177
	Null statement	178
	Inline assembly statements (C only)	178
	Examples of inline assembly statements	181
	Restrictions on inline assembly statements	181
	Chapter 8. Functions	183
	Function declarations and definitions	183
	Function declarations	183
	Function definitions	184
	Examples of function declarations	185
	Examples of function definitions	186
	Compatible functions	186
	Multiple function declarations	187
	Function storage class specifiers	188
	The static storage class specifier	188
	The extern storage class specifier	188
	Function specifiers	190
	The inline function specifier	190
	The <code>__cdecl</code> function specifier (C++ only)	194
	The <code>_Export</code> function specifier (C++ only)	197
	Function return type specifiers	198
	Function return values	199
	Function declarators	199
	Parameter declarations	200
	Function attributes	203
	The <code>armode noarmode</code> function attribute (C only)	204
	The <code>main()</code> function	205
	Command-line arguments	206
	Function calls	207
	Pass by value	208
	Pass by reference	209
	Allocation and deallocation functions (C++ only)	210
	Default arguments in C++ functions	211
	Restrictions on default arguments	212
	Evaluation of default arguments	213
	Pointers to functions	214
	Chapter 9. Namespaces (C++ only)	217
	Defining namespaces (C++ only)	217
	Declaring namespaces (C++ only)	217
	Creating a namespace alias (C++ only)	217
	Creating an alias for a nested namespace (C++ only)	218
	Extending namespaces (C++ only)	218
	Namespaces and overloading (C++ only)	219
	Unnamed namespaces (C++ only)	219
	Namespace member definitions (C++ only)	221
	Namespaces and friends (C++ only)	221
	The <code>using</code> directive (C++ only)	222
	The <code>using</code> declaration and namespaces (C++ only)	222
	Explicit access (C++ only)	223

Chapter 10. Overloading (C++ only)	225
Overloading functions (C++ only)	225
Restrictions on overloaded functions (C++ only)	226
Overloading operators (C++ only)	227
Overloading unary operators (C++ only)	229
Overloading increment and decrement operators (C++ only)	230
Overloading binary operators (C++ only)	231
Overloading assignments (C++ only)	232
Overloading function calls (C++ only)	233
Overloading subscripting (C++ only)	234
Overloading class member access (C++ only)	235
Overload resolution (C++ only)	236
Implicit conversion sequences (C++ only)	237
Resolving addresses of overloaded functions (C++ only)	238
Chapter 11. Classes (C++ only)	241
Declaring class types (C++ only)	242
Using class objects (C++ only)	242
Classes and structures (C++ only)	244
Scope of class names (C++ only)	245
Incomplete class declarations (C++ only)	246
Nested classes (C++ only)	246
Local classes (C++ only)	248
Local type names (C++ only)	249
Chapter 12. Class members and friends (C++ only)	251
Class member lists (C++ only)	251
Data members (C++ only)	252
Member functions (C++ only)	253
Inline member functions (C++ only)	253
Constant and volatile member functions (C++ only)	254
Virtual member functions (C++ only)	254
Special member functions (C++ only)	254
Member scope (C++ only)	255
Pointers to members (C++ only)	256
The this pointer (C++ only)	257
Static members (C++ only)	260
Using the class access operators with static members (C++ only)	260
Static data members (C++ only)	261
Static member functions (C++ only)	263
Member access (C++ only)	265
Friends (C++ only)	267
Friend scope (C++ only)	269
Friend access (C++ only)	271
Chapter 13. Inheritance (C++ only)	273
Derivation (C++ only)	275
Inherited member access (C++ only)	278
Protected members (C++ only)	278
Access control of base class members (C++ only)	279
The using declaration and class members (C++ only)	280
Overloading member functions from base and derived classes (C++ only)	281
Changing the access of a class member (C++ only)	283
Multiple inheritance (C++ only)	284
Virtual base classes (C++ only)	285
Multiple access (C++ only)	286

Ambiguous base classes (C++ only)	287
Virtual functions (C++ only)	291
Ambiguous virtual function calls (C++ only)	294
Virtual function access (C++ only)	296
Abstract classes (C++ only)	296
Chapter 14. Special member functions (C++ only)	299
Overview of constructors and destructors (C++ only)	299
Constructors (C++ only)	301
Default constructors (C++ only)	301
Explicit initialization with constructors (C++ only)	302
Initialization of base classes and members (C++ only)	304
Construction order of derived class objects (C++ only)	307
Destructors (C++ only)	308
Pseudo-destructors (C++ only)	310
User-defined conversions (C++ only)	311
Conversion constructors (C++ only)	312
The explicit specifier (C++ only)	314
Conversion functions (C++ only)	314
Copy constructors (C++ only)	315
Copy assignment operators (C++ only)	317
Chapter 15. Templates (C++ only)	319
Template parameters (C++ only)	320
Type template parameters (C++ only)	320
Non-type template parameters (C++ only)	320
Template template parameters (C++ only)	321
Default arguments for template parameters (C++ only)	321
Template arguments (C++ only)	322
Template type arguments (C++ only)	322
Template non-type arguments (C++ only)	323
Template template arguments (C++ only)	325
Class templates (C++ only)	326
Class template declarations and definitions (C++ only)	327
Static data members and templates (C++ only)	328
Member functions of class templates (C++ only)	329
Friends and templates (C++ only)	329
Function templates (C++ only)	330
Template argument deduction (C++ only)	331
Overloading function templates (C++ only)	337
Partial ordering of function templates (C++ only)	337
Template instantiation (C++ only)	338
Implicit instantiation (C++ only)	339
Explicit instantiation (C++ only)	340
Template specialization (C++ only)	341
Explicit specialization (C++ only)	341
Partial specialization (C++ only)	346
Name binding and dependent names (C++ only)	348
The typename keyword (C++ only)	349
The template keyword as qualifier (C++ only)	350
Chapter 16. Exception handling (C++ only)	353
try blocks (C++ only)	353
Nested try blocks (C++ only)	354
catch blocks (C++ only)	355
Function try block handlers (C++ only)	356

Arguments of catch blocks (C++ only)	359
Matching between exceptions thrown and caught (C++ only)	359
Order of catching (C++ only)	359
throw expressions (C++ only)	361
Rethrowing an exception (C++ only)	361
Stack unwinding (C++ only)	362
Exception specifications (C++ only)	364
Special exception handling functions (C++ only)	366
The unexpected() function (C++ only)	367
The terminate() function (C++ only)	368
The set_unexpected() and set_terminate() functions (C++ only)	369
Example using the exception handling functions (C++ only)	369
Chapter 17. Preprocessor directives	373
Macro definition directives	373
The #define directive	373
The #undef directive	378
The # operator	378
The ## operator	379
Standard predefined macro names	380
File inclusion directives	381
The #include directive	382
The #include_next directive	383
Conditional compilation directives	384
The #if and #elif directives	385
The #ifdef directive	386
The #ifndef directive	386
The #else directive	387
The #endif directive	387
Message generation directives	388
The #error directive	388
The #line directive	389
The null directive (#)	390
Pragma directives	390
The _Pragma preprocessing operator (C only)	391
Standard pragmas (C only)	391
Chapter 18. z/OS XL C/C++ pragmas	393
Pragma directive syntax	393
Scope of pragma directives	393
IPA considerations	394
Summary of compiler pragmas by functional category	394
Language element control	394
C++ template pragmas	395
Floating-point and integer control	395
Error checking and debugging	395
Listings, messages and compiler information	396
Optimization and tuning	396
Object code control	396
Portability and migration	397
Individual pragma descriptions	398
#pragma chars	398
#pragma checkout	399
#pragma comment	400
#pragma convert	402
#pragma convlit	403

|
|

#pragma csect	404
#pragma define (C++ only)	405
#pragma disjoint	406
#pragma enum	407
#pragma environment (C only).	409
#pragma export	409
#pragma extension	410
#pragma filetag	411
#pragma implementation (C++ only)	412
#pragma info (C++ only)	413
#pragma inline (C only) / noinline.	414
#pragma isolated_call	415
#pragma langlvl directive (C only)	417
#pragma leaves	418
#pragma linkage (C only)	419
#pragma longname/nolongname	422
The #pragma map directive	423
#pragma margins	425
#pragma namemangling (C++ only)	426
#pragma namemanglingrule (C++ only)	427
#pragma object_model (C++ only)	429
#pragma operator_new (C++ only)	430
#pragma option_override	431
#pragma options (C only)	433
#pragma pack	435
#pragma page (C only)	438
#pragma pagesize (C only)	439
#pragma priority (C++ only)	439
#pragma prolog (C only), #pragma epilog (C only)	440
#pragma reachable	441
#pragma report (C++ only)	442
#pragma runopts	443
#pragma sequence	445
#pragma skip (C only)	446
#pragma strings	446
#pragma subtitle (C only)	447
#pragma target (C only)	447
#pragma title (C only)	449
#pragma unroll	449
#pragma variable	451
#pragma wsizeof	452
#pragma XOPTS.	454

Chapter 19. Compiler predefined macros	455
General macros	455
Macros indicating the z/OS XL C/C++ compiler product	457
Macros related to the platform	458
Macros related to compiler features	459
Macros related to compiler option settings	459
Macros related to language levels	464

Appendix A. C and C++ compatibility on the z/OS platform	469
String initialization	469
Class/structure and typedef names	469
Class/structure and scope declarations	469
const object initialization	470

Definitions	470
Definitions within return or argument types	470
Enumerator type	470
Enumeration type	470
Function declarations	470
Functions with an empty argument list	470
Global constant linkage	471
Jump statements.	471
Keywords	471
main() recursion	471
Names of nested classes/structures.	471
Pointers to void	471
Prototype declarations.	471
Return without declared value	471
__STDC__ macro	472
Appendix B. Common Usage C language level for the z/OS Platform	473
Appendix C. Conforming to POSIX 1003.1	475
Appendix D. Implementation-defined behavior.	477
Identifiers	477
Characters	477
String conversion	478
Integers	478
Floating-point numbers	479
C/C++ data mapping	480
Arrays and pointers.	480
Registers	480
Structures, unions, enumerations, bit fields	481
Declarators	481
Statements	481
Preprocessing directives	481
Translation limits.	482
Appendix E. Accessibility	485
Using assistive technologies	485
Keyboard navigation of the user interface.	485
z/OS information	485
Notices	487
Programming interface information	488
Trademarks.	488
Standards	489
Index	491

About this document

This document describes the syntax, semantics, and IBM® z/OS® XL C/C++ implementation of the C and C++ programming languages. Although the XL C and XL C++ compilers conform to the specifications maintained by the ISO standards for the C and C++ programming languages, the compilers also incorporate many extensions to the core languages. These extensions have been implemented with the aims of enhancing usability in specific operating environments, supporting compatibility with other compilers, and supporting new hardware capabilities. For example, on the z/OS platform, language constructs have been added to provide support for data types that are specific to the IBM System z™ environment.

Note: As of z/OS V1R7, the z/OS C/C++ compiler has been rebranded to z/OS XL C/C++.

Who should read this document

This document is a reference for users who already have experience programming applications in C or C++. Users new to C or C++ can still use this document to find information on the language and features unique to XL C/C++; however, this reference does not aim to teach programming concepts nor to promote specific programming practices.

How to use this document

Unless indicated otherwise, all of the text in this reference pertains to both C and C++ languages. Where there are differences between languages, these are indicated through qualifying text and other graphical elements (see below for the conventions used).

While this document covers both standard and implementation-specific features, it does not include the following topics:

- Standard C and C++ library functions and headers. For standard C/C++ library documentation, refer to the *Standard C++ Library Reference*.

How this document is organized

This document is organized to loosely follow the structure of the ISO standard language specifications and topics are grouped into similar headings.

Chapters 1 through 8 discuss language elements that are common to both C and C++, including lexical elements, data types, declarations, declarators, type conversions, expressions, operators, statements, and functions. Throughout these chapters, both standard features and extensions are discussed. Chapters 9 through 16 discuss standard C++ features exclusively, including classes, overloading, inheritance, templates, and exception handling. Chapters 17 through 19 discuss directives to the preprocessor and macros that are predefined by the compiler.

Conventions used in this document

Typographical conventions

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

Table 1. Typographical conventions

Typeface	Indicates	Example
bold	Lowercase commands, executable names, compiler options and directives.	The xlc utility provides two basic compiler invocation commands, xlc and xlc (xlc++) , along with several other compiler invocation commands to support various C/C++ language levels and compilation environments.
<i>italics</i>	Parameters or variables whose actual names or values are to be supplied by the user. Italics are also used to introduce new terms.	Make sure that you update the <i>size</i> parameter if you return more than the <i>size</i> requested.
monospace	Programming keywords and library functions, compiler built-in functions, examples of program code, command strings, or user-defined names.	If one or two cases of a <code>switch</code> statement are typically executed much more frequently than other cases, break out those cases by handling them separately before the <code>switch</code> statement.

Qualifying elements (icons and bracket separators)

This document uses marked bracket separators to delineate large blocks of text and icons to delineate small segments of text as follows:

Table 2. Qualifying elements

Qualifier/Icon	Meaning
C only 	The text describes a feature that is supported in the C language only; or describes behavior that is specific to the C language.
C++ only 	The text describes a feature that is supported in the C++ language only; or describes behavior that is specific to the C++ language.
IBM extension 	The text describes a feature that is an IBM z/OS XL C/C++ compiler extension to the standard language specifications.
z/OS only 	The text describes a feature that is supported only on the z/OS implementation of the XL C/C++ compilers.

Syntax diagrams

Throughout this document, diagrams illustrate z/OS XL C/C++ syntax. This section will help you to interpret and use those diagrams.

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

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

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

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

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

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

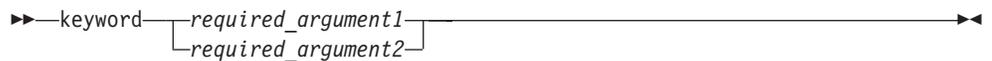
- Required items are shown on the horizontal line (the main path):



- Optional items are shown below the main path:



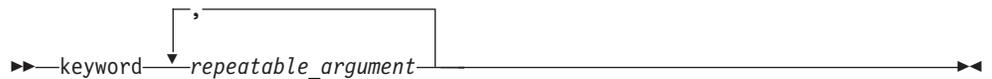
- If you can choose from two or more items, they are shown vertically, in a stack. If you *must* choose one of the items, one item of the stack is shown on the main path.



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



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



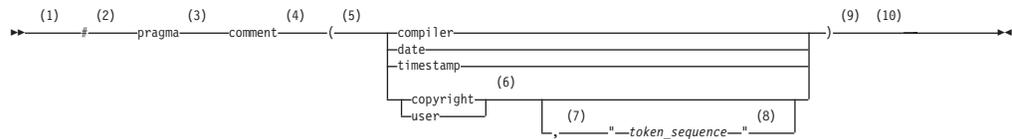
- The item that is the default is shown above the main path.



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

Sample syntax diagram

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 appear following the # symbol.
- 4 The name of the pragma comment must appear following the keyword pragma.
- 5 An opening parenthesis must be present.
- 6 The comment type must be entered only as one of the types indicated: compiler, date, timestamp, copyright, or user.
- 7 A comma must appear between the comment type copyright or user, and an optional character string.
- 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 shown above:

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

Examples

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

z/OS XL C/C++ and related documents

This topic summarizes the content of the z/OS XL C/C++ documents and shows where to find related information in other documents.

Table 3. z/OS XL C/C++ and related documents

Document Title and Number	Key Sections/Chapters in the Document
<i>z/OS XL C/C++ Programming Guide</i> , SC09-4765	Guidance information for: <ul style="list-style-type: none">• XL C/C++ input and output• Debugging z/OS XL C programs that use input/output• Using linkage specifications in C++• Combining C and assembler• Creating and using DLLs• Using threads in z/OS UNIX® System Services applications• Reentrancy• Handling exceptions, error conditions, and signals• Performance optimization• Network communications under z/OS UNIX System Services• Interprocess communications using z/OS UNIX System Services• Structuring a program that uses C++ templates• Using environment variables• Using System Programming C facilities• Library functions for the System Programming C facilities• Using run-time user exits• Using the z/OS XL C multitasking facility• Using other IBM products with z/OS XL C/C++ (CICS® Transaction Server for z/OS, CSP, DWS, DB2®, GDDM®, IMS™, ISPF, QMF™)• Internationalization: locales and character sets, code set conversion utilities, mapping variant characters• POSIX character set• Code point mappings• Locales supplied with z/OS XL C/C++• Charmap files supplied with z/OS XL C/C++• Examples of charmap and locale definition source files• Converting code from coded character set IBM-1047• Using built-in functions• Programming considerations for z/OS UNIX System Services C/C++
<i>z/OS XL C/C++ User's Guide</i> , SC09-4767	Guidance information for: <ul style="list-style-type: none">• z/OS XL C/C++ examples• Compiler options• Binder options and control statements• Specifying Language Environment® run-time options• Compiling, IPA Linking, binding, and running z/OS XL C/C++ programs• Utilities (Object Library, CXXFILT, DSECT Conversion, Code Set and Locale, ar and make, BPXBATCH, c89, xlc)• Diagnosing problems• Cataloged procedures and REXX™ EXECs supplied by IBM• Customizing default options for the z/OS XL C/C++ compiler

Table 3. z/OS XL C/C++ and related documents (continued)

Document Title and Number	Key Sections/Chapters in the Document
z/OS XL C/C++ Language Reference, SC09-4815	Reference information for: <ul style="list-style-type: none"> • The C and C++ languages • Lexical elements of z/OS XL C and C++ • Declarations, expressions, and operators • Implicit type conversions • Functions and statements • Preprocessor directives • C++ classes, class members, and friends • C++ overloading, special member functions, and inheritance • C++ templates and exception handling • z/OS XL C and C++ compatibility
z/OS XL C/C++ Messages, GC09-4819	Provides error messages and return codes for the compiler, and its related application interface libraries and utilities. For the XL C/C++ run-time library messages, refer to <i>z/OS Language Environment Run-Time Messages</i> , SA22-7566. For the c89 and xlc utility messages, refer to <i>z/OS UNIX System Services Messages and Codes</i> , SA22-7807.
z/OS XL C/C++ Run-Time Library Reference, SA22-7821	Reference information for: <ul style="list-style-type: none"> • header files • library functions
z/OS C Curses, SA22-7820	Reference information for: <ul style="list-style-type: none"> • Curses concepts • Key data types • General rules for characters, renditions, and window properties • General rules of operations and operating modes • Use of macros • Restrictions on block-mode terminals • Curses functional interface • Contents of headers • The terminfo database
z/OS XL C/C++ Compiler and Run-Time Migration Guide for the Application Programmer, GC09-4913	Guidance and reference information for: <ul style="list-style-type: none"> • Common migration questions • Application executable program compatibility • Source program compatibility • Input and output operations compatibility • Class library migration considerations • Changes between releases of z/OS • Pre-z/OS C and C++ compilers to current compiler migration • Other migration considerations
z/OS Metal C Programming Guide and Reference, SA23-2225	Guidance and reference information for: <ul style="list-style-type: none"> • Metal C run time • Metal C programming • AR mode
Standard C++ Library Reference, SC09-4949	The documentation describes how to use the following three main components of the Standard C++ Library to write portable C/C++ code that complies with the ISO standards: <ul style="list-style-type: none"> • ISO Standard C Library • ISO Standard C++ Library • Standard Template Library (C++) The ISO Standard C++ library consists of 51 required headers. These 51 C++ library headers (along with the additional 18 Standard C headers) constitute a hosted implementation of the C++ library. Of these 51 headers, 13 constitute the Standard Template Library, or STL.

Table 3. z/OS XL C/C++ and related documents (continued)

Document Title and Number	Key Sections/Chapters in the Document
<i>C/C++ Legacy Class Libraries Reference</i> , SC09-7652	Reference information for: <ul style="list-style-type: none"> • UNIX System Laboratories (USL) I/O Stream Library • USL Complex Mathematics Library This reference is part of the Run-Time Library Extensions documentation.
<i>IBM Open Class Library Transition Guide</i> , SC09-4948	The documentation explains the various options to application owners and users for migrating from the IBM Open Class [®] library to the Standard C++ Library.
<i>z/OS Common Debug Architecture User's Guide</i> , SC09-7653	This documentation is the user's guide for IBM's libddpi library. It includes: <ul style="list-style-type: none"> • Overview of the architecture • Information on the order and purpose of API calls for model user applications and for accessing DWARF information • Information on using the Common Debug Architecture with C/C++ source This user's guide is part of the Run-Time Library Extensions documentation.
<i>z/OS Common Debug Architecture Library Reference</i> , SC09-7654	This documentation is the reference for IBM's libddpi library. It includes: <ul style="list-style-type: none"> • General discussion of Common Debug Architecture • Description of APIs and data types related to stacks, processes, operating systems, machine state, storage, and formatting This reference is part of the Run-Time Library Extensions documentation.
<i>DWARF/ELF Extensions Library Reference</i> , SC09-7655	This documentation is the reference for IBM's extensions to the libdwarf and libelf libraries. It includes information on: <ul style="list-style-type: none"> • Consumer APIs • Producer APIs This reference is part of the Run-Time Library Extensions documentation.
Debug Tool documentation, available on the Debug Tool for z/OS library page on the World Wide Web	The documentation, which is available at www.ibm.com/software/awdtools/debugtool/library/ , provides guidance and reference information for debugging programs, using Debug Tool in different environments, and language-specific information.
APAR and README files (Shipped with Program materials)	Partitioned data set CBC.SCCNDOC on the product tape contains the members, APAR, and README, which provide additional information for using the z/OS XL C/C++ licensed program, including: <ul style="list-style-type: none"> • Isolating reportable problems • Keywords • Preparing an Authorized Program Analysis Report (APAR) • Problem identification worksheet • Maintenance on z/OS • Late changes to z/OS XL C/C++ publications
<p>Note: For complete and detailed information on linking and running with Language Environment services and using the Language Environment run-time options, refer to <i>z/OS Language Environment Programming Guide</i>, SA22-7561. For complete and detailed information on using interlanguage calls, refer to <i>z/OS Language Environment Writing Interlanguage Communication Applications</i>, SA22-7563.</p>	

The following table lists the z/OS XL C/C++ and related documents. The table groups the documents according to the tasks they describe.

Table 4. Documents by task

Tasks	Documents
Planning, preparing, and migrating to z/OS XL C/C++	<ul style="list-style-type: none"> • <i>z/OS XL C/C++ Compiler and Run-Time Migration Guide for the Application Programmer</i>, GC09-4913 • <i>z/OS Language Environment Customization</i>, SA22-7564 • <i>z/OS Language Environment Run-Time Application Migration Guide</i>, GA22-7565 • <i>z/OS UNIX System Services Planning</i>, GA22-7800 • <i>z/OS Planning for Installation</i>, GA22-7504
Installing	<ul style="list-style-type: none"> • <i>z/OS Program Directory</i> • <i>z/OS Planning for Installation</i>, GA22-7504 • <i>z/OS Language Environment Customization</i>, SA22-7564
Option customization	<ul style="list-style-type: none"> • <i>z/OS XL C/C++ User's Guide</i>, SC09-4767
Coding programs	<ul style="list-style-type: none"> • <i>z/OS XL C/C++ Run-Time Library Reference</i>, SA22-7821 • <i>z/OS XL C/C++ Language Reference</i>, SC09-4815 • <i>z/OS XL C/C++ Programming Guide</i>, SC09-4765 • <i>z/OS Metal C Programming Guide and Reference</i>, SA23-2225 • <i>z/OS Language Environment Concepts Guide</i>, SA22-7567 • <i>z/OS Language Environment Programming Guide</i>, SA22-7561 • <i>z/OS Language Environment Programming Reference</i>, SA22-7562
Coding and binding programs with interlanguage calls	<ul style="list-style-type: none"> • <i>z/OS XL C/C++ Programming Guide</i>, SC09-4765 • <i>z/OS XL C/C++ Language Reference</i>, SC09-4815 • <i>z/OS Language Environment Programming Guide</i>, SA22-7561 • <i>z/OS Language Environment Writing Interlanguage Communication Applications</i>, SA22-7563 • <i>z/OS MVS Program Management: User's Guide and Reference</i>, SA22-7643 • <i>z/OS MVS Program Management: Advanced Facilities</i>, SA22-7644
Compiling, binding, and running programs	<ul style="list-style-type: none"> • <i>z/OS XL C/C++ User's Guide</i>, SC09-4767 • <i>z/OS Language Environment Programming Guide</i>, SA22-7561 • <i>z/OS Language Environment Debugging Guide</i>, GA22-7560 • <i>z/OS MVS Program Management: User's Guide and Reference</i>, SA22-7643 • <i>z/OS MVS Program Management: Advanced Facilities</i>, SA22-7644
Compiling and binding applications in the z/OS UNIX System Services (z/OS UNIX) environment	<ul style="list-style-type: none"> • <i>z/OS XL C/C++ User's Guide</i>, SC09-4767 • <i>z/OS UNIX System Services User's Guide</i>, SA22-7801 • <i>z/OS UNIX System Services Command Reference</i>, SA22-7802 • <i>z/OS MVS Program Management: User's Guide and Reference</i>, SA22-7643 • <i>z/OS MVS Program Management: Advanced Facilities</i>, SA22-7644

Table 4. Documents by task (continued)

Tasks	Documents
Debugging programs	<ul style="list-style-type: none"> • README file • <i>z/OS XL C/C++ User's Guide</i>, SC09-4767 • <i>z/OS XL C/C++ Messages</i>, GC09-4819 • <i>z/OS XL C/C++ Programming Guide</i>, SC09-4765 • <i>z/OS Language Environment Programming Guide</i>, SA22-7561 • <i>z/OS Language Environment Debugging Guide</i>, GA22-7560 • <i>z/OS Language Environment Run-Time Messages</i>, SA22-7566 • <i>z/OS UNIX System Services Messages and Codes</i>, SA22-7807 • <i>z/OS UNIX System Services User's Guide</i>, SA22-7801 • <i>z/OS UNIX System Services Command Reference</i>, SA22-7802 • <i>z/OS UNIX System Services Programming Tools</i>, SA22-7805 • Debug Tool documentation, available on the Debug Tool Library page on the World Wide Web (www.ibm.com/software/awdtools/debugtool/library/) • z/OS messages database, available on the z/OS Library page at http://www.ibm.com/servers/eserver/zseries/zos/bkserv/ through the LookAt Internet message search utility.
Developing debuggers and profilers	<ul style="list-style-type: none"> • <i>z/OS Common Debug Architecture User's Guide</i>, SC09-7653 • <i>z/OS Common Debug Architecture Library Reference</i>, SC09-7654 • <i>DWARF/ELF Extensions Library Reference</i>, SC09-7655
Packaging XL C/C++ applications	<ul style="list-style-type: none"> • <i>z/OS XL C/C++ Programming Guide</i>, SC09-4765 • <i>z/OS XL C/C++ User's Guide</i>, SC09-4767
Using shells and utilities in the z/OS UNIX System Services environment	<ul style="list-style-type: none"> • <i>z/OS XL C/C++ User's Guide</i>, SC09-4767 • <i>z/OS UNIX System Services Command Reference</i>, SA22-7802 • <i>z/OS UNIX System Services Messages and Codes</i>, SA22-7807
Using sockets library functions in the z/OS UNIX System Services environment	<ul style="list-style-type: none"> • <i>z/OS XL C/C++ Run-Time Library Reference</i>, SA22-7821
Using the ISO Standard C++ Library to write portable C/C++ code that complies with ISO standards	<ul style="list-style-type: none"> • <i>Standard C++ Library Reference</i>, SC09-4949
Migrating from the IBM Open Class Library to the Standard C++ Library	<ul style="list-style-type: none"> • <i>IBM Open Class Library Transition Guide</i>, SC09-4948
Porting a z/OS UNIX System Services application to z/OS	<ul style="list-style-type: none"> • <i>z/OS UNIX System Services Porting Guide</i> This guide contains useful information about supported header files and C functions, sockets in z/OS UNIX System Services, 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 web address: www.ibm.com/servers/eserver/zseries/zos/unix/bpxa1por.html
Working in the z/OS UNIX System Services Parallel Environment	<ul style="list-style-type: none"> • <i>z/OS UNIX System Services Parallel Environment: Operation and Use</i>, SA22-7810 • <i>z/OS UNIX System Services Parallel Environment: MPI Programming and Subroutine Reference</i>, SA22-7812
Performing diagnosis and submitting an Authorized Program Analysis Report (APAR)	<ul style="list-style-type: none"> • <i>z/OS XL C/C++ User's Guide</i>, SC09-4767 • CBC.SCCND0C(APAR) on z/OS XL C/C++ product tape
<p>Note: For information on using the prelinker, see the appendix on prelinking and linking z/OS XL C/C++ programs in <i>z/OS XL C/C++ User's Guide</i>.</p>	

Softcopy documents

The z/OS XL C/C++ publications are supplied in PDF and BookMaster[®] formats on the following CD: *z/OS Collection*, SK3T-4269. They are also available at www.ibm.com/software/awdtools/czos/library/.

To read a PDF file, use the Adobe[®] Reader. If you do not have the Adobe Reader, you can download it (subject to Adobe license terms) from the Adobe Web site at www.adobe.com.

You can also browse the documents on the World Wide Web by visiting the z/OS library at <http://www.ibm.com/servers/eserver/zseries/zos/bkserv/>.

Note: For further information on viewing and printing softcopy documents and using BookManager[®], see *z/OS Information Roadmap*.

Softcopy examples

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

- *z/OS XL C/C++ Language Reference*, SC09-4815
- *z/OS XL C/C++ User's Guide*, SC09-4767
- *z/OS XL C/C++ Programming Guide*, SC09-4765

In the following documents, a label on an example indicates that the example is distributed as a softcopy file:

- *z/OS XL C/C++ Language Reference*, SC09-4815
- *z/OS XL C/C++ Programming Guide*, SC09-4765
- *z/OS XL C/C++ User's Guide*, SC09-4767

The label is the name of a member in the CBC.SCCNSAM data set. The labels begin with the form CCN or CLB. Examples labelled as CLB appear only in the *z/OS XL C/C++ User's Guide*, while examples labelled as CCN appear in all three documents, and are further distinguished by *x* following CCN, where *x* represents one of the following:

- R and X refer to *z/OS XL C/C++ Language Reference*, SC09-4815
- G refers to *z/OS XL C/C++ Programming Guide*, SC09-4765
- U refers to *z/OS XL C/C++ User's Guide*, SC09-4767

z/OS XL C/C++ on the World Wide Web

Additional information on z/OS XL C/C++ is available on the World Wide Web on the z/OS XL C/C++ home page at: www.ibm.com/software/awdtools/czos/

This page contains late-breaking information about the z/OS XL C/C++ product, including the compiler, the C/C++ libraries, and utilities. There are links to other useful information, such as the z/OS XL C/C++ information library and the libraries of other z/OS elements that are available on the Web. The z/OS XL C/C++ home page also contains links to other related Web sites.

Where to find more information

Please see *z/OS Information Roadmap* for an overview of the documentation associated with z/OS, including the documentation available for z/OS Language Environment.

Using LookAt to look up message explanations

LookAt is an online facility that lets you look up explanations for most of the IBM messages you encounter, as well as for some system abends and codes. Using LookAt to find information is faster than a conventional search because in most cases LookAt goes directly to the message explanation.

You can use LookAt from these locations to find IBM message explanations for z/OS elements and features, z/VM[®], VSE/ESA[™], and Clusters for AIX[®] and Linux[®]:

- The Internet. You can access IBM message explanations directly from the LookAt Web site at www.ibm.com/servers/eserver/zseries/zos/bkserv/lookat/.
- Your z/OS TSO/E host system. You can install code on your z/OS systems to access IBM message explanations using LookAt from a TSO/E command line (for example: TSO/E prompt, ISPF, or z/OS UNIX System Services).
- Your Microsoft[®] Windows[®] workstation. You can install LookAt directly from the *z/OS Collection* (SK3T-4269) or the *z/OS and Software Products DVD Collection* (SK3T-4271) and use it from the resulting Windows graphical user interface (GUI). The command prompt (also known as the DOS > command line) version can still be used from the directory in which you install the Windows version of LookAt.
- Your wireless handheld device. You can use the LookAt Mobile Edition from www.ibm.com/servers/eserver/zseries/zos/bkserv/lookat/lookatm.html with a handheld device that has wireless access and an Internet browser (for example: Internet Explorer for Pocket PCs, Blazer or Eudora for Palm OS, or Opera for Linux handheld devices).

You can obtain code to install LookAt on your host system or Microsoft Windows workstation from:

- A CD-ROM in the *z/OS Collection* (SK3T-4269).
- The *z/OS and Software Products DVD Collection* (SK3T-4271).
- The LookAt Web site (click **Download** and then select the platform, release, collection, and location that suit your needs). More information is available in the LOOKAT.ME files available during the download process.

Using IBM Health Checker for z/OS

IBM Health Checker for z/OS is a z/OS component that installations can use to gather information about their system environment and system parameters to help identify potential configuration problems before they impact availability or cause outages. Individual products, z/OS components, or ISV software can provide checks that take advantage of the IBM Health Checker for z/OS framework. This book might refer to checks or messages associated with this component.

For additional information about checks and about IBM Health Checker for z/OS, see *IBM Health Checker for z/OS: User's Guide*. Starting with z/OS V1R4, z/OS users can obtain the IBM Health Checker for z/OS from the z/OS Downloads page at <http://www.ibm.com/servers/eserver/zseries/zos/downloads/>.

SDSF also provides functions to simplify the management of checks. See *z/OS SDSF Operation and Customization* for additional information.

Information updates on the web

For the latest information updates that have been provided in PTF cover letters and Documentation APARs for z/OS, see the online document at: http://publibz.boulder.ibm.com/cgi-bin/bookmgr_OS390/BOOKS/ZIDOCMST/CCONTENTS

This document is updated weekly and lists documentation changes before they are incorporated into z/OS publications.

Technical support

Additional technical support is available from the z/OS XL C/C++ Support page. This page provides a portal with search capabilities to a large selection of technical support FAQs and other support documents. You can find the z/OS XL C/C++ Support page on the Web at:

www.ibm.com/software/awdtools/czos/support

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

compinfo@ca.ibm.com

For the latest information about z/OS XL C/C++, visit the product information site at:

www.ibm.com/software/awdtools/czos/

How to send your comments

Your feedback is important in helping to provide accurate and high-quality information. If you have any comments about this document or any other z/OS XL C/C++ documentation, send your comments by e-mail to:

compinfo@ca.ibm.com

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

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.

About IBM z/OS XL C/C++

The XL C/C++ feature of the IBM z/OS licensed program provides support for C and C++ application development on the z/OS platform.

z/OS XL C/C++ includes:

- A C compiler (referred to as the z/OS XL C compiler)
- A C++ compiler (referred to as the z/OS XL C++ compiler)
- Performance Analyzer host component, which supports the IBM C/C++ Productivity Tools for OS/390® product
- A set of utilities for C/C++ application development

Notes:

1. The Run-Time Library Extensions base element was introduced in z/OS V1R5. It includes the Common Debug Architecture (CDA) Libraries, the c89 utility, and, as of z/OS V1R6, the xlc utility. The Common Debug Architecture provides a consistent and common format for debugging information across the various languages and operating systems that are supported on the IBM System z platform. Run-Time Library Extensions also includes legacy libraries to support existing programs. These are the UNIX System Laboratories (USL) I/O Stream Library and USL Complex Mathematics Library. The IBM Open Class Library is not supported.
2. The Standard C++ Library is included with Language Environment libraries.
3. The z/OS XL C/C++ compiler works with the mainframe interactive Debug Tool product and WebSphere® Developer for System z, integrated with IBM Debug Tool for z/OS and IBM Debug Tool Utilities and Advanced Functions for z/OS.

IBM offers the C and C++ compilers on other platforms, such as the AIX, Linux, OS/400®, and z/VM operating systems. The C compiler is also available on the VSE/ESA platform.

Changes for z/OS V1R9

z/OS XL C/C++ has made the following performance and usability enhancements for the z/OS V1R9 release:

z/OS XL C support for system program development – METAL option

Prior to V1R9, z/OS XL C compiler-generated code required Language Environment for C run-time library functions and required the establishment of an overall execution context, including the heap storage and dynamic storage area. This dependency meant that the XL C compiler could only generate code to run in an environment where Language Environment services existed.

The METAL compiler option, which is provided with z/OS V1R9, generates code that does not have Language Environment run-time dependencies. In addition, language features are provided to embed small pieces of HLASM source within C statements. Most assembler macros can be used in this embedded HLASM code. Function prolog and epilog code can be provided for all functions that have extern scope. When applied together, these features can assist with writing programs suitable for use in an MVS™ system level environment, as well as for inter-operating with most programs written in HLASM. Any system services that the program needs can be obtained directly by calling Assembler Services.

z/OS XL C/C++ support for decimal floating-point formats (DFP)

z/OS V1R9 XL C/C++ supports the decimal floating-point formats in addition to the current hex and binary floating-point formats. The decimal formats are specified by the revised IEEE 754 Floating Point Standard. This support assists with avoiding potential rounding problems, which can result from using binary or hexadecimal floating-point types to handle decimal calculations.

z/OS XL C/C++ provides the following:

- Decimal type specifiers through the `_Decimal32`, `_Decimal64`, and `_Decimal128` reserved keywords
- Decimal literal support
- Conversion between decimal types and the other floating-point types

CDAHLASM utility to produce debug information

The CDAHLASM utility produces debug information in DWARF format and ADATA format. Debuggers can use the DWARF debug information to debug Metal C applications. The CDAASMC cataloged procedure is provided to execute this utility.

New compiler options

z/OS V1R9 XL C/C++ introduces the following new compiler options:

- `ARMODE` (C compiler only)
- `ASMDATASIZE` (C compiler only)
- `ASSERT(RESTRICT)`
- `DFP`
- `EPILOG` (C compiler only)
- `GENASM` (C compiler only)
- `METAL` (C compiler only)
- `PROLOG` (C compiler only)
- `RESERVED_REG` (C compiler only)

New compiler suboption

z/OS V1R9 XL C/C++ introduces the following new compiler suboption:

- `TARGET(zOSV1R9)`

New pragma directives

z/OS V1R9 XL C introduces the following new pragma directives:

- `#pragma epilog`
- `#pragma prolog`

Performance improvements

As an ongoing effort, improvements are provided in the generated code to take advantage of new instructions in the hardware architecture. You can benefit from these improvements by recompiling your existing source without code changes. Additional built-in functions are provided to help programs use selected hardware instructions directly. For information on these built-in functions, see the built-in functions described in *z/OS XL C/C++ Programming Guide*.

For information on the changes that the Language Environment element has made for z/OS V1R9, see "What's New in Language Environment for z/OS" in *z/OS Language Environment Concepts Guide*.

The XL C/C++ compilers

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

The C language

The C language is a general purpose, versatile, and functional programming language that 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 and includes all of the advantages of C listed above. In addition, C++ also supports object-oriented concepts, generic types or templates, and an extensive library.

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 z/OS XL C and XL C++ compilers

The XL C and XL C++ compilers, when used with the Language Environment element, offer many features to increase your productivity and improve program execution times:

- Optimization support:
 - Extra Performance Linkage (XPLINK) function calling convention, which has the potential for a significant performance increase when used in an environment of frequent calls between small functions. XPLINK makes subroutine calls more efficient by removing non-essential instructions from the main path.
 - Algorithms to take advantage of the zSeries® architecture to achieve improved optimization and memory usage through the OPTIMIZE and IPA compiler options.
 - The OPTIMIZE compiler option, which instructs the compiler to optimize the machine instructions it generates to try to produce faster-running object code and improve application performance at run time.
 - Interprocedural Analysis (IPA), to perform optimizations across procedural and compilation unit boundaries, thereby optimizing application performance at run time.

- Additional optimization capabilities are available with the INLINE compiler option.

- DLLs (dynamic link libraries) to share parts among applications or parts of applications, and dynamically link to exported variables and functions at run time. 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 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 Link Pack Area (LPA) or the Extended Link Pack Area (ELPA) 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. z/OS XL C programmers can design programs that are naturally reentrant. For those programs that are not naturally reentrant, z/OS XL C programmers can use constructed reentrancy. To do this, compile programs with the RENT option and use the program management binder supplied with z/OS or the Language Environment prelinker and program management binder. The z/OS XL C++ compiler always uses the constructed reentrancy algorithms.

- Locale-based internationalization support derived from *IEEE POSIX 1003.2-1992* standard. Also derived from *X/Open CAE Specification, System Interface Definitions, Issue 4* and *Issue 4 Version 2*. This allows you 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, compiled Java™, and Fortran, to enable you to integrate z/OS XL C/C++ code with existing applications.
- Exploitation of z/OS and z/OS UNIX System Services technology.
z/OS UNIX System Services is the IBM implementation of the open operating system environment, as defined in the XPG4 and POSIX standards.
- Support features in the following standards at the system level:
 - *ISO/IEC 9899:1999*
 - *ISO/IEC 9945-1:1990 (POSIX-1)/IEEE POSIX 1003.1-1990*
 - The core features of *IEEE POSIX 1003.1a, Draft 6, July 1991*
 - *IEEE Portable Operating System Interface (POSIX) Part 2, P1003.2*
 - The core features 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*
 - The core features of *IEEE 754-1985 (R1990) IEEE Standard for Binary Floating-Point Arithmetic (ANSI)*, as applicable to the IBM System z environment.
 - *X/Open CAE Specification, Networking Services, Issue 4*
 - A subset of *IEEE Std. 1003.1-2001 (Single UNIX Specification, Version 3)*.
- Support for the Euro currency

z/OS XL C compiler-specific features

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

- The ability to write portable code that supports the following standards:

- ISO/IEC 9899:1999
- ANSI/ISO 9899:1990[1992] (formerly ANSI X3.159-1989 C)
- X/Open Specification Programming Languages, Issue 3, Common Usage C
- FIPS-160
- System programming capabilities, which allow you to use z/OS XL C in place of assembler
- Extensions of the standard definitions of the C language to provide you with support for the z/OS environment, such as fixed-point (packed) decimal data support

z/OS XL C++ compiler-specific features

In addition to the features common to z/OS XL C and XL C++, the z/OS XL C++ compiler supports the *Programming languages - C++ (ISO/IEC 14882:1998)* standard. Also, it further conforms to the *Programming languages - C++ (ISO/IEC 14882:2003(E))* standard, which incorporates the latest Technical Corrigendum 1.

Class libraries

z/OS V1R9 XL C/C++ uses the following thread-safe class libraries:

- Standard C++ Library, including the Standard Template Library (STL), and other library features of *Programming languages - C++ (ISO/IEC 14882:1998)* and *Programming languages - C++ (ISO/IEC 14882:2003(E))*
- UNIX System Laboratories (USL) C++ Language System Release I/O Stream and Complex Mathematics Class Libraries

Note: As of z/OS V1R5, all application development using the C/C++ IBM Open Class Library (Application Support Class and Collection Class Libraries) is not supported. As of z/OS V1R9, the execution of applications using the C/C++ IBM Open Class Library is not supported. For additional information, see *IBM Open Class Library Transition Guide*.

For new code and enhancements to existing applications, the Standard C++ Library should be used. The Standard C++ Library includes the following:

- Stream classes for performing input and output (I/O) operations
- The Standard C++ Complex Mathematics Library for manipulating complex numbers
- The Standard Template Library (STL) which is composed of C++ template-based algorithms, container classes, iterators, localization objects, and the string class

Utilities

The z/OS XL C/C++ compilers provide the following utilities:

- The xlc utility to invoke the compiler using a customizable configuration file.
- The c89 utility to invoke the compiler using host environment variables.
- The CXXFILT utility to map z/OS XL C++ mangled names to their original function names.
- The DSECT conversion utility to convert descriptive assembler DSECTs into z/OS XL C/C++ data structures.
- The makedepend utility to derive all dependencies in the source code and write these into the makefile. The **make** command will determine which source files to recompile, whenever a dependency has changed. This frees the user from manually monitoring such changes in the source code.

- The CDAHLASM utility, which produces debug information in DWARF (for Metal C applications) and ADATA formats. This utility uses the HLASM assembler to compile the source files produced by compiling Metal C code.

Language Environment utilities include:

- The object library utility (C370LIB; also known as EDCALIAS) to update partitioned data set (PDS and PDSE) libraries of object modules. The object library utility supports XPLINK, IPA, and LP64 compiled objects.
- The prelinker which combines object modules that comprise a z/OS XL C/C++ application to produce a single object module. The prelinker supports only object and extended object format input files, and does not support GOFF.

Note: IBM has stabilized the prelinker. Further enhancements will not be made to the prelinker utility. IBM recommends that you use the binder instead of the prelinker and linker.

dbx

You can use the **dbx** shell command to debug programs, as described in *z/OS UNIX System Services Programming Tools* and *z/OS UNIX System Services Command Reference*.

Refer to www.ibm.com/servers/eserver/zseries/zos/unix/bpxa1dbx.html for further information on dbx.

Language Environment element

z/OS XL C/C++ exploits the C/C++ run-time environment and library of run-time services available with the Language Environment element provided with z/OS.

The Language Environment element consists of four language-specific run-time libraries, and Base Routines and Common Services, as shown in Figure 1. This element establishes a common run-time environment and common run-time services for language products, user programs, and other products.

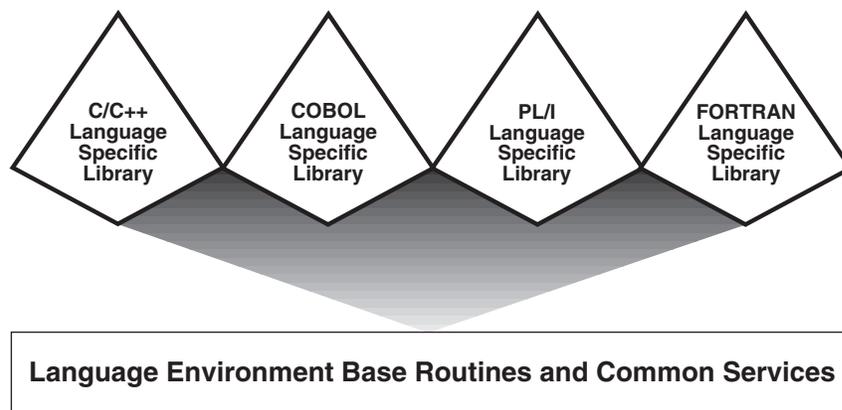


Figure 1. Language Environment libraries

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. Language Environment services include:

- 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. z/OS XL C/C++ contains these functions within a library of callable routines, and includes interfaces to operating system functions and a variety of other commonly used functions.
- Run-time options that help in the execution, performance, and diagnosis of your application.
- Access to operating system services; z/OS UNIX System Services is available to you or your program through the z/OS XL C/C++ language bindings.
- Access to language-specific library routines, such as the z/OS XL C/C++ library functions.

Note: The Language Environment run-time option TRAP(ON) should be set when using z/OS XL C/C++. Refer to *z/OS Language Environment Programming Reference* for details on the Language Environment run-time options.

Language Environment downward compatibility

Language Environment downward compatibility support is provided. Assuming that you have met the required programming guidelines and restrictions, described in *z/OS Language Environment Programming Guide*, this support enables you to develop applications on higher release levels of z/OS for use on platforms that are running lower release levels of z/OS. In XL C and XL C++, downward compatibility support is provided through the XL C/C++ TARGET compiler option. See TARGET in *z/OS XL C/C++ User's Guide* for details on this compiler option.

For example, a company may use z/OS V1R9 with the Language Environment element on a development system where applications are coded, link-edited, and tested, while using any supported lower Language Environment release on their production systems where the finished application modules are used.

Downward compatibility support is not the roll-back of new function to prior releases of the operating system. Applications developed that exploit the downward compatibility support must not use any Language Environment function that is unavailable on the lower release of z/OS where the application will be used.

The downward compatibility support includes toleration PTFs for lower releases of z/OS to assist in diagnosing applications that do not meet the programming requirements for this support. (Specific PTF numbers can be found in the PSP buckets.)

The diagnosis assistance that will be provided by the toleration PTFs includes detection of an unsupported program object format. If the program object format is at a level which is not supported by the target deployment system, then the deployment system will produce an abend when trying to load the application program. The abend will indicate that the Data Facility Storage Management Subsystem (DFSMS™) software was unable to find or load the application program. Correcting this problem does not require the installation of any toleration PTFs. Instead, you will need to recreate the program object that is compatible with the older deployment system.

Language Environment downward compatibility support and the downward compatibility support that is provided by toleration PTFs does not change upward

compatibility. That is, applications coded and link-edited with one Language Environment release will continue to run on later Language Environment releases without the need to recompile or re-link edit the application, independent of the downward compatibility support.

The current z/OS level header files and SYSLIB can be used (the user no longer has to copy header files and SYSLIB data sets from the deployment release).

Note: As of z/OS V1R3, the executables produced with the binder's COMPAT=CURRENT setting will not run on lower levels of z/OS. You will have to explicitly override to a particular program object level, or use the COMPAT=MIN setting introduced in z/OS V1R3.

About prelinking, linking, and binding

When describing the process to build an application, this document refers to the *bind step*.

Normally, the program management binder is used to perform the bind step. However, in many cases the prelink and link steps can be used in place of the bind step. When they cannot be substituted, and the program management binder alone must be used, it will be stated. For more information, refer to Prelinking and linking z/OS XL C/C++ programs and Binding z/OS XL C/C++ programs in *z/OS XL C/C++ User's Guide*.

The terms *bind* and *link* have multiple meanings.

- With respect to building an application:

In both instances, the program management binder is performing the actual processing of converting the object file(s) into the application executable module. Object files with longname symbols, reentrant writable static symbols, and DLL-style function calls require additional processing to build global data for the application.

The term *link* refers to the case where the binder does not perform this additional processing, due to one of the following:

- The processing is not required, because none of the object files in the application use constructed reentrancy, use long names, are DLL or are C++.
- The processing is handled by executing the prelinker step before running the binder.

The term *bind* refers to the case where the binder is required to perform this processing.

- With respect to executing code in an application:

The *linkage definition* refers to the program call linkage between program functions and methods. This includes the passing of control and parameters. Refer to "Program Linkage" for more information on linkage specification.

Some platforms have a single linkage convention. z/OS has a number of linkage conventions, including standard operating system linkage, Extra Performance Linkage (XPLINK), and different non-XPLINK linkage conventions for C and C++.

Notes on the prelinking process

You cannot use the prelinker if you are using the XPLINK, GOFF, or LP64 compiler options. IBM recommends using the binder instead of the prelinker whenever possible.

The prelinker was designed to process long names and support constructed reentrancy in earlier versions of the C compiler on the MVS and OS/390 operating systems. The Language Environment prelinker provides output that is compatible with the linkage editor, which is shipped with the binder.

The *binder* is designed to include the functions of the prelinker, the linkage editor, the loader, and a number of APIs to manipulate the program object. Thus, the binder is a superset of the linkage editor. Its functionality provides a high level of compatibility with the prelinker and linkage editor, but provides additional functionality in some areas. Generally, the terms *binding* and *linking* are interchangeable. In particular, the binder supports:

- Inputs from the object module
- XOBJ, GOFF, load module and program object
- Auto call resolutions from z/OS UNIX archives and C370LIB object directories
- Long external names
- All prelinker control statements

Notes:

1. You need to use the binder for XPLINK objects.
2. As of z/OS V1R7, the Hierarchical File System (HFS) functionality has been stabilized and zSeries File System (zFS) is the strategic file system for z/OS UNIX System Services. The term *z/OS UNIX file system* includes both HFS and zFS.

For more information on the compatibility between the binder, and the linker and prelinker, see *z/OS MVS Program Management: User's Guide and Reference*.

Updates to the prelinking, linkage-editing, and loading functions that are performed by the binder are delivered through the binder. If you use the Language Environment prelinker and the linkage editor (supplied through the binder), you have to apply the latest maintenance for the Language Environment element as well as the binder.

If you still need to use the prelinker and linkage editor, see Prelinker and linkage editor options in *z/OS XL C/C++ User's Guide*.

File format considerations

You can use the binder in place of the prelinker and linkage editor but there are exceptions, which are file format considerations. For further information, on when you cannot use the binder, see Binding z/OS XL C/C++ programs in *z/OS XL C/C++ User's Guide*.

The program management binder

The binder provided with z/OS combines the object modules, load modules, and program objects comprising an application. It produces a single z/OS 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 member or a z/OS UNIX System Services file.

If you cannot use a PDSE member or z/OS UNIX file, and your program contains C++ code, or C code that is compiled with any of the RENT, LONGNAME, DLL or IPA compiler options, you must use the prelinker. C and C++ code compiled with the GOFF or XPLINK compiler options cannot be processed by 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 Language Environment prelinker combines the object modules comprising a z/OS XL 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 (which is stored in a PDS, PDSE, or z/OS UNIX file).

z/OS UNIX System Services

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

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

z/OS UNIX provides support for both existing z/OS applications and new z/OS UNIX applications through the following:

- C programming language support as defined by ISO C
- C++ programming language support as defined by ISO C++
- 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)
- z/OS UNIX extensions that provide z/OS-specific support beyond the defined standards
- The z/OS UNIX Shell and Utilities feature, which provides:
 - A shell, based on the Korn Shell and compatible with the Bourne Shell
 - A shell, `tcsh`, based on the C shell, `csh`
 - Tools and utilities that support the *X/Open Single UNIX Specification*, also known as *X/Open Portability Guide (XPG) Version 4, Issue 2*, and provide z/OS support. The following is a partial list of utilities that are included:

ar	Creates and maintains library archives
as	Invokes HLASM to create assembler applications
BPXBATCH	Allows you to submit batch jobs that run shell commands, scripts, or z/OS XL C/C++ executable files in z/OS UNIX files from a shell session

c89	Uses host environment variables to compile, assemble, and bind z/OS UNIX, C/C++ and assembler applications
dbx	Provides an environment to debug and run programs
gencat	Merges the message text source files (usually *.msg) into a formatted message catalog file (usually *.cat)
iconv	Converts characters from one code set to another
ld	Combines object files and archive files into an output executable file, resolving external references
lex	Automatically writes large parts of a lexical analyzer based on a description that is supplied by the programmer
localedef	Creates a compiled locale object
make	Helps you manage projects containing a set of interdependent files, such as a program with many z/OS source and object files, keeping all such files up to date with one another
xlc	Allows you to invoke the compiler using a customizable configuration file
yacc	Allows you to write compilers and other programs that parse input according to strict grammar rules

– Support for other utilities such as:

dspcat	Displays all or part of a message catalog
dspmsg	Displays a selected message from a message catalog
mkcatdefs	Preprocesses a message source file for input to the gencat utility
runcat	Invokes mkcatdefs and pipes the message catalog source data (the output from mkcatdefs) to gencat

- Access to the Hierarchical File System (HFS), with support for the POSIX.1 and XPG4 standards
- Access to the zSeries File System (zFS), which provides performance improvements over HFS, and also supports the POSIX.1 and XPG4 standards
- z/OS XL C/C++ I/O routines, which support using z/OS UNIX files, standard z/OS data sets, or a mixture of both
- Application threads (with support for a subset of POSIX.4a)
- Support for z/OS XL C/C++ DLLs

z/OS UNIX System Services 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.

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

z/OS XL C/C++ applications with z/OS UNIX System Services C functions

All z/OS UNIX System Services C functions are available at all times. In some situations, you must specify the POSIX(ON) run-time option. This is required for the POSIX.4a threading functions, the POSIX `system()` function, and signal handling functions where the behavior is different between POSIX/XPG4 and ISO. Refer to *z/OS XL C/C++ Run-Time Library Reference* for more information about requirements for each function.

You can invoke a z/OS XL C/C++ program that uses z/OS UNIX C functions using the following methods:

- Directly from a shell.
- From another program, or from a 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) run-time option.

Input and output

The z/OS XL C/C++ run-time library that supports the z/OS XL C/C++ compiler supports different input and output (I/O) interfaces, file types, and access methods. The Standard C++ Library provides additional support.

I/O interfaces

The z/OS XL C/C++ run-time library supports the following I/O interfaces:

C Stream I/O

This is the default and the ISO-defined I/O method. This method processes all input and output on a per-character basis.

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 a z/OS XL C/C++ extension to the ISO standard.

TCP/IP Sockets I/O

z/OS UNIX System Services 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 z/OS UNIX sockets. z/OS UNIX sockets correspond closely to the sockets used by UNIX applications that use the Berkeley Software Distribution (BSD) 4.3 standard (also known as Berkeley 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 z/OS 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 z/OS, 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 each other in the network using TCP/IP.

In addition, the Standard C++ Library provides stream classes, which support 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 z/OS XL C/C++ run-time library supports the following file types:

Virtual Storage Access Method (VSAM) data sets

z/OS XL C/C++ has native support for the following VSAM data sets:

- 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 reverse order).
- Relative-Record Data Sets (RRDS). Use RRDS for data in which each item has a particular number (for example, a telephone system where a record is associated with each telephone number).

For more information on how to perform I/O operations on these VSAM file types, see "Performing VSAM I/O operations" in *z/OS XL C/C++ Programming Guide*.

Hierarchical File System files

z/OS XL C/C++ recognizes Hierarchical File System (HFS) file names. The name specified on the `fopen()` or `freopen()` call has to conform to certain rules. See "Opening Files" in *z/OS XL C/C++ Programming Guide* for the details of these rules. You can create regular HFS files, special character HFS files, or FIFO HFS files. You can also create links or directories.

Note: As of z/OS V1R7, the Hierarchical File System (HFS) functionality has been stabilized and zSeries File System (zFS) is the strategic UNIX System Services file system for z/OS.

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 GB of contiguous virtual storage space. A program can use this storage as a buffer (1 gigabyte(GB) = 2^{30} bytes).

zSeries File System

zSeries File System (zFS) is a z/OS UNIX file system that can be used in addition to the Hierarchical File System (HFS). zFS may provide performance gains in accessing files that are frequently accessed and updated. The I/O functions in the z/OS XL C/C++ run-time library support zFS.

Additional I/O features

z/OS XL C/C++ provides additional I/O support through the following features:

- Large file support, which enables I/O to and from z/OS UNIX System Services files that are larger than 2 GB (see `_LARGE_FILES` macro description in “Macros related to compiler option settings” on page 459)
- User error handling for serious I/O failures (SIGIOERR)
- Improved sequential data access performance through enablement of the DFSMS software support for 31-bit sequential data buffers and sequential data striping on extended format data sets
- Full support of PDSEs on z/OS (including support for multiple members opened for write)
- Overlapped I/O support under z/OS (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 (SPC) facility allows you to build applications that do not require dynamic loading of Language Environment libraries. It also allows you to tailor your application for better utilization of the low-level services available on your operating system. SPC offers a number of advantages:

- You can develop applications that can be executed in a customized environment rather than with Language Environment services. Note that if you do not use Language Environment services, only some built-in functions and a limited set of z/OS XL C/C++ run-time library functions are available to you.
- You can substitute the z/OS XL C language in place of assembly language when writing system exit routines by using the interfaces that are provided by SPC.
- SPC lets you develop applications featuring a user-controlled environment in which a z/OS XL 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 independent of the user. The application is then suspended when control is returned to the user application.

Interaction with other IBM products

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

- CICS Transaction Server for z/OS
You can use the CICS 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.
- DB2 Universal Database™ for z/OS
DB2 programs manage data that is stored in relational databases. You can access the data by using a structured set of queries that are written in Structured Query Language (SQL).

A DB2 program uses SQL statements that are embedded in the application program. The SQL translator (DB2 preprocessor) translates the embedded SQL into host language statements, which are then compiled by the z/OS XL C/C++ compilers. Alternatively, use the SQL compiler option to compile a DB2 program with embedded SQL without using the DB2 preprocessor. The DB2 program processes requests, then returns control to the application program.

- Debug Tool

z/OS XL C/C++ supports program development by using the Debug Tool. This tool allows you to debug applications in their native host environment, such as CICS Transaction Server for z/OS, IMS, and DB2. 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 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 z/OS UNIX System Services files as source files.

For further information, see www.ibm.com/software/awdtools/debugtool/.

- WebSphere Developer for System z

z/OS V1R7 XL C/C++ and later releases enable you to use WebSphere Developer for System z V7.0 to improve the efficiency of application development. For information on WebSphere Developer for System z, see: <http://www.ibm.com/software/awdtools/devzseries/>.

- IBM C/C++ Productivity Tools for OS/390

Note: Starting with z/OS V1R5, both the C/C++ compiler optional feature and the Debug Tool product will need to be installed if you wish to use IBM C/C++ Productivity Tools for OS/390. For more information on Debug Tool, refer to www.ibm.com/software/awdtools/debugtool/.

With the IBM C/C++ Productivity Tools for OS/390 product, you can expand your z/OS application development environment out to the workstation, while remaining close to your familiar host environment. IBM C/C++ Productivity Tools for OS/390 includes the following workstation-based tools to increase your productivity and code quality:

- Performance Analyzer to help you analyze, understand, and tune your C and C++ applications for improved performance
- Distributed Debugger that allows you to debug C or C++ programs from the convenience of the workstation
- Workstation-based editor to improve the productivity of your C and C++ source entry
- Advanced online help, with full text search and hypertext topics as well as printable, viewable, and searchable Portable Document Format (PDF) documents

Note: References to *Performance Analyzer* in this document refer to the IBM OS/390 Performance Analyzer included in the IBM C/C++ Productivity Tools for OS/390 product.

In addition, IBM C/C++ Productivity Tools for OS/390 includes the following host components:

- Debug Tool
- Host Performance Analyzer

Use the Performance Analyzer on your workstation to graphically display and analyze a profile of the execution of your host z/OS XL C or C++ application. Use this information to time and tune your code so that you can increase the performance of your application.

Use the Distributed Debugger to debug your z/OS XL C or C++ application remotely from your workstation. Set a breakpoint with the simple click of the mouse. Use the windowing capabilities of your workstation to view multiple segments of your source and your storage, while monitoring a variable at the same time.

Use the workstation-based editor to quickly develop C and C++ application code that runs on z/OS. Context-sensitive help information is available to you when you need it.

- IBM Fault Analyzer for z/OS

The IBM Fault Analyzer helps developers analyze and fix application and system failures. It gathers information about an application and the surrounding environment at the time of the abend, providing the developer with valuable information needed for developing and testing new and existing applications. For more information, refer to: www.ibm.com/software/awdtools/faultanalyzer/.

- Application Performance Analyzer for z/OS

The Application Performance Analyzer for z/OS is an application program performance analysis tool that helps you to:

- Optimize the performance of your existing application
- Improve the response time of your online transactions and batch turnaround times
- Isolate performance problems in applications

For more information, refer to: www.ibm.com/software/awdtools/apa/.

- ISPF Software Configuration and Library Manager facility (SCLM)

The ISPF Software Configuration and Library Manager facility (SCLM) maintains information about the source code, objects and load modules. It also keeps track of other relationships in your application, such as test cases, JCL, and publications. The SCLM Build function translates input to output, managing not only compilation and linking, but all associating processes required to build an application. This facility helps to ensure that your production load modules match the source in your production source libraries. For more information, refer to: www.ibm.com/software/awdtools/ispf/features/sclm-ov.html.

- Graphical Data Display Manager (GDDM)

GDDM programs provide 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 the double-byte character set)
- Business image support
- Saving and restoring graphic pictures
- Support for many types of display terminals, printers, and plotters

For more information, refer to: www.ibm.com/software/applications/gddm/.

- Query Management Facility (QMF)

z/OS XL 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. For more information, refer to: www.ibm.com/software/data/qmf/.

- z/OS Java support

The Java language supports the Java Native Interface (JNI) for making calls to and from C/C++. These calls do not use ILC support but rather the Java-defined JNI, which is supported by both compiled and interpreted Java code. Calls to C or C++ do not distinguish between these two.

Additional features of z/OS XL C/C++

Feature	Description
long long Data Type	z/OS XL C/C++ supports long long as a native data type when the compiler option LANGLVL(LONGLONG) is turned on. This option is turned on by default by the compiler option LANGLVL(EXTENDED). As of z/OS V1R7, the XL C compiler supports long long as a native data type (according to the ISO/IEC 9899:1999 standard), when the LANGLVL(STDC99) option or LANGLVL(EXTC99) option is in effect.
Multibyte Character Support	z/OS XL C/C++ supports multibyte characters for those national languages such as Japanese whose characters cannot be represented by a single byte.
Wide Character Support	Multibyte characters can be normalized by z/OS XL 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>mbsrtowcs()</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	z/OS XL C/C++ provides three z/Architecture® 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> . Extended precision floating-point numbers give greater accuracy to mathematical calculations. z/OS XL C/C++ also supports IEEE 754 floating-point representation (base-2 or binary floating-point formats). By default, <code>float</code> , <code>double</code> , and <code>long double</code> values are represented in z/Architecture floating-point formats (base-16 floating-point formats). However, the IEEE 754 floating-point representation is used if you specify the <code>FLOAT(IEEE)</code> compiler option. For details on this support, see the description of the <code>FLOAT</code> option in <i>z/OS XL C/C++ User's Guide</i> . As of z/OS V1R9, XL C/C++ also supports IEEE 754 decimal floating-point representation (base-10 floating-point formats), with the types <code>_Decimal32</code> , <code>_Decimal64</code> , and <code>_Decimal128</code> , if you specify the <code>DFP</code> compiler option. For details on this support, see the description of the <code>DFP</code> option in <i>z/OS XL C/C++ User's Guide</i> .
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	z/OS XL C/C++ provides message text in either American English or Japanese. You can dynamically switch between these two languages.
Coded Character Set (Code Page) Support	The z/OS XL 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.

Feature	Description
Selected Built-in Library Functions	For selected library functions, the compiler generates an instruction sequence directly into the object code during optimization to improve execution performance. String and character functions are examples of these built-in functions. No actual calls to the library are generated when built-in functions are used.
Multi-threading	Threads are efficient in applications that allow them to take advantage of any underlying parallelism available in the host environment. This underlying parallelism in the host can be exploited either by forking a process and creating a new address space, or by using multiple threads within a single process. For more information, refer to "Using Threads in z/OS UNIX Applications" in <i>z/OS XL C/C++ Programming Guide</i> .
Packed Structures and Unions	z/OS XL C provides support for packed structures and unions. Structures and unions may be packed to reduce the storage requirements of a z/OS XL C program or to define structures that are laid out according to COBOL or PL/I structure alignment rules.
Fixed-point (Packed) Decimal Data	z/OS XL 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 COMP-3 or the PL/I data type FIXED DEC, with up to 31 digits of precision.
Long Name Support	For portability, external names can be mixed case and up to 32 K - 1 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 z/OS, z/OS UNIX System Services, and TSO. You can also use the <code>system()</code> function to call EXECs on z/OS and TSO, or shell scripts using z/OS UNIX System Services.
Exploitation of Hardware	<p>Use the ARCHITECTURE compiler option to select the minimum level of machine architecture on which your program will run. Note that certain features provided by the compiler require a minimum architecture level. For more information, refer to the ARCHITECTURE compiler option in <i>z/OS XL C/C++ User's Guide</i>.</p> <p>Use the TUNE compiler option to optimize your application for a specific machine architecture within the constraints imposed by the ARCHITECTURE option. The TUNE level must not be lower than the setting in the ARCHITECTURE option. For more information, refer to the TUNE compiler option in <i>z/OS XL C/C++ User's Guide</i>.</p>
Built-in Functions for Floating-Point and Other Hardware Instructions	Use built-in functions for floating-point and other hardware instructions that are otherwise inaccessible to XL C/C++ programs. For more information, see the built-in functions described in <i>z/OS XL C/C++ Programming Guide</i> .

Chapter 1. Scope and linkage

Scope is the largest region of program text in which a name can potentially be used without qualification to refer to an entity; that is, the largest region in which the name is potentially valid. Broadly speaking, scope is the general context used to differentiate the meanings of entity names. The rules for scope combined with those for name resolution enable the compiler to determine whether a reference to an identifier is legal at a given point in a file.

The scope of a declaration and the visibility of an identifier are related but distinct concepts. Scope is the mechanism by which it is possible to limit the visibility of declarations in a program. The *visibility* of an identifier is the region of program text from which the object associated with the identifier can be legally accessed. Scope can exceed visibility, but visibility cannot exceed scope. Scope exceeds visibility when a duplicate identifier is used in an inner declarative region, thereby hiding the object declared in the outer declarative region. The original identifier cannot be used to access the first object until the scope of the duplicate identifier (the lifetime of the second object) has ended.

Thus, the scope of an identifier is interrelated with the *storage duration* of the identified object, which is the length of time that an object remains in an identified region of storage. The lifetime of the object is influenced by its storage duration, which in turn is affected by the scope of the object identifier.

Linkage refers to the use or availability of a name across multiple translation units or within a single translation unit. The term *translation unit* refers to a source code file plus all the header and other source files that are included after preprocessing with the `#include` directive, minus any source lines skipped because of conditional preprocessing directives. Linkage allows the correct association of each instance of an identifier with one particular object or function.

Scope and linkage are distinguishable in that scope is for the benefit of the compiler, whereas linkage is for the benefit of the linker. During the translation of a source file to object code, the compiler keeps track of the identifiers that have external linkage and eventually stores them in a table within the object file. The linker is thereby able to determine which names have external linkage, but is unaware of those with internal or no linkage.

The distinctions between the different types of scopes are discussed in “Scope.” The different types of linkages are discussed in “Program linkage” on page 7.

Related information

- “Storage class specifiers” on page 43
- Chapter 9, “Namespaces (C++ only),” on page 217

Scope

The *scope* of an identifier is the largest region of the program text in which the identifier can potentially be used to refer to its object. In C++, the object being referred to must be unique. However, the name to access the object, the identifier itself, can be reused. The meaning of the identifier depends upon the context in which the identifier is used. Scope is the general context used to distinguish the meanings of names.

The scope of an identifier is possibly noncontiguous. One of the ways that breakage occurs is when the same name is reused to declare a different entity, thereby creating a contained declarative region (inner) and a containing declarative region (outer). Thus, point of declaration is a factor affecting scope. Exploiting the possibility of a noncontiguous scope is the basis for the technique called *information hiding*.

The concept of scope that exists in C was expanded and refined in C++. The following table shows the kinds of scopes and the minor differences in terminology.

Table 5. Kinds of scope

C	C++
block	local
function	function
Function prototype	Function prototype
file	global namespace
	namespace
	class

In all declarations, the identifier is in scope before the initializer. The following example demonstrates this:

```
int x;
void f() {
    int x = x;
}
```

The x declared in function f() has local scope, not global scope.

Related information

- Chapter 9, “Namespaces (C++ only),” on page 217

Block/local scope

A name has *local scope* or *block 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.

Parameter names for a function have the scope of the outermost block of that function. Also, if the function is declared and not defined, these parameter names have function prototype scope.

When one block is nested inside another, the variables from the outer block are usually visible in the nested block. However, if the declaration of a variable in a nested block has the same name as a variable that is declared in an enclosing block, the declaration in the nested block hides the variable that was declared in the enclosing block. The original declaration is restored when program control returns to the outer block. This is called *block visibility*.

Name resolution in a local scope begins in the immediately enclosing scope in which the name is used and continues outward with each enclosing scope. The order in which scopes are searched during name resolution causes the

phenomenon of information hiding. A declaration in an enclosing scope is hidden by a declaration of the same identifier in a nested scope.

Related information

- “Block statements” on page 163

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.

A label can be used in a `goto` statement before the actual label is seen.

Related information

- “Labeled statements” on page 161

Function prototype scope

In a function declaration (also called a *function prototype*) or in any function declarator—except the declarator of a function definition—parameter names have *function prototype scope*. Function prototype scope terminates at the end of the nearest enclosing function declarator.

Related information

- “Function declarations” on page 183

File/global scope

C only

A name has *file scope* if the identifier’s declaration appears outside of any block. A name with file scope and internal linkage is visible from the point where it is declared to the end of the translation unit.

End of C only

C++ only

Global scope or *global namespace scope* is the outermost namespace scope of a program, in which objects, functions, types and templates can be defined. A name has *global namespace scope* if the identifier’s declaration appears outside of all blocks, namespaces, and classes.

A name with global namespace scope and internal linkage is visible from the point where it is declared to the end of the translation unit.

A name with global (namespace) scope is also accessible for the initialization of global variables. If that name is declared `extern`, it is also visible at link time in all object files being linked.

A user-defined namespace can be nested within the global scope using namespace definitions, and each user-defined namespace is a different scope, distinct from the global scope.

End of C++ only

Related information

- Chapter 9, “Namespaces (C++ only),” on page 217
- “Internal linkage” on page 7
- “The extern storage class specifier” on page 46

Examples 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[])
        {
1         printf("%d\n", i);          /* Prints 1 */
1
1         {
1 2         int i = 2, j = 3;        /* i and j defined at block scope */
1 2                                 /* global definition of i is hidden */
1 2         printf("%d\n%d\n", i, j); /* Prints 2, 3 */
1 2
1 2         {
1 2 3         int i = 0; /* i is redefined in a nested block */
1 2 3                                 /* previous definitions of i are hidden */
1 2 3         printf("%d\n%d\n", i, j); /* Prints 0, 3 */
1 2 3         }
1 2         }
1 2         printf("%d\n", i);      /* Prints 2 */
1 2
1         }
1
1         printf("%d\n", i);      /* Prints 1 */
1
1         return 0;
1
1     }
```

Class scope (C++ only)

A name declared within a member function hides a declaration of the same name whose scope extends to or past the end of the member function's class.

When the scope of a declaration extends to or past the end of a class definition, the regions defined by the member definitions of that class are included in the scope of the class. Members defined lexically outside of the class are also in this scope. In

addition, the scope of the declaration includes any portion of the declarator following the identifier in the member definitions.

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, as long as the derived class does not hide the name
- After the -> (arrow) 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, as long as the derived class does not hide the name
- 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

Related information

- Chapter 11, “Classes (C++ only),” on page 241
- “Scope of class names (C++ only)” on page 245
- “Member scope (C++ only)” on page 255
- “Friend scope (C++ only)” on page 269
- “Access control of base class members (C++ only)” on page 279
- “Scope resolution operator :: (C++ only)” on page 116

Namespaces of identifiers

Namespaces are the various syntactic contexts within which an identifier can be used. Within the same context and the same scope, an identifier must uniquely identify an entity. Note that the term *namespace* as used here applies to C as well as C++ and does not refer to the C++ namespace language feature. The compiler sets up *namespaces* to distinguish among identifiers referring to different kinds of entities. Identical identifiers in different namespaces do not interfere with each other, even if they are in the same scope.

The same identifier can declare different objects as long as each identifier is unique within its namespace. The syntactic context of an identifier within a program lets the compiler resolve its namespace without ambiguity.

Within each of the following four namespaces, 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:
 - C function names (C++ function names can be overloaded)
 - Variable names
 - Names of function parameters
 - Enumeration constants

- typedef names.

You can redefine identifiers in the same namespace but within enclosed program blocks.

Structure tags, structure members, variable names, and statement labels are in four different namespaces. No name conflict occurs among the 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;
}
```

The compiler interprets each occurrence of `student` by its context in the program: when `student` appears after the keyword `struct`, it is a structure tag; when it appears in the block defining the `student` type, it is a structure member variable; when it appears at the end of the structure definition, it declares a structure variable; and when it appears after the `goto` statement, it is a label.

Name hiding (C++ only)

If a class name or enumeration name is in scope and not hidden, it is *visible*. A class name or enumeration name can be hidden by an explicit declaration of that same name — as an object, function, or enumerator — in a nested declarative region or derived class. The class name or enumeration name is hidden wherever the object, function, or enumerator name is visible. This process is referred to as *name hiding*.

In a member function definition, the declaration of a local name hides the declaration of a member of the class with the same name. The declaration of a member in a derived class hides the declaration of a member of a base class of the same name.

Suppose a name `x` is a member of namespace `A`, and suppose that the members of namespace `A` are visible in a namespace `B` because of a `using` declaration. A declaration of an object named `x` in namespace `B` will hide `A::x`. The following example demonstrates this:

```
#include <iostream>
#include <typeinfo>
using namespace std;

namespace A {
    char x;
};

namespace B {
    using namespace A;
    int x;
};
```

```
int main() {
    cout << typeid(B::x).name() << endl;
}
```

The following is the output of the above example:

```
int
```

The declaration of the integer `x` in namespace `B` hides the character `x` introduced by the using declaration.

Related information

- Chapter 11, “Classes (C++ only),” on page 241
- “Member functions (C++ only)” on page 253
- “Member scope (C++ only)” on page 255
- Chapter 9, “Namespaces (C++ only),” on page 217

Program linkage

Linkage determines whether identifiers that have identical names refer to the same object, function, or other entity, even if those identifiers appear in different translation units. The linkage of an identifier depends on how it was declared. There are three types of linkages:

- Internal linkage : identifiers can only be seen within a translation unit.
- External linkage : identifiers can be seen (and referred to) in other translation units.
- No linkage: identifiers can only be seen in the scope in which they are defined.

Linkage does not affect scoping, and normal name lookup considerations apply.

C++ only

You can also have linkage between C++ and non-C++ code fragments, which is called *language linkage*. Language linkage enables the close relationship between C++ and C by allowing C++ code to link with that written in C. All identifiers have a language linkage, which by default is C++. Language linkage must be consistent across translation units, and non-C++ language linkage implies that the identifier has external linkage.

End of C++ only

Related information

- “The static storage class specifier” on page 44
- “The extern storage class specifier” on page 46
- “Function storage class specifiers” on page 188
- “Type qualifiers” on page 67
- “Anonymous unions” on page 61

Internal linkage

The following kinds of identifiers have internal linkage:

- Objects, references, or functions explicitly declared `static`

- Objects or references declared in namespace scope (or global scope in C) with the specifier `const` and neither explicitly declared `extern`, nor previously declared to have external linkage
- Data members of an anonymous union
-  C++ Function templates explicitly declared `static`
-  C++ Identifiers declared in the unnamed namespace

A function declared inside a block will usually have external linkage. An object declared inside a block will usually have external linkage if it is specified `extern`. If a variable that has `static` storage is defined outside a function, the variable has internal linkage and is available from the point where it is defined to the end of the current translation unit.

If the declaration of an identifier has the keyword `extern` and if a previous declaration of the identifier is visible at namespace or global scope, the identifier has the same linkage as the first declaration.

External linkage

C only

In global scope, identifiers for the following kinds of entities declared without the `static` storage class specifier have external linkage:

- An object
- A function

If an identifier in C is declared with the `extern` keyword and if a previous declaration of an object or function with the same identifier is visible, 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. However, a variable or function that has no linkage and was later declared with a linkage specifier will have the linkage that was expressly specified.

End of C only

C++ only

In namespace scope, the identifiers for the following kinds of entities have external linkage:

- A reference or an object that does not have internal linkage
- A function that does not have internal linkage
- A named class or enumeration
- An unnamed class or enumeration defined in a `typedef` declaration
- An enumerator of an enumeration that has external linkage
- A template, unless it is a function template with internal linkage
- A namespace, unless it is declared in an unnamed namespace

If the identifier for a class has external linkage, then, in the implementation of that class, the identifiers for the following will also have external linkage:

- A member function.
- A static data member.
- A class of class scope.

- An enumeration of class scope.

End of C++ only

No linkage

The following kinds of identifiers have no linkage:

- Names that have neither external or internal linkage
- Names declared in local scopes (with exceptions like certain entities declared with the `extern` keyword)
- Identifiers that do not represent an object or a function, including labels, enumerators, typedef names that refer to entities with no linkage, type names, function parameters, and template names

You cannot use a name with no linkage to declare an entity with linkage. For example, you cannot use the name of a structure or enumeration or a typedef name referring to an entity with no linkage to declare an entity with linkage. The following example demonstrates this:

```
int main() {
    struct A { };
    // extern A a1;
    typedef A myA;
    // extern myA a2;
}
```

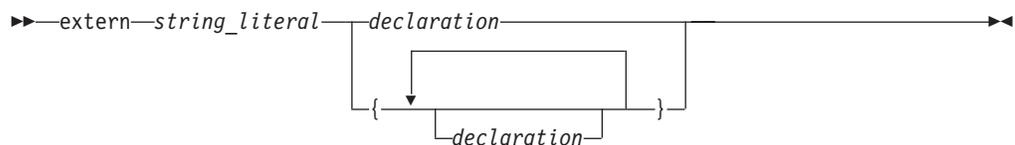
The compiler will not allow the declaration of `a1` with external linkage. Structure `A` has no linkage. The compiler will not allow the declaration of `a2` with external linkage. The typedef name `myA` has no linkage because `A` has no linkage.

Language linkage (C++ only)

Linkage between C++ and non-C++ code fragments is called *language linkage*. All function types, function names, and variable names have a language linkage, which by default is C++.

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

Linkage specification syntax



The *string_literal* is used to specify the linkage associated with a particular function. String literals used in linkage specifications should be considered as case-sensitive. All platforms support the following values for *string_literal*:

- "C++" Unless otherwise specified, objects and functions have this default linkage specification.
- "C" Indicates linkage to a C procedure

Calling shared libraries that were written before C++ needed to be taken into account requires the `#include` directive to be within an `extern "C" {}` declaration.

```
extern "C" {
#include "shared.h"
}
```

The following example shows a C printing function that is called from C++.

```
// in C++ program
extern "C" int displayfoo(const char *);
int main() {
    return displayfoo("hello");
}

/* in C program */
#include <stdio.h>
extern int displayfoo(const char * str) {
    while (*str) {
        putchar(*str);
        putchar(' ');
        ++str;
    }
    putchar('\n');
}
```

CCNX02J

// This example illustrates linkage specifications.

```
extern "C" int printf(const char*,...);

int main(void)
{
    printf("hello\n");
}
```

Here the *string_literal* "C" tells the compiler that the routine `printf(const char*,...)` is a C function.

Note: This example is not guaranteed to work on all platforms. The only safe way to declare a C function in a C++ program is to include the appropriate header. In this example you would substitute the line of code with `extern` with the following line:

```
#include <stdio.h>
```

Related information

- “The extern storage class specifier” on page 46
- “The extern storage class specifier” on page 188
- “#pragma linkage (C only)” on page 419

Name mangling (C++ only)

Name mangling is the encoding of function and variable names into unique names so that linkers can separate common names in the language. Type names may also be mangled. Name mangling is commonly used to facilitate the overloading feature and visibility within different scopes. The compiler generates function names with an encoding of the types of the function arguments when the module is compiled. If a variable is in a namespace, the name of the namespace is mangled into the variable name so that the same variable name can exist in more than one namespace. The C++ compiler also mangles C variable names to identify the namespace in which the C variable resides.

The scheme for producing a mangled name differs with the object model used to compile the source code: the mangled name of an object of a class compiled using

one object model will be different from that of an object of the same class compiled using a different object model. The object model is controlled by compiler option or by pragma.

Name mangling is not desirable when linking C modules with libraries or object files compiled with a C++ compiler. To prevent the C++ compiler from mangling the name of a function, you can apply the extern "C" linkage specifier to the declaration or declarations, as shown in the following example:

```
extern "C" {
    int f1(int);
    int f2(int);
    int f3(int);
};
```

This declaration tells the compiler that references to the functions f1, f2, and f3 should not be mangled.

The extern "C" linkage specifier can also be used to prevent mangling of functions that are defined in C++ so that they can be called from C. For example,

```
extern "C" {
    void p(int){
        /* not mangled */
    }
};
```

In multiple levels of nested extern declarations, the innermost extern specification prevails.

```
extern "C" {
    extern "C++" {
        void func();
    }
}
```

In this example, func has C++ linkage.

Related information

- “The extern storage class specifier” on page 188
- “The __cdecl function specifier (C++ only)” on page 194

Chapter 2. Lexical Elements

A *lexical element* refers to a character or groupings of characters that may legally appear in a source file. This section contains discussions of the basic lexical elements and conventions of the C and C++ programming languages:

- “Tokens”
- “Source program character set” on page 31
- “Comments” on page 36

Tokens

Source code is treated during preprocessing and compilation as a sequence of *tokens*. A token is the smallest independent unit of meaning in a program, as defined by the compiler. There are four different types of tokens:

- Keywords
- Identifiers
- Literals
- Punctuators and operators

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

Keywords

Keywords are identifiers reserved by the language for special use. Although you can use them for preprocessor macro names, it is considered poor programming style. Only the exact spelling of keywords is reserved. For example, `auto` is reserved but `AUTO` is not.

Table 6. C and C++ keywords

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

C only

Standard C at the C99 level also reserves the following keywords:

Table 7. C99 keywords

<code>_Bool</code>	<code>_Imaginary</code> ¹
<code>_Complex</code>	<code>inline</code> ²
	<code>restrict</code> ³

Notes:

1. The keyword `_Imaginary` is reserved for possible future use. For complex number functionality, use `_Complex`; see “Complex literals (C only)” on page 25 for details.
2. The keyword `inline` is only recognized under compilation with **c99** or with the `LANGLVL(STDC99)` or `LANGLVL(EXTC99)` options.
3. The keyword `restrict` is only recognized under compilation with **c99** or with the `LANGLVL(STDC99)` or `LANGLVL(EXTC99)` options.

End of C only

C++ only

The C++ language also reserves the following keywords:

Table 8. C++ keywords

<code>asm</code>	<code>export</code>	<code>private</code>	<code>true</code>
<code>bool</code>	<code>false</code>	<code>protected</code>	<code>try</code>
<code>catch</code>	<code>friend</code>	<code>public</code>	<code>typeid</code>
<code>class</code>	<code>inline</code>	<code>reinterpret_cast</code>	<code>typename</code>
<code>const_cast</code>	<code>mutable</code>	<code>static_cast</code>	<code>using</code>
<code>delete</code>	<code>namespace</code>	<code>template</code>	<code>virtual</code>
<code>dynamic_cast</code>	<code>new</code>	<code>this</code>	<code>wchar_t</code>
<code>explicit</code>	<code>operator</code>	<code>throw</code>	

End of C++ only

Keywords for language extensions

IBM extension

In addition to standard language keywords, z/OS XL C/C++ reserves the following keywords for use in language extensions:

Table 9. Keywords for C and C++ language extensions

<code>__asm</code> (C only)	<code>__Decimal32¹</code>
<code>__asm__</code> (C only)	<code>__Decimal64¹</code>
<code>__attribute__</code>	<code>__Decimal128¹</code>
<code>__attribute</code>	<code>__imag__</code> (C only)
<code>__complex__</code> (C only)	<code>__inline__</code>
<code>__const__</code>	
	<code>__real__</code> (C only)

Notes:

1. These keywords are recognized only when the DFP compiler option is in effect.

C++ only

XL C++ reserves the following keywords as language extensions for compatibility with C99.

Table 10. Keywords for C++ language extensions related to C99

`restrict`

End of C++ only

z/OS only

z/OS XL C/C++ additionally reserves the following for use as extensions:

Table 11. Keywords for C/C++ language extensions on z/OS

C	C++	
		__callback
		__ptr32
__Packed	__cdecl	__ptr64
__packed	__Export	__far ¹

Notes:

1. Recognized only when the METAL compiler option is in effect, which is currently only supported by z/OS XL C.

z/OS XL C/C++ also reserves the following keywords for future use in both C and C++:

Table 12. Reserved keywords for future use

__alignof__
__extension__
__label__
C++
__Pragma

End of z/OS only

More detailed information regarding the compilation contexts in which extension keywords are valid is provided in the sections of this document that describe each keyword.

End of IBM extension

Identifiers

Identifiers provide names for the following language elements:

- Functions
- Objects
- Labels
- Function parameters
- Macros and macro parameters
- Type definitions
- Enumerated types and enumerators
- Structure and union names
- C++ Classes and class members
- C++ Templates
- C++ Template parameters
- C++ Namespaces

An identifier consists of an arbitrary number of letters, digits, or the underscore character in the form:



Related information

- “Identifier expressions (C++ only)” on page 114
- “The Unicode standard” on page 34
- “Keywords” on page 13

Characters in identifiers

The first character in an identifier must be a letter or the `_` (underscore) character; however, beginning identifiers with an underscore is considered poor programming style.

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

Note:  If the names have external linkage, and you do not specify the `LONGNAME` compiler option, names are truncated to eight characters and uppercased in the object file. For example, `STOCKONHOLD` and `stockonhold` will both refer to the same object. For more information on external name mapping, see “External identifiers.”

The universal character names for letters and digits outside of the basic source character set are allowed in C++ and at the C99 language level.  In C++, you must compile with the `LANGLVL(UCS)` option for universal character name support.

External identifiers

z/OS only

By default, external names in C object modules, and external names without C++ linkage in C++ object modules, are formatted as follows:

- All characters are converted to uppercase.
- Names longer than 8 characters are truncated to 8 characters.
- Each underscore character is converted to an at sign (`@`).

For example, if you compile the following C program:

```
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);
}

```

The C compiler displays the following message:

```
ERROR CCN3244 ./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.

To avoid this problem, you can do either of the following:

- Map long external names in the source code to 8 or less characters that you specify, by using the **#pragma map** directive. For example:

```
#pragma map(verylongname,"sname")
```
- Compile with the `LONGNAME` compiler option, and use the binder to produce a program object in a PDSE, or use the prelinker. This allows up to 1024 characters in external names, mixed-case characters, and preserves the underscore character. For more information on the binder, prelinker, and `LONGNAME` compile-time option, see the *z/OS XL C/C++ User's Guide*.

IBM-provided functions have names that begin with `IBM`, `CEE`, and `PLI`. In order to prevent conflicts between runtime functions and user-defined names, the compiler 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 altogether. The compiler of the calling or called language may or may not change these prefixes in the same manner as the `z/OS XL C/C++` compiler does.

To call an external program or access an external variable that begins with `IBM`, `CEE`, or `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")
```

Related information

- “#pragma longname/nolongname” on page 422
- “The #pragma map directive” on page 423

End of z/OS only

Reserved identifiers

Identifiers with two initial underscores or an initial underscore followed by an uppercase letter are reserved globally for use by the compiler.

C Identifiers that begin with a single underscore are reserved as identifiers with file scope in both the ordinary and tag namespaces.

C++ Identifiers that begin with a single underscore are reserved in the global namespace.

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.

The `__func__` predefined identifier

The C99 predefined identifier `__func__` makes a function name available for use within the function. Immediately following the opening brace of each function definition, `__func__` is implicitly declared by the compiler. The resulting behavior is as if the following declaration had been made:

```
static const char __func__[] = "function-name";
```

where *function-name* is the name of the lexically-enclosing function. The function name is not mangled.

C++ only

The function name is qualified with the enclosing class name or function name. For example, if `foo` is a member function of class `X`, the predefined identifier of `foo` is `X::foo`. If `foo` is defined within the body of `main`, the predefined identifier of `foo` is `main::X::foo`.

The names of template functions or member functions reflect the instantiated type. For example, the predefined identifier for the template function `foo` instantiated with `int`, `template<classT> void foo()` is `foo<int>`.

End of C++ only

For debugging purposes, you can explicitly use the `__func__` identifier to return the name of the function in which it appears. For example:

```
#include <stdio.h>

void myfunc(void) {
    printf("%s\n", __func__);
    printf("size of __func__ = %d\n", sizeof(__func__));
}

int main() {
    myfunc();
}
```

The output of the program is:

```
myfunc
size of __func__=7
```

When the `assert` macro is used inside a function definition, the macro adds the name of the enclosing function on the standard error stream.

Related information

- “Function declarations and definitions” on page 183

Literals

The term *literal constant*, or *literal*, refers to a value that occurs in a program and cannot be changed. The C language uses the term *constant* in place of the noun *literal*. The adjective *literal* adds to the concept of a constant the notion that we can speak of it only in terms of its value. A literal constant is nonaddressable, which means that its value is stored somewhere in memory, but we have no means of accessing that address.

Every *literal* has a value and a data type. The value of any literal does not change while the program runs and must be in the range of representable values for its type. The following are the available types of literals:

- Integer literals
- Boolean literals
- Floating-point literals
- z/OS Fixed-point decimal literals
- Character literals
- String literals

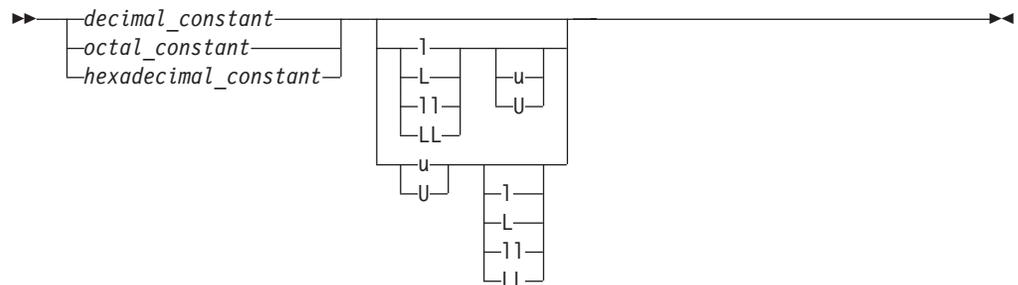
Integer literals

Integer literals are numbers that do not have a decimal point or an exponential part. They can be represented as:

- Decimal integer literals
- Hexadecimal integer literals
- Octal integer literals

An integer literal may have a prefix that specifies its base, or a suffix that specifies its type.

Integer literal syntax



The data type of an integer literal is determined by its form, value, and suffix. The following table lists the integer literals and shows the possible data types. The smallest data type that can represent the constant value is used to store the constant.

Integer literal	Possible data types
unsuffixed decimal	int, long int, C long long int ¹
unsuffixed octal or hexadecimal	int, unsigned int, long int, unsigned long int, C long long int ¹ , C unsigned long long int ¹

Integer literal	Possible data types
decimal, octal, or hexadecimal suffixed by u or U	unsigned int, unsigned long int, <code>> C</code> unsigned long long int ¹
decimal suffixed by l or L	long int, <code>> C</code> long long int ¹
octal or hexadecimal suffixed by l or L	long int, unsigned long int, <code>> C</code> long long int ¹ , <code>> C</code> unsigned long long int ¹
decimal, octal, or hexadecimal suffixed by both u or U, and l or L	unsigned long int, <code>> C</code> unsigned long long int ¹
decimal suffixed by ll or LL	long long int
octal or hexadecimal suffixed by ll or LL	long long int, unsigned long long int
decimal, octal, or hexadecimal suffixed by both u or U, and ll or LL	unsigned long long int
Notes:	
1. Requires compilation with c99 or with <code>LANGVL(STDC99)</code> or <code>LANGVL(EXTC99)</code> .	

z/OS only

In 32-bit mode, an unsuffixed decimal constant of type signed long long is given the type signed long in 64-bit mode.

An unsuffixed integer constant cannot have a value greater than `ULONGLONG_MAX`. An integer constant with a suffix that contains LL cannot have a value greater than `ULONGLONG_MAX`. In these cases, the compiler will issue an *out of range* error message. For information on the `ULONG_MAX` and the `ULONGLONG_MAX` macros, see the *z/OS XL C/C++ Run-Time Library Reference*.

End of z/OS only

Related information

- “Integral types” on page 49
- “Integral conversions” on page 104
- `LANGVL` option in the *z/OS XL C/C++ User’s Guide*

Decimal integer literals: A decimal integer literal contains any of the digits 0 through 9. The first digit cannot be 0. Integer literals beginning with the digit 0 are interpreted as an octal integer literal rather than as a decimal integer literal.

Decimal integer literal syntax



A plus (+) or minus (-) symbol can precede a decimal integer literal. The operator is treated as a unary operator rather than as part of the literal.

The following are examples of decimal literals:

485976
-433132211
+20
5

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

Hexadecimal integer literal syntax



The following are examples of hexadecimal integer literals:

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

Octal integer literals: An *octal integer literal* begins with the digit 0 and contains any of the digits 0 through 7.

Octal integer literal syntax



The following are examples of octal integer literals:

0
0125
034673
03245

Boolean literals

C At the C99 level, C defines `true` and `false` as macros in the header file `stdbool.h`.

C++ There are only two Boolean literals: `true` and `false`.

Related information

- “Boolean types” on page 50
- “Boolean conversions” on page 104

Floating-point literals

Floating-point literals are numbers that have a decimal point or an exponential part. They can be represented as:

- Real literals
 - Binary floating-point literals

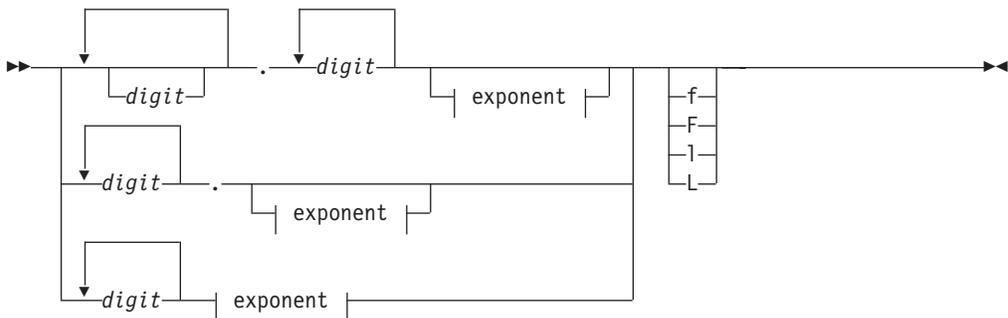
-  Hexadecimal floating-point literals (C only)
-  “Decimal floating-point literals” on page 24
-  Complex literals (C only)

Binary floating-point literals: A real binary floating-point constant consists of the following:

- 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.

Binary floating-point literal syntax



Exponent:



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

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

The following are examples of floating-point literals:

Floating-point constant	Value
5.3876e4	53,876
4e-11	0.000000000004
1e+5	100000
7.321E-3	0.007321
3.2E+4	32000

Floating-point constant	Value
0.5e-6	0.0000005
0.45	0.45
6.e10	60000000000

Related information

- “Floating-point types” on page 51
- “Floating-point conversions” on page 104
- “Unary expressions” on page 118

Hexadecimal floating-point literals (C only): Real hexadecimal floating constants, which are a C99 feature, consist of the following:

- a hexadecimal prefix
- a significant part
- a binary exponent part
- an optional suffix

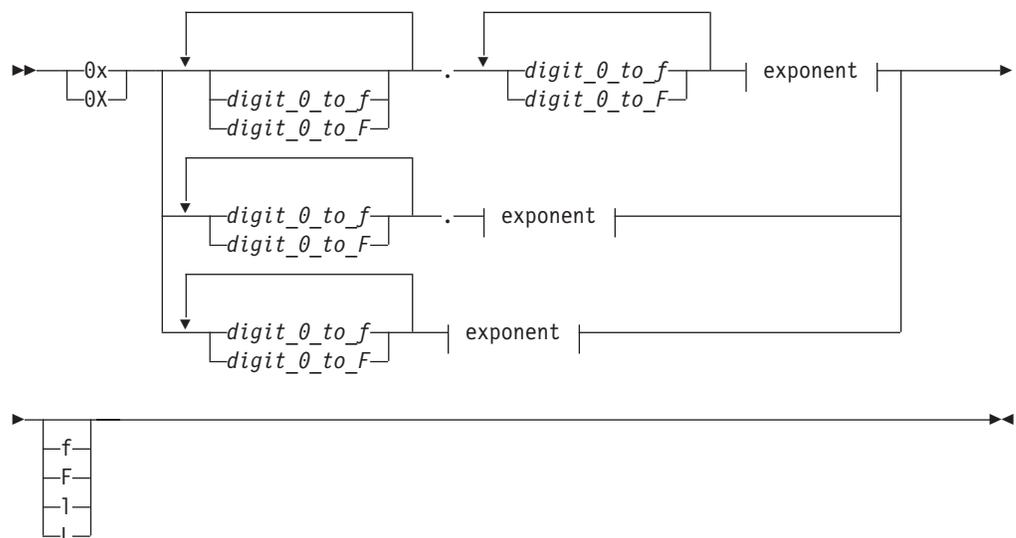
The significant part represents a rational number and is composed of the following:

- a sequence of hexadecimal digits (whole-number part)
- an optional fraction part

The optional fraction part is a period followed by a sequence of hexadecimal digits.

The exponent part indicates the power of 2 to which the significant part is raised, and is an optionally signed decimal integer. The type suffix is optional. The full syntax is as follows:

Hexadecimal floating-point literal syntax



Exponent:



The suffix `f` or `F` indicates a type of `float`, and the suffix `l` or `L` indicates a type of long double. If a suffix is not specified, the floating-point constant has a type `double`. You can omit either the whole-number part or the fraction part, but not both. The binary exponent part is required to avoid the ambiguity of the type suffix `F` being mistaken for a hexadecimal digit.

Decimal floating-point literals:

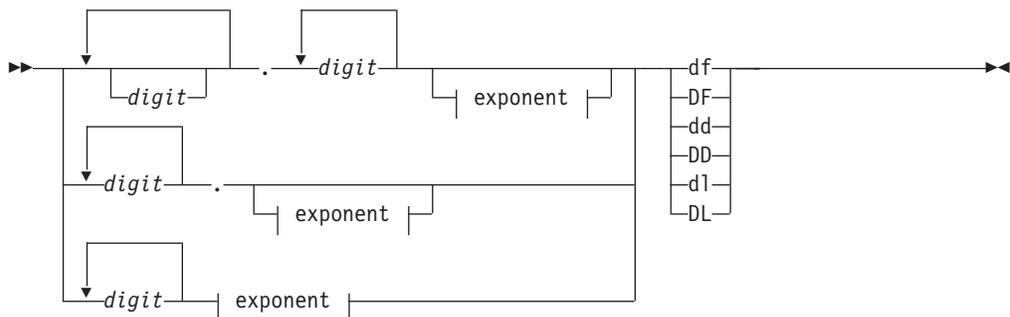
IBM extension

A real decimal floating-point constant consists of the following:

- 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.

Decimal floating-point literal syntax



Exponent:



The suffix `df` or `DF` indicates a type of `_Decimal32`, the suffix `dd` or `DD` indicates a type of `_Decimal64`, and the suffix `d1` or `DL` indicates a type of `_Decimal128`. If a suffix is not specified, the floating-point constant has a type `double`.

You cannot mix cases in the literal suffix.

The following are examples of decimal floating-point literal declarations:

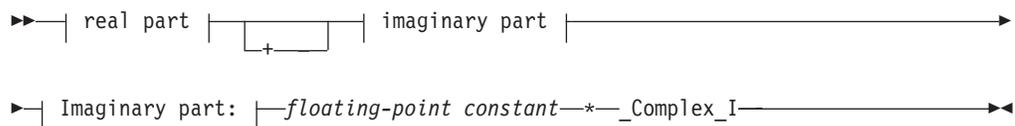
```
_Decimal32 a = 22.2df;  
_Decimal64 b = 33.3dd;
```

Note: Decimal floating-point literal suffixes are recognized only when the **-qdfp** option is enabled.

End of IBM extension

Complex literals (C only): Complex literals, which are a C99 feature, are constructed in two parts: the real part, and the imaginary part.

Complex literal syntax



Real part:

`floating-point constant`

The *floating-point constant* can be specified as a decimal or hexadecimal floating-point constant (including optional suffixes), in any of the formats described in the previous sections.

`_Complex_I` is a macro defined in the `complex.h` header file, representing the imaginary unit *i*, the square root of -1.

For example, the declaration:

```
varComplex = 2.0f + 2.0f * _Complex_I;
```

initializes the complex variable `varComplex` to a value of $2.0 + 2.0i$.

Related information

- “Complex floating-point types (C only)” on page 52

Fixed-point decimal literals

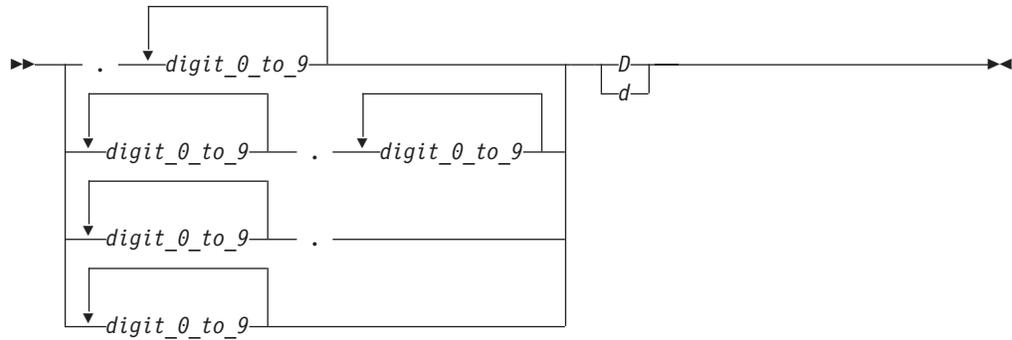
z/OS only

C only

Fixed-point decimal constants are a z/OS XL C extension to Standard 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:



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 *z/OS XL C/C++ Programming Guide*.

Related information

- “Fixed-point decimal types (C only)” on page 53
- “The digitsof and precisionof operators (C only)” on page 126

_____ **End of C only** _____

_____ **End of z/OS only** _____

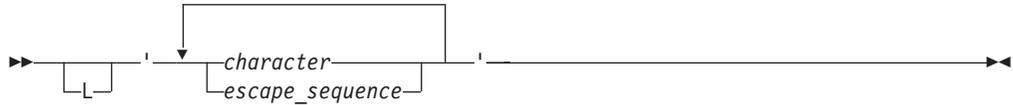
Character literals

A *character literal* contains a sequence of characters or escape sequences enclosed in single quotation mark symbols, for example 'c'. A character literal may be prefixed with the letter L, for example L'c'. A character literal without the L prefix is an *ordinary character literal* or a *narrow character literal*. A character literal with the L prefix is a *wide character literal*. An ordinary character literal that contains more than one character or escape sequence (excluding single quotes (')), backslashes (\) or new-line characters) is a *multicharacter literal*.

C The type of a narrow character literal is `int`. The type of a wide character literal is `wchar_t`. The type of a multicharacter literal is `int`.

► **C++** The type of a character literal that contains only one character is `char`, which is an integral type. The type of a wide character literal is `wchar_t`. The type of a multicharacter literal is `int`.

Character literal syntax



At least one character or escape sequence must appear in the character literal, and the character literal must appear on a single logical source line.

The characters can be from the source program character set. You can represent the double quotation mark symbol by itself, but to represent the single quotation mark symbol, you must use the backslash symbol followed by a single quotation mark symbol (`'` escape sequence). (See “Escape sequences” on page 33 for a list of other characters that are represented by escape characters.)

Outside of the basic source character set, the universal character names for letters and digits are allowed in C++ and at the C99 language level. ► **C++** In C++, you must compile with the `LANGVL(UCS)` option for universal character name support.

The following are examples of character literals:

```
'a'  
'\ '  
L'\0'  
'('
```

Related information

- “Source program character set” on page 31
- “The Unicode standard” on page 34
- “Character types” on page 54

String literals

A *string literal* contains a sequence of characters or escape sequences enclosed in double quotation mark symbols. A string literal with the prefix `L` is a *wide string literal*. A string literal without the prefix `L` is an *ordinary* or *narrow string literal*.

► **C** The type of narrow string literal is array of `char`. The type of a wide character string literal is array of `wchar_t`. Both types have static storage duration.

► **C++** The type of a narrow string literal is array of `const char`. The type of a wide string literal is array of `const wchar_t`. Both types have static storage duration.

A null (`'\0'`) character is appended to each string. For a wide string literal, the value `'\0'` of type `wchar_t` is appended. By convention, programs recognize the end of a string by finding the null character.

String literal syntax



Multiple spaces contained within a string literal are retained.

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 a single quotation mark symbol either by itself or with the escape sequence `\'`. You must use the escape sequence `\"` to represent a double quotation mark.

Outside of the basic source character set, the universal character names for letters and digits are allowed in C++ and at the C99 language level.  In C++, you must compile with the `LANGlvl(UCS)` option for universal character name support.

The following are examples of string literals:

```
char titles[ ] = "Handel's \"Water Music\"";
char *temp_string = "abc" "def" "ghi"; /* *temp_string = "abcdefghi\0" */
wchar_t *wide_string = L"longstring";
```

This example illustrates escape sequences in string literals:

CCNX02K

```
#include <iostream> using namespace std;

int 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 ".
```

To continue a string on the next line, use the line continuation character (`\` symbol) followed by optional whitespace and a new-line character (required). For example:

```
char *mail_addr = "Last Name   First Name   MI   Street Address \
                   893   City   Province   Postal code ";
```

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. */
```

Note: When a string literal appears more than once in the program source, how that string is stored depends on whether strings are read-only or writeable. By default, the compiler considers strings to be read-only. z/OS XL C/C++ may allocate only one location for a read-only string; all occurrences will refer to that one location. However, that area of storage is potentially write-protected. If strings are writeable, then each occurrence of the string

will have a separate, distinct storage location that is always modifiable. You can use the directive or the ROSTRING compiler option to change the default storage for string literals.

Related information

- “Character types” on page 54
- “Source program character set” on page 31
- “The Unicode standard” on page 34
- “#pragma strings” on page 446

String concatenation: Another way to continue a string is to have two or more consecutive strings. Adjacent string literals will be concatenated to produce a single string. For example:

```
"hello " "there"    /* is equivalent to "hello there" */
"hello" "there"    /* is equivalent to "hellothere" */
```

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

C If a wide string literal and a narrow string literal are adjacent, as in the following:

```
"hello " L"there"
```

the result is a wide string literal.

Following any concatenation, '\0' of type char is appended at the end of each string. For a wide string literal, '\0' of type wchar_t is appended. C++ programs find the end of a string by scanning for this value. 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" */
```

Punctuators and operators

A *punctuator* is a token that has syntactic and semantic meaning to the compiler, but the exact significance depends on the context. A punctuator can also be a token that is used in the syntax of the preprocessor.

C99 and C++ define the following tokens as punctuators, operators, or preprocessing tokens:

Table 13. C and C++ punctuators

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

C++ only

In addition to the C99 preprocessing tokens, operators, and punctuators, C++ allows the following tokens as punctuators:

Table 14. C++ punctuators

::	.*	->*	new	delete	
and	and_eq	bitand	bitor	comp	
not	not_eq	or	or_eq	xor	xor_eq

End of C++ only

Related information

- Chapter 6, “Expressions and operators,” on page 111

Alternative tokens

Both C and C++ provide the following alternative representations for some operators and punctuators. The alternative representations are also known as *digraphs*.

Operator or punctuator	Alternative representation
{	<%
}	%>
[<:
]	:>
#	%:
##	%:%:

Note: The recognition of these alternative representations is controlled by the DIGRAPHS option; for more information, see “Digraph characters” on page 35.

In addition to the operators and punctuators listed above, C++ and C at the C99 language level provide the following alternative representations. In C, they are defined as macros in the header file `iso646.h`.

Operator or punctuator	Alternative representation
&&	and
	bitor
	or
^	xor
~	compl
&	bitand
&=	and_eq
=	or_eq
^=	xor_eq
!	not
!=	not_eq

Related information

- “Digraph characters” on page 35

Source program character set

The following lists the basic source character sets that are available at both compile time 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 1 2 3 4 5 6 7 8 9
- The following graphic characters:
! " # % & ' () * + , - . / : ; < = > ? [\] _ { } ~
 - The caret (^) character in ASCII (bitwise exclusive OR symbol) or the equivalent not (~) character in EBCDIC
 - The split vertical bar character in ASCII, which may be represented by the vertical bar (|) character on EBCDIC systems .
- The space character
- The control characters representing new-line, horizontal tab, vertical tab, form feed, end of string (NULL character), alert, backspace, and carriage return.

 Depending on the compiler option, other specialized identifiers, such as the dollar sign (\$) or characters in national character sets, may be allowed to appear in an identifier.

z/OS only

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

If you use the **#pragma filetag** directive to specify the encoding of input files, the compiler converts this encoding to the encoding defined by code page IBM-1047. If you use the LOCALE to specify the encoding for output, the compiler converts the encoding from code page IBM-1047 to the encoding you have specified. These conversions apply to:

- 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

They do not apply to source-code annotation in the pseudo-assembly listings.

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

! # ' [] \ { } ~ ^ |

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 *z/OS XL C/C++ User's Guide*.

End of z/OS only

Related information

- “Characters in identifiers” on page 16
- “#pragma filetag” on page 411

Multibyte characters

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 *multibyte character* sets. A *multibyte character* is a character whose bit representation fits into more than one byte.

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

```
<SO> x y z <SI>
```

Or they can be mixed:

```
<SO> x <SI> y z x <SO> y <SI> z
```

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

Multibyte characters can appear in any of the following contexts:

- String literals and character constants. To declare a multibyte literal, use a wide-character representation, prefixed by L. For example:

```
wchar_t *a = L"wide_char_string";  
wchar_t b = L'wide_char';
```

Strings containing multibyte characters are treated essentially the same way as strings without multibyte characters. Generally, wide characters are permitted anywhere multibyte characters are, but they are incompatible with multibyte characters in the same string because their bit patterns differ. Wherever permitted, you can mix single-byte and multibyte characters in the same string.

- Preprocessor directives. The following preprocessor directives permit multibyte-character constants and string literals:

- #define
- #pragma comment
- #include

A file name specified in an #include directive can contain multibyte characters. For example:

```
#include <multibyte_char/mydir/mysource/multibyte_char.h>  
#include "multibyte_char.h"
```

- Macro definitions. Because string literals and character constants can be part of #define statements, multibyte characters are also permitted in both object-like and function-like macro definitions.
- The # and ## operators
- Program comments

The following are restrictions on the use of multibyte characters:

- Multibyte characters are not permitted in identifiers.
- Hexadecimal values for multibyte characters must be in the range of the code page being used.

on a system using the ASCII character codes, the value of the escape sequence `\x56` is the letter V. On a system using 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. An error message is issued if an escape sequence is not recognized.

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 Unicode standard

The *Unicode Standard* is the specification of an encoding scheme for written characters and text. It is a universal standard that enables consistent encoding of multilingual text and allows text data to be interchanged internationally without conflict. The ISO standards for C and C++ refer to *Information technology — Programming Languages — Universal Multiple-Octet Coded Character Set (UCS), ISO/IEC 10646:2003*. (The term *octet* is used by ISO to refer to a byte.) The ISO/IEC 10646 standard is more restrictive than the Unicode Standard in the number of encoding forms: a character set that conforms to ISO/IEC 10646 is also conformant to the Unicode Standard.

The Unicode Standard specifies a unique numeric value and name for each character and defines three encoding forms for the bit representation of the numeric value. The name/value pair creates an identity for a character. The hexadecimal value representing a character is called a *code point*. The specification also describes overall character properties, such as case, directionality, alphabetic properties, and other semantic information for each character. Modeled on ASCII, the Unicode Standard treats alphabetic characters, ideographic characters, and symbols, and allows implementation-defined character codes in reserved code point ranges. According to the Unicode Standard, the encoding scheme of the standard is therefore sufficiently flexible to handle all known character encoding requirements, including coverage of all the world's historical scripts.

C99 and C++ allow the universal character name construct defined in ISO/IEC 10646 to represent characters outside the basic source character set. Both languages permit universal character names in identifiers, character constants, and string literals.  In C++, you must compile with the `LANGVLV(UCS)` option for universal character name support.

The following table shows the generic universal character name construct and how it corresponds to the ISO/IEC 10646 short name.

Universal character name	ISO/IEC 10646 short name
<i>where N is a hexadecimal digit</i>	
<code>\UNNNNNNNN</code>	NNNNNNNN
<code>\uNNNN</code>	0000NNNN

C99 and C++ disallow the hexadecimal values representing characters in the basic character set (base source code set) and the code points reserved by ISO/IEC 10646 for control characters.

The following characters are also disallowed:

- Any character whose short identifier is less than 00A0. The exceptions are 0024 (\$), 0040 (@), or 0060 (').
- Any character whose short identifier is in the code point range D800 through DFFF inclusive.

Digraph characters

You can represent unavailable characters in a source program by using a combination of two keystrokes that are called a *digraph character*. The preprocessor reads digraphs as tokens during the preprocessor phase. To enable processing of digraphs, use the DIGRAPH compiler option (which is enabled by default).

The digraph characters are:

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

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

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

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

Trigraph sequences

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

Trigraph	Single character	Description
??=	#	pound 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. For example,

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

Represents:

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

z/OS only

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. If you do not specify the LOCALE option, the preprocessor uses code page IBM-1047.

The z/OS XL C/C++ compiler will compile source files that were edited using different encoding of character sets. However, they might not compile cleanly. z/OS XL 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 *z/OS XL C/C++ Programming Guide*.

End of z/OS only

Comments

A *comment* is text replaced during preprocessing by a single space character; the compiler therefore ignores all comments.

There are two kinds of comments:

- The /* (slash, asterisk) characters, followed by any sequence of characters (including new lines), followed by the */ characters. This kind of comment is commonly called a *C-style comment*.
- The // (two slashes) characters followed by any sequence of characters. A new line not immediately preceded by a backslash terminates this form of comment. This kind of comment is commonly called a *single-line comment* or a *C++ comment*. A C++ comment can span more than one physical source line if it is joined into one logical source line with line-continuation (\) characters. The backslash character can also be represented by a trigraph.  *C only* To enable C++ comments in C, you must compile with **c99**, or with the SSCOMM or LANGLVL(STDC99) or LANGLVL(EXTC99) options.

You can put comments anywhere the language allows white space. You cannot nest C-style comments inside other C-style comments. Each comment ends at the first occurrence of */.

You can also include multibyte characters.

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

In the following program, the second printf() is a comment:

```

#include <stdio.h>

int main(void)
{
    printf("This program has a comment.\n");
    /* printf("This is a comment line and will not print.\n"); */
    return 0;
}

```

Because the second `printf()` 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, `printf()` in the following program is not a comment.

```

#include <stdio.h>

int main(void)
{
    printf("This program does not have \
/* NOT A COMMENT */ a comment.\n");
    return 0;
}

```

The output of the program is:

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

In the following example, the comments are highlighted:

/* A program with nested comments. */

```

#include <stdio.h>

int main(void)
{
    test_function();
    return 0;
}

int test_function(void)
{
    int number;
    char letter;
    /*
    number = 55;
    letter = 'A';
    /* number = 44; */
    */
    return 999;
}

```

In `test_function`, the compiler reads the first `/*` through to the first `*/`. The second `*/` causes an error. To avoid commenting over comments already in the source code, you should use conditional compilation preprocessor directives to cause the compiler to bypass sections of a program. For example, instead of commenting out the above statements, change the source code in the following way:

```

/* A program with conditional compilation to avoid nested comments. */

#define TEST_FUNCTION 0
#include <stdio.h>

int main(void)

```

```

{
    test_function();
    return 0;
}

int test_function(void)
{
    int number;
    char letter;
    #if TEST_FUNCTION
        number = 55;
        letter = 'A';
        /*number = 44;*/
    #endif /*TEST_FUNCTION */
}

```

You can nest single line comments within C-style comments. For example, the following program will not output anything:

```

#include <stdio.h>

int main(void)
{
    /*
    printf("This line will not print.\n");
    // This is a single line comment
    // This is another single line comment
    printf("This line will also not print.\n");
    */
    return 0;
}

```

Note: You can also use the **#pragma comment** directive to place comments into an object module.

Related information

- “#pragma comment” on page 400
- “Multibyte characters” on page 32

Chapter 3. Data objects and declarations

This section discusses the various elements that constitute a declaration of a data object, and includes the following topics:

- “Overview of data objects and declarations”
- “Storage class specifiers” on page 43
- “Type specifiers” on page 49
- “User-defined types” on page 55
- “Type qualifiers” on page 67
- “Type attributes” on page 74

Topics are sequenced to loosely follow the order in which elements appear in a declaration. The discussion of the additional elements of data declarations is also continued in Chapter 4, “Declarators,” on page 77.

Overview of data objects and declarations

The following sections introduce some fundamental concepts regarding data objects and data declarations that will be used throughout this reference.

Overview of data objects

A data *object* is a region of storage that contains a value or group of values. Each value can be accessed using its identifier or a more complex expression that refers to the object. In addition, each object has a unique *data type*. The data type of an object determines the storage allocation for that object and the interpretation of the values during subsequent access. It is also used in any type checking operations. Both the identifier and data type of an object are established in the object *declaration*.

 An instance of a class type is commonly called a *class object*. The individual class members are also called objects. The set of all member objects comprises a class object.

Data types are often grouped into type categories that overlap, such as:

Fundamental types versus derived types

Fundamental data types are also known as “basic”, “fundamental” or “built-in” to the language. These include integers, floating-point numbers, and characters. *Derived* types, also known as “compound” types in Standard C++, are created from the set of basic types, and include arrays, pointers, structures, unions, and enumerations. All C++ classes are considered compound types.

Built-in types versus user-defined types

Built-in data types include all of the fundamental types, plus types that refer to the addresses of basic types, such as arrays and pointers. *User-defined* types are created by the user from the set of basic types, in typedef, structure, union, and enumeration definitions. C++ classes are considered user-defined types.

Scalar types versus aggregate types

Scalar types represent a single data value, while *aggregate* types represent multiple values, of the same type or of different types. Scalars include the

arithmetic types and pointers. Aggregate types include arrays and structures. C++ classes are considered aggregate types.

The following matrix lists the supported data types and their classification into fundamental, derived, scalar, and aggregate types.

Table 15. C/C++ data types

Data object	Basic	Compound	Built-in	User-defined	Scalar	Aggregate
integer types	+		+		+	
floating-point types ¹	+		+		+	
character types			+		+	
Booleans	+		+		+	
void type	+ ²		+		+	
pointers		+	+		+	
arrays		+	+			+
structures		+		+		+
unions		+		+		
enumerations		+		+	see note ³	
 classes		+		+		+
typedef types		+		+		

Notes:

-  Although complex floating-point types are represented internally as an array of two elements, they behave in the same way as real floating-pointing types in terms of alignment and arithmetic operations, and can therefore be considered scalar types.
- The void type is really an incomplete type, as discussed in “Incomplete types.” Nevertheless, Standard C++ defines it as a fundamental type.
-  The C standard does not classify enumerations as either scalar or aggregate.  Standard C++ classifies enumerations as scalars.

Related information

- Chapter 11, “Classes (C++ only),” on page 241

Incomplete types

The following are incomplete types:

- The void type
- Arrays of unknown size
- Arrays of elements that are of incomplete type
- Structure, union, or enumerations that have no definition
-  Pointers to class types that are declared but not defined
-  Classes that are declared but not defined

C only

However, if an array size is specified by [*], indicating a variable length array, the size is considered as having been specified, and the array type is then considered

a complete type. For more information, see “Variable length arrays (C only)” on page 86.

End of C only

The following examples illustrate incomplete types:

```
void *incomplete_ptr;
struct dimension linear; /* no previous definition of dimension */
```

Related information

- “The void type” on page 54
- “Incomplete class declarations (C++ only)” on page 246

Compatible and composite types

C only

In C, compatible types are defined as:

- two types that can be used together without modification (as in an assignment expression)
- two types that can be substituted one for the other without modification

When two compatible types are combined, the result is a *composite type*. Determining the resultant composite type for two compatible types is similar to following the usual binary conversions of integral types when they are combined with some arithmetic operators.

Obviously, two types that are identical are compatible; their composite type is the same type. Less obvious are the rules governing type compatibility of non-identical types, user-defined types, type-qualified types, and so on. “Type specifiers” on page 49 discusses compatibility for basic and user-defined types in C.

End of C only

C++ only

A separate notion of type compatibility as distinct from being of the same type does not exist in C++. Generally speaking, type checking in C++ is stricter than in C: identical types are required in situations where C would only require compatible types.

End of C++ only

Related information

- “Compatibility of arrays (C only)” on page 87
- “Compatibility of pointers (C only)” on page 83
- “Compatible functions” on page 186

Overview of data declarations and definitions

A *declaration* establishes the names and characteristics of data objects used in a program. A *definition* allocates storage for data objects, and associates an identifier with that object. When you declare or define a *type*, no storage is allocated.

The following table shows examples of declarations and definitions. The identifiers declared in the first column do not allocate storage; they refer to a corresponding definition. The identifiers declared in the second column allocate storage; they are both declarations and definitions.

Declarations	Declarations and definitions
<code>extern double pi;</code>	<code>double pi = 3.14159265;</code>
<code>struct payroll;</code>	<code>struct payroll { char *name; float salary; } employee;</code>

Note:  The C99 standard no longer requires that all declarations appear at the beginning of a function before the first statement. As in C++, you can mix declarations with other statements in your code.

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

- Scope, which describes the region of program text in which an identifier can be used to access its object
- Visibility, which describes the region of program text from which legal access can be made to the identifier's object
- Duration, which defines the period during which the identifiers have real, physical objects allocated in memory
- Linkage, which describes the correct association of an identifier to one particular object
- Type, which determines how much memory is allocated to an object and how the bit patterns found in the storage allocation of that object should be interpreted by the program

The elements of a declaration for a data object are as follows:

- Storage class specifiers, which specify storage duration and linkage
- Type specifiers, which specify data types
- Type qualifiers, which specify the mutability of data values
- Declarators, which introduce and include identifiers
- Initializers, which initialize storage with initial values

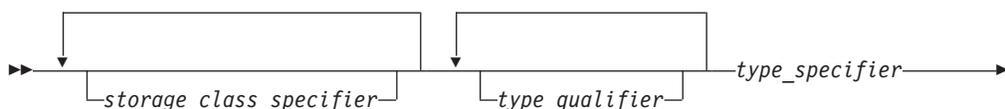
IBM extension

In addition, for compatibility with GCC, z/OS XL C/C++ allows you to use *attributes* to modify the properties of data objects. They are described in "Variable attributes" on page 100.

End of IBM extension

All declarations have the form:

Data declaration syntax





Related information

- “Function declarations and definitions” on page 183

Tentative definitions

C only

A *tentative definition* is any external data declaration that has no storage class specifier and no initializer. A tentative definition becomes a full definition if the end of the translation unit is reached and no definition has appeared with an initializer for the identifier. In this situation, the compiler reserves uninitialized space for the object defined.

The following statements show normal definitions and tentative definitions.

```
int i1 = 10;          /* definition, external linkage */
static int i2 = 20; /* definition, internal linkage */
extern int i3 = 30; /* definition, external linkage */
int i4;              /* tentative definition, external linkage */
static int i5;      /* tentative definition, internal linkage */

int i1;              /* valid tentative definition */
int i2;              /* not legal, linkage disagreement with previous */
int i3;              /* valid tentative definition */
int i4;              /* valid tentative definition */
int i5;              /* not legal, linkage disagreement with previous */
```

End of C only

C++ only

C++ does not support the concept of a tentative definition: an external data declaration without a storage class specifier is always a definition.

End of C++ only

Storage class specifiers

A storage class specifier is used to refine the declaration of a variable, a function, and parameters. Storage classes determine whether:

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

For a variable, its default storage duration, scope, and linkage depend on where it is declared: whether inside or outside a block statement or the body of a function. When these defaults are not satisfactory, you can use a storage class specifier to explicitly set its storage class. The storage class specifiers in C and C++ are:

- auto
- static
- extern
-  mutable
- register

Related information

- “Function storage class specifiers” on page 188
- “Initializers” on page 88

The auto storage class specifier

The auto storage class specifier lets you explicitly declare a variable with *automatic storage*. The auto storage class is the default for variables declared inside a block. A variable *x* that has automatic storage is deleted when the block in which *x* was declared exits.

You can only apply the auto storage class specifier to names of variables declared in a block or to names of function parameters. However, these names by default have automatic storage. Therefore the storage class specifier `auto` is usually redundant in a data declaration.

Storage duration of automatic variables

Objects with the auto storage class specifier have automatic storage duration. Each time a block is entered, storage for auto objects defined in that block is made available. When the block is exited, the objects are no longer available for use. An object declared with no linkage specification and without the `static` storage class specifier has automatic storage duration.

If an auto object is defined within a function that is recursively invoked, memory is allocated for the object at each invocation of the block.

Linkage of automatic variables

An auto variable has block scope and no linkage.

Related information

- “Initialization and storage classes” on page 89
- “Block statements” on page 163
- “The goto statement” on page 177

The static storage class specifier

Objects declared with the `static` storage class specifier have *static storage duration*, which means that memory for these objects is allocated when the program begins running and is freed when the program terminates. Static storage duration for a variable is different from file or global scope: a variable can have static duration but local scope.

 The keyword `static` is the major mechanism in C to enforce information hiding.

▶ **C++** C++ enforces information hiding through the namespace language feature and the access control of classes. The use of the keyword `static` to limit the scope of external variables is deprecated for declaring objects in namespace scope.

The `static` storage class specifier can be applied to the following declarations:

- Data objects
- ▶ **C++** Class members
- Anonymous unions

You cannot use the `static` storage class specifier with the following:

- Type declarations
- Function parameters

C only

At the C99 language level, the `static` keyword can be used in the declaration of an array parameter to a function. The `static` keyword indicates that the argument passed into the function is a pointer to an array of at least the specified size. In this way, the compiler is informed that the pointer argument is never null. See “Static array indices in function parameter declarations (C only)” on page 202 for more information.

End of C only

Related information

- “The `static` storage class specifier” on page 188
- “Static members (C++ only)” on page 260

Linkage of static variables

A declaration of an object that contains the `static` storage class specifier and has file scope gives the identifier internal linkage. Each instance of the particular identifier therefore represents the same object within one file only. For example, if a static variable `x` has been declared in function `f`, when the program exits the scope of `f`, `x` is not destroyed:

```
#include <stdio.h>

int f(void) {
    static int x = 0;
    x++;
    return x;
}

int main(void) {
    int j;
    for (j = 0; j < 5; j++) {
        printf("Value of f(): %d\n", f());
    }
    return 0;
}
```

The following is the output of the above example:

```
Value of f(): 1
Value of f(): 2
Value of f(): 3
Value of f(): 4
Value of f(): 5
```

Because `x` is a static variable, it is not reinitialized to 0 on successive calls to `f`.

Related information

- “Initialization and storage classes” on page 89
- “Internal linkage” on page 7
- Chapter 9, “Namespaces (C++ only),” on page 217

The extern storage class specifier

The extern storage class specifier lets you declare objects that several source files can use. An extern declaration makes the described variable usable by the succeeding part of the current source file. This declaration does not replace the definition. The declaration is used to describe the variable that is externally defined.

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 a declaration for an identifier already exists at file scope, any extern declaration of the same identifier found within a block refers to that same object. If no other declaration for the identifier exists at file scope, the identifier has external linkage.

C++ C++ restricts the use of the extern storage class specifier to the names of objects or functions. Using the extern specifier with type declarations is illegal. An extern declaration cannot appear in class scope.

Storage duration of external variables

All extern objects have static storage duration. Memory is allocated for extern objects before the `main` function begins running, and is freed when the program terminates. The scope of the variable depends on the location of the declaration in the program text. If the declaration appears within a block, the variable has block scope; otherwise, it has file scope.

Linkage of external variables

C Like the scope, the linkage of a variable declared extern depends on the placement of the declaration in the program text. If the variable declaration appears outside of any function definition and has been declared `static` earlier in the file, the variable has internal linkage; otherwise, it has external linkage in most cases. All object declarations that occur outside a function and that do not contain a storage class specifier declare identifiers with external linkage.

C++ For objects in the unnamed namespace, the linkage may be external, but the name is unique, and so from the perspective of other translation units, the name effectively has internal linkage.

Related information

- “External linkage” on page 8
- “Initialization and storage classes” on page 89
- “The extern storage class specifier” on page 188
- “Class scope (C++ only)” on page 4
- Chapter 9, “Namespaces (C++ only),” on page 217

The mutable storage class specifier (C++ only)

The mutable storage class specifier is used only on a class data member to make it modifiable even though the member is part of an object declared as `const`. You cannot use the mutable specifier with names declared as `static` or `const`, or reference members.

In the following example:

```
class A
{
public:
    A() : x(4), y(5) { };
    mutable int x;
    int y;
};

int main()
{
    const A var2;
    var2.x = 345;
    // var2.y = 2345;
}
```

the compiler would not allow the assignment `var2.y = 2345` because `var2` has been declared as `const`. The compiler will allow the assignment `var2.x = 345` because `A::x` has been declared as `mutable`.

Related information

- “Type qualifiers” on page 67
- “References (C++ only)” on page 87

The register storage class specifier

The register storage class specifier indicates to the compiler that the object should be stored in a machine register. The register storage class specifier is typically specified for heavily used variables, such as a loop control variable, in the hopes of enhancing performance by minimizing access time. However, the compiler is not required to honor this request. Because of the limited size and number of registers available on most systems, few variables can actually be put in registers. If the compiler does not allocate a machine register for a register object, the object is treated as having the storage class specifier `auto`.

An object having the register storage class specifier must be defined within a block or declared as a parameter to a function.

The following restrictions apply to the register storage class specifier:

-  You cannot use pointers to reference objects that have the register storage class specifier.
-  You cannot use the register storage class specifier when declaring objects in global scope.
-  A register does not have an address. Therefore, you cannot apply the address operator (`&`) to a register variable.
-  You cannot use the register storage class specifier when declaring objects in namespace scope.

C++ only

Unlike C, C++ lets you take the address of an object with the register storage class. For example:

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

End of C++ only

Storage duration of register variables

Objects with the register storage class specifier have automatic storage duration. Each time a block is entered, storage for register objects defined in that block 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 is recursively invoked, memory is allocated for the variable at each invocation of the block.

Linkage of register variables

Since a register object is treated as the equivalent to an object of the auto storage class, it has no linkage.

Related information

- “Initialization and storage classes” on page 89
- “Block/local scope” on page 2
- “References (C++ only)” on page 87

Variables in specified registers (C only)

IBM extension

When the GENASM compiler option is in effect, you can specify that a particular hardware register is dedicated to a global variable by using an asm *register variable* declaration. Global register variables reserve registers throughout the program; stores into the reserved register are never deleted. The register variable must be of type pointer.

Register variable declaration syntax

```
▶▶ register variable_declaration [__asm__] ("register_specifier") ▶▶
```

The *register_specifier* is a string representing a hardware register. The register name is CPU-specific. The following are valid register names:

r0 to r15 or R0 to R15

General purpose registers

The following are the rules of use for register variables:

- Registers can only be reserved for variables of pointer type.
- A global register variable cannot be initialized.
- The register dedicated for a global register variable should not be a volatile register, or the value stored into the global variable might not be preserved across a function call.

- More than one register variable can reserve the same register; however, the two variables become aliases of each other, and this is diagnosed with a warning.
- The same global register variable cannot reserve more than one register.

Related information

- “Inline assembly statements (C only)” on page 178

End of IBM extension

Type specifiers

Type specifiers indicate the type of the object being declared. The following are the available kinds of type specifiers:

- Fundamental or built-in types:
 - Arithmetic types
 - Integral types
 - Boolean types
 - Floating-point types
 -  Fixed-point decimal types (C only)
 - Character types
 - The void type
- User-defined types.

Related information

- “Function return type specifiers” on page 198
- “C/C++ data mapping” on page 480

Integral types

Integer types fall into the following categories:

- Signed integer types:
 - signed char
 - short int
 - int
 - long int
 - long long int
- Unsigned integer types:
 - unsigned char
 - unsigned short int
 - unsigned int
 - unsigned long int
 - unsigned long long int

C++ only

z/OS XL C++ supports the long long data type for language levels other than ANSI by default. You can also control the support for long long using the LONGLONG suboption of LANGLVL. For example, specifying LANGLVL(ANSI, LONGLONG) would add the long long data type to the ISO language level. Refer to the z/OS XL

C/C++ User's Guide for information on using the LANGLVL option.

End of C++ only

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

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

C++ only

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.

End of C++ only

Related information

- “Integer literals” on page 19
- “Integral conversions” on page 104
- “Arithmetic conversions and promotions” on page 103
- Chapter 10, “Overloading (C++ only),” on page 225
- “Integers” on page 478

Boolean types

A Boolean variable can be used to hold the integer values 0 or 1, or the literals `true` or `false`, which are implicitly promoted to the integers 0 and 1 whenever an arithmetic value is necessary. The Boolean type is unsigned and has the lowest ranking in its category of standard unsigned integer types; it may not be further qualified by the specifiers `signed`, `unsigned`, `short`, or `long`. In simple assignments, if the left operand is a Boolean type, then the right operand must be either an arithmetic type or a pointer.

C only

Boolean types are a C99 feature. To declare a Boolean variable, use the `_Bool` type specifier.

End of C only

C++ only

To declare a Boolean variable in C++, use the `bool` type specifier. The result of the equality, relational, and logical operators is of type `bool`: either of the Boolean constants `true` or `false`.

End of C++ only

You can use Boolean types make *Boolean logic tests*. A Boolean logic test is used to express the results of a logical operation. For example:

```
_Bool f(int a, int b)
{
    return a==b;
}
```

If a and b have the same value, f returns true. If not, f returns false.

Related information

- “Boolean literals” on page 21
- “Boolean conversions” on page 104

Floating-point types

Floating-point type specifiers fall into the following categories:

- “Real floating-point types”
- Complex floating-point types (C only)

Real floating-point types

Generic, or binary, floating-point types consist of the following:

- float
- double
- long double

IBM extension

Decimal floating-point types consist of the following:

- `_Decimal32`
- `_Decimal64`
- `_Decimal128`

Note: In order for the `_Decimal32`, `_Decimal64`, and `_Decimal128` keywords to be recognized, you must compile with the DFP option. See DFP in the *z/OS XL C/C++ User's Guide* for details.

End of IBM extension

The magnitude ranges of the real floating-point types are given in the following table.

Table 16. Magnitude ranges of real floating-point types

Type	Range
FLOAT(HEX):	
float	5.397605^{-79} - 7.237005^{75}
double	5.397605^{-79} - 7.237006^{75}
long double	5.397605^{-79} - 7.237006^{75}
FLOAT(IEEE):	
float	1.175494^{-38} - 3.402823^{38}
double	2.225074^{-308} - 1.797693^{308}
long double	3.362103^{-4932} - 1.189731^{4932}

The representation and alignment requirements of a complex type are the same as an array type containing two elements of the corresponding real type. The real part is equal to the first element; the imaginary part is equal to the second element.

The equality and inequality operators have the same behavior as for real types. None of the relational operators may have a complex type as an operand.

IBM As an extension to C99, complex numbers may also be operands to the unary operators ++ (increment), -- (decrement), and ~ (bitwise negation).

Related information

- “Complex literals (C only)” on page 25
- “Arithmetic conversions and promotions” on page 103
- “The `__real__` and `__imag__` operators (C only)” on page 126

Fixed-point decimal types (C only)

z/OS only

Fixed-point decimal types are classified as arithmetic types. To declare fixed-point decimal variables and initialize them with fixed-point decimal constants, you use the type specifier `decimal`. For this type specifier, `decimal` is a macro that is defined in the `decimal.h` header file. Remember to include `decimal.h` if you use fixed-point decimals in your program.

Fixed-point decimal syntax

► `decimal` `(` `—` *significant_digits* `—` `,` `—` *precision_digits* `—` `)` ◄

The *significant_digits* is a positive integral constant expression. The second argument, *precision_digits* is optional. If you leave it out, the default value is 0. The type specifiers `decimal(n,0)` and `decimal(n)` are type-compatible.

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

1. *precision_digits* <= *significant_digits*
2. 1 <= *significant_digits* <= DEC_DIG
3. 0 <= *precision_digits* <= DEC_PRECISION

The `decimal.h` file defines DEC_DIG (the maximum number of digits) and DEC_PRECISION (the maximum precision). 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).

Related information

- “Fixed-point decimal literals” on page 25
- “The digitsof and precisionof operators (C only)” on page 126

End of z/OS only

Character types

Character types fall into the following categories:

- Narrow character types:
 - char
 - signed char
 - unsigned char
- Wide character type wchar_t

The char specifier is an integral type. The wchar_t type specifier is an integral type that has enough storage to represent a wide character literal. (A wide character literal is a character literal that is prefixed with the letter L, for example L'x')

C A char is a distinct type from signed char and unsigned char, and the three types are not compatible.

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

If it does not matter if 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 to declare numeric variables that occupy a single byte. When a char (signed or unsigned) is widened to an int, its value is preserved.

By default, char behaves like an unsigned char. To change this default, you can use the CHARS option or the **#pragma chars** directive. See “#pragma chars” on page 398 and CHARS in the *z/OS XL C/C++ User's Guide*, SC09-4767 for more information.

Related information

- “Character literals” on page 26
- “String literals” on page 27
- “Arithmetic conversions and promotions” on page 103

The void type

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

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.

Related information

- “Pointers” on page 80
- “Comma operator ,” on page 139
- “Conditional expressions” on page 141
- “Function declarations and definitions” on page 183

Compatibility of arithmetic types (C only)

Two arithmetic types are compatible only if they are the same type.

The presence of type specifiers in various combinations for arithmetic types may or may not indicate different types. For example, the type `signed int` is the same as `int`, except when used as the types of bit fields; but `char`, `signed char`, and `unsigned char` are different types.

The presence of a type qualifier changes the type. That is, `const int` is not the same type as `int`, and therefore the two types are not compatible.

User-defined types

The following are user-defined types:

- Structures and unions
- Enumerations
- typedef definitions
-  Classes
-  Elaborated type specifiers

C++ classes are discussed in Chapter 11, “Classes (C++ only),” on page 241. Elaborated type specifiers are discussed in “Scope of class names (C++ only)” on page 245.

Structures and unions

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*.

A *union* is an object similar to a structure except that all of its members start at the same location in memory. A union variable can represent the value of only one of its members at a time.

 In C++, structures and unions are the same as classes except that their members and inheritance are public by default.

You can declare a structure or union type separately from the definition of variables of that type, as described in “Structure and union type definition” on page 56 and “Structure and union variable declarations” on page 59; or you can define a structure or union data type and all variables that have that type in one statement, as described in “Structure and union type and variable definitions in a single statement” on page 60.

Structures and unions are subject to alignment considerations. For information on changing alignment and packing structures, see “The `_Packed` qualifier (C only)” on page 99 and “`#pragma pack`” on page 435.

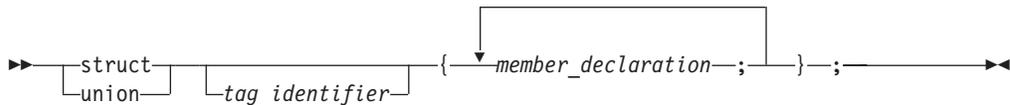
Related information

- “Classes and structures (C++ only)” on page 244

Structure and union type definition

A structure or union *type definition* contains the `struct` or `union` keyword followed by an optional identifier (the structure tag) and a brace-enclosed list of members.

Structure or union type definition syntax



The `tag_identifier` gives a name to the type. If you do not provide a tag name, you must put all variable definitions that refer to the type within the declaration of the type, as described in “Structure and union type and variable definitions in a single statement” on page 60. Similarly, you cannot use a type qualifier with a structure or union definition; type qualifiers placed in front of the `struct` or `union` keyword can only apply to variables that are declared within the type definition.

Member declarations

The list of members provides a structure or union data type with a description of the values that can be stored in the structure or union. The definition of a member has the form of a standard variable declaration. The names of member variables must be distinct within a single structure or union, but the same member name may be used in another structure or union type that is defined within the same scope, and may even be the same as a variable, function, or type name.

A structure or union member may be of any type except:

- any variably modified type
- any `void` type
- **C** a function
- any incomplete type

Because incomplete types are not allowed as members, a structure or union type may not contain an instance of itself as a member, but is allowed to contain a pointer to an instance of itself. **C** As a special case, the last element of a structure with more than one member may have an incomplete array type, which is called a *flexible array member*, as described in “Flexible array members (C only)” on page 57.

C++ A union member cannot be a class object that has a constructor, destructor, or overloaded copy assignment operator, nor can it be of reference type. A union member cannot be declared with the keyword `static`.

A member that does not represent a bit field can be qualified with either of the type qualifiers `volatile` or `const`. The result is an lvalue.

Structure members are assigned to memory addresses in increasing order, with the first component starting at the beginning address of the structure name itself. To allow proper alignment of components, padding bytes may appear between any consecutive members in the structure layout.

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 member having the most stringent requirements). All of a union's components are effectively overlaid in memory: each member of a union is allocated storage starting at the beginning of the union, and only one member can occupy the storage at a time.

Flexible array members (C only): A *flexible array member* is permitted as the last element of a structure even though it has incomplete type, provided that the structure has more than one named member. A flexible array member is a C99 feature and can be used to access a variable-length object. It is declared with an empty index, as follows:

```
array_identifier[ ];
```

For example, `b` is a flexible array member of `Foo`.

```
struct Foo{
    int a;
    int b[];
};
```

Since a flexible array member has incomplete type, you cannot apply the `sizeof` operator to a flexible array.

Any structure containing a flexible array member cannot be a member of another structure or array.

Related information

- “Variable length arrays (C only)” on page 86

Bit field members: Both C and C++ allow integer members to be stored into memory spaces smaller than the compiler would ordinarily allow. These space-saving structure members are called *bit fields*, and their width in bits can be explicitly declared. Bit fields are used in programs that must force a data structure to correspond to a fixed hardware representation and are unlikely to be portable.

Bit field member declaration syntax

```
▶▶—type_specifier—┐—constant_expression—;▶▶  
└—declarator—┘
```

The *constant_expression* is a constant integer expression that indicates the field width in bits. A bit field declaration may not use either of the type qualifiers `const` or `volatile`.

C only

In C99, the allowable data types for a bit field include qualified and unqualified `_Bool`, `signed int`, and `unsigned int`. The default integer type for a bit field is

unsigned.

End of C only

C++ only

A bit field can be any integral type or enumeration type.

End of C++ only

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;
};
```

When you assign a value that is out of range to a bit field, the low-order bit pattern is preserved and the appropriate bits are assigned.

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

If a series of bit fields does not add up to the size of an int, padding can take place. The amount of padding is determined by the alignment characteristics of the members of the structure. In some instances, bit fields can cross word boundaries.

Bit fields with a length of 0 must be unnamed. Unnamed bit fields cannot be referenced or initialized.

A zero-width bit field causes the next field to be aligned on the next container boundary where the container is the same size as the underlying type of the bit field. The padding to the next container boundary only takes place if the zero-width bit field has the same underlying type as the preceding bit field member. If the types are different, the zero-width bit field has no effect. A `_Packed` structure, which has a bit field of length 0, causes the next field to align on the next byte boundary.

The following example demonstrates padding, and is valid for all implementations. Suppose that an int occupies 4 bytes. The example declares the identifier kitchen to be of type struct on_off:

```
struct on_off {
    unsigned light : 1;
    unsigned toaster : 1;
    int count; /* 4 bytes */
    unsigned ac : 4;
    unsigned : 4;
    unsigned clock : 1;
    unsigned : 0;
    unsigned flag : 1;
} kitchen ;
```

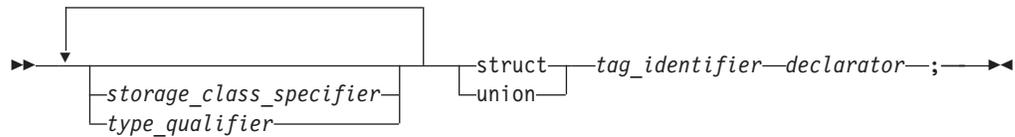
The structure kitchen contains eight members totalling 16 bytes. The following table describes the storage that each member occupies:

Member name	Storage occupied
light	1 bit
toaster	1 bit
(padding — 30 bits)	To the next int boundary
count	The size of an int (4 bytes)
ac	4 bits
(unnamed field)	4 bits
clock	1 bit
(padding — 23 bits)	To the next int boundary (unnamed field)
flag	1 bit
(padding — 31 bits)	To the next int boundary

Structure and union variable declarations

A structure or union *declaration* has the same form as a definition except the declaration does not have a brace-enclosed list of members. You must declare the structure or union data type before you can define a variable having that type.

Structure or union variable declaration syntax



The *tag_identifier* indicates the previously-defined data type of the structure or union.

► C++ The keyword `struct` is optional in structure variable declarations.

You can declare structures or unions having any storage class. The storage class specifier and any type qualifiers for the variable must appear at the beginning of the statement. Structures or unions declared with the `register` storage class specifier are treated as automatic variables.

The following example defines structure type `address`:

```

struct address {
    int street_no;
    char *street_name;
    char *city;
    char *prov;
    char *postal_code;
};
  
```

The following examples declare two structure variables of type `address`:

```

struct address perm_address;
struct address temp_address;
  
```

Related information

- “The aligned variable attribute” on page 101
- “Initialization of structures and unions” on page 92

- “Compatibility of structures, unions, and enumerations (C only)” on page 64
- “Dot operator .” on page 117
- “Arrow operator ->” on page 117

Structure and union type and variable definitions in a single statement

You can define a structure (or union) type and a structure (or union) variable in one statement, by putting a declarator and an optional initializer after the variable definition. The following example defines a union data type (not named) and a union variable (named `length`):

```
union {
    float meters;
    double centimeters;
    long inches;
} length;
```

Note that because this example does not name the data type, `length` is the only variable that can have this data type. Putting an identifier after `struct` or `union` keyword provides a name for the data type and lets you declare additional variables of this data type later in the program.

To specify a storage class specifier for the variable or variables, 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;
```

In this case, both `perm_address` and `temp_address` are assigned static storage.

Type qualifiers can be applied to the variable or variables declared in a type definition. Both of the following examples are valid:

```
volatile struct class1 {
    char descript[20];
    long code;
    short complete;
} file1, file2;

struct class1 {
    char descript[20];
    long code;
    short complete;
} volatile file1, file2;
```

In both cases, the structures `file1` and `file2` are qualified as `volatile`.

Related information

- “Initialization of structures and unions” on page 92
- “Storage class specifiers” on page 43
- “Type qualifiers” on page 67

Access to structure and union members

Once structure or union variables have been declared, members are referenced by specifying the variable name with the dot operator (`.`) or a pointer with the arrow operator (`->`) and the member name. For example, both of the following:

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

assign the string "Ontario" to the pointer prov that is in the structure perm_address.

All references to members of structures and unions, including bit fields, must be fully qualified. In the previous example, the fourth field cannot be referenced by prov alone, but only by perm_address.prov.

Related information

- “Dot operator .” on page 117
- “Arrow operator ->” on page 117

Anonymous unions

An *anonymous union* is a union without a name. It cannot be followed by a declarator. An anonymous union is not a type; it defines an unnamed object.

z/OS XL C supports anonymous unions only when you use the LANGLVL(COMMONC) or LANGLVL(EXTENDED) compiler option.

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 containing 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.

```
void f()  
{  
    union { int i; char* cptr ; };  
    /* . . . */  
    i = 5;  
    cptr = "string_in_union"; // overrides the value 5  
}
```

 An anonymous union cannot have protected or private members, and it cannot have member functions. A global or namespace anonymous union must be declared with the keyword static.

Related information

- “The static storage class specifier” on page 44
- “Member functions (C++ only)” on page 253

Enumerations

An *enumeration* is a data type consisting of a set of named values that represent integral constants, known as *enumeration constants*. An enumeration also referred to as an *enumerated type* because you must list (enumerate) each of the values in creating a name for each of them. In addition to providing a way of defining and grouping sets of integral constants, enumerations are useful for variables that have a small number of possible values.

You can declare an enumeration type separately from the definition of variables of that type, as described in “Enumeration type definition” on page 62 and

“Enumeration variable declarations” on page 63; or you can define an enumeration data type and all variables that have that type in one statement, as described in “Enumeration type and variable definitions in a single statement” on page 64.

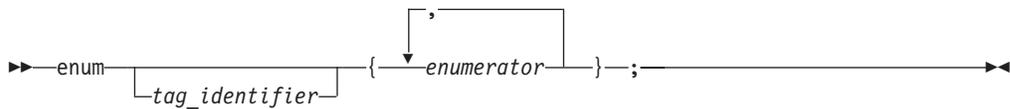
Related information

- “Arithmetic conversions and promotions” on page 103
- “#pragma enum” on page 407

Enumeration type definition

An enumeration type definition contains the `enum` keyword followed by an optional identifier (the enumeration tag) and a brace-enclosed list of enumerators. A comma separates each enumerator in the enumerator list.  C99 allows a trailing comma between the last enumerator and the closing brace.

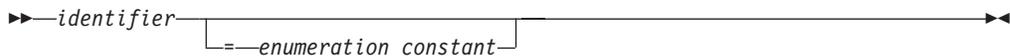
Enumeration definition syntax



The `tag_identifier` gives a name to the enumeration type. If you do not provide a tag name, you must put all variable definitions that refer to the enumeration type within the declaration of the type, as described in “Enumeration type and variable definitions in a single statement” on page 64. Similarly, you cannot use a type qualifier with an enumeration definition; type qualifiers placed in front of the `enum` keyword can only apply to variables that are declared within the type definition.

Enumeration members: The list of enumeration members, or *enumerators*, provides the data type with a set of values.

Enumeration member declaration syntax



 In C, an *enumeration constant* is of type `int`. If a constant expression is used as an initializer, the value of the expression cannot exceed the range of `int` (that is, `INT_MIN` to `INT_MAX` as defined in the header `limits.h`). Otherwise, the condition is tolerated, a diagnostic message is issued, but the value of the enumeration constant is undefined.

 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. You can use an enumeration constant anywhere an integer constant is allowed, or anywhere a value of the enumeration type is allowed.

The value of an enumeration 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 enumeration constant. The enumeration constant represents the value of the constant expression.
2. If no explicit value is assigned, the leftmost enumeration constant in the list receives the value zero (0).

- Enumeration constants with no explicitly assigned values receive the integer value that is one greater than the value represented by the previous enumeration constant.

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      */
```

Each enumeration constant must be unique within the scope in which the enumeration is defined. In the following example, the second declarations of `average` and `poor` cause compiler errors:

```
func()
{
    enum score { poor, average, good };
    enum rating { below, average, above };
    int poor;
}
```

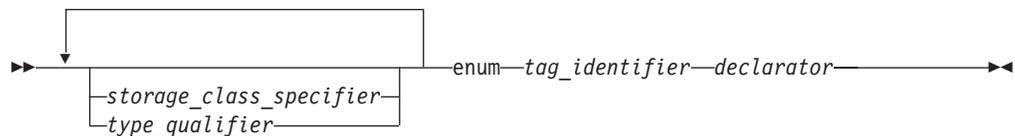
Related information

- “Integral types” on page 49

Enumeration variable declarations

You must declare the enumeration data type before you can define a variable having that type.

Enumeration variable declaration syntax



The *tag_identifier* indicates the previously-defined data type of the enumeration.



The keyword `enum` is optional in enumeration variable declarations.

Related information

- “Initialization of enumerations” on page 94
- “Compatibility of structures, unions, and enumerations (C only)” on page 64

Enumeration type and variable definitions in a single statement

You can define a type and a variable in one statement by using a declarator and an optional initializer after the variable 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++ only

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

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

End of C++ only

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`. `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 having that data type lets you leave the data type unnamed. For example:

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

defines the variable `weekday`, which can be assigned any of the specified enumeration constants. However, you can not declare any additional enumeration variables using this set of enumeration constants.

Compatibility of structures, unions, and enumerations (C only)

Within a single source file, each structure or union definition creates a new type that is neither the same as nor compatible with any other structure or union type.

However, a type specifier that is a reference to a previously defined structure or union type is the same type. The tag associates the reference with the definition, and effectively acts as the type name. To illustrate this, only the types of structures `j` and `k` are compatible in this example:

```
struct { int a; int b; } h;  
struct { int a; int b; } i;  
struct S { int a; int b; } j;  
struct S k;
```

Compatible structures may be assigned to each other.

Structures or unions with identical members but different tags are not compatible and cannot be assigned to each other. Structures and unions with identical members but using different alignments are not also compatible and cannot be assigned to each other.

z/OS only

You cannot perform comparisons between packed and nonpacked structures or

unions of the same type. You cannot assign packed and nonpacked structures or unions to each other, regardless of their type. You cannot pass a packed structure or union argument to a function that expects a nonpacked structure or union of the same type and vice versa.

End of z/OS only

Since the compiler treats enumeration variables and constants as integer types, you can freely mix the values of different enumerated types, regardless of type compatibility. Compatibility between an enumerated type and the integer type that represents it is controlled by compiler options and related pragmas. For a discussion of the `ENUMSIZE` compiler option, see the *z/OS XL C/C++ User's Guide*. For a discussion of the `#pragma enum` directive, see “`#pragma enum`” on page 407.

Related information

- “Arithmetic conversions and promotions” on page 103
- Chapter 11, “Classes (C++ only),” on page 241
- “Structure and union type definition” on page 56
- “Incomplete types” on page 40
- “The `_Packed` qualifier (C only)” on page 99
- “`#pragma pack`” on page 435

Compatibility across separate source files

When the definitions for two structures, unions, or enumerations are defined in separate source files, each file can theoretically contain a different definition for an object of that type with the same name. The two declarations must be compatible, or the run time behavior of the program is undefined. Therefore, the compatibility rules are more restrictive and specific than those for compatibility within the same source file. For structure, union, and enumeration types defined in separately compiled files, the composite type is the type in the current source file.

The requirements for compatibility between two structure, union, or enumerated types declared in separate source files are as follows:

- If one is declared with a tag, the other must also be declared with the same tag.
- If both are completed types, their members must correspond exactly in number, be declared with compatible types, and have matching names.

For enumerations, corresponding members must also have the same values.

For structures and unions, the following additional requirements must be met for type compatibility:

- Corresponding members must be declared in the same order (applies to structures only).
- Corresponding bit fields must have the same widths.

typedef definitions

A typedef declaration lets you define your own identifiers that can be used 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, but synonyms for the data types or combinations of data types they represent.

The namespace for a typedef name is the same as other identifiers. The exception to this rule is if the typedef name specifies a variably modified type. In this case, it has block scope.

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.

Related information

- “Type names” on page 79
- “Type specifiers” on page 49
- “Structures and unions” on page 55
- Chapter 11, “Classes (C++ only),” on page 241

Examples of typedef definitions

The following statements define LENGTH as a synonym for int and then use this typedef to declare length, width, and height as integer variables:

```
typedef int LENGTH;  
LENGTH length, width, height;
```

The following declarations are equivalent to the above declaration:

```
int length, width, height;
```

Similarly, typedef can be used to define a structure, union, or C++ class. For example:

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

The structure WEIGHT can then be used in the following declarations:

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

In the following example, the type of yds is “pointer to function with no parameter specified, returning int”.

```
typedef int SCROLL();  
extern SCROLL *yds;
```

In the following typedefs, the token struct is part of the type name: the type of ex1 is struct a; the type of ex2 is struct b.

```
typedef struct a { char x; } ex1, *ptr1;  
typedef struct b { char x; } ex2, *ptr2;
```

Type ex1 is compatible with the type struct a and the type of the object pointed to by ptr1. Type ex1 is not compatible with char, ex2, or struct b.

C++ only

In C++, a typedef name must be different from any class type name declared within the same scope. If the typedef name is the same as a class type name, it can only be so if that typedef is a synonym of the class name. This condition is not the same as in C. The following can be found in standard C headers:

```
typedef class C { /* data and behavior */ } C;
```

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, so `Trees()` cannot be a constructor for that class.

End of C++ only

Type qualifiers

A type qualifier is used to refine the declaration of a variable, a function, and parameters, by specifying whether:

- The value of an object can be changed
- The value of an object must always be read from memory rather than from a register
- More than one pointer can access a modifiable memory address

z/OS XL C/C++ recognizes the following type qualifiers:

- `const`
- `restrict`
- `volatile`

z/OS only

z/OS XL C/C++ includes the following additional type qualifiers to meet the special needs of the z/OS environment:

- `__callback`
- `__far`
- `__ptr32`

End of z/OS only

Standard C++ refers to the type qualifiers `const` and `volatile` as *cv-qualifiers*. In both languages, the cv-qualifiers are only meaningful in expressions that are lvalues.

When the `const` and `volatile` keywords are used with pointers, the placement of the qualifier is critical in determining whether it is the pointer itself that is to be qualified, or the object to which the pointer points. For a pointer that you want to qualify as `volatile` or `const`, 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 and qualifier can be in any order, provided that the qualifier does not follow the `*` operator. 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 */

```

The following examples contrast the semantics of these declarations:

Declaration	Description
<code>const int * ptr1;</code>	Defines a pointer to a constant integer: the value pointed to cannot be changed.
<code>int * const ptr2;</code>	Defines a constant pointer to an integer: the integer can be changed, but ptr2 cannot point to anything else.
<code>const int * const ptr3;</code>	Defines a constant pointer to a constant integer: neither the value pointed to nor the pointer itself can be changed.

You can put more than one qualifier on a declaration: the compiler ignores duplicate type qualifiers.

A type qualifier cannot apply to user-defined types, but only to objects created from a user-defined type. Therefore, the following declaration is illegal:

```

volatile struct omega {
    int limit;
    char code;
}

```

However, if a variable or variables are declared within the same definition of the type, a type qualifier can be applied to the variable or variables by placing at the beginning of the statement or before the variable declarator or declarators.

Therefore:

```

volatile struct omega {
    int limit;
    char code;
} group;

```

provides the same storage as:

```

struct omega {
    int limit;
    char code;
} volatile group;

```

In both examples, the `volatile` qualifier only applies to the structure variable `group`.

When type qualifiers are applied to a structure, class, or union, or class variable, they also apply to the members of the structure, class or union.

Related information

- “Pointers” on page 80
- “Constant and volatile member functions (C++ only)” on page 254

The `__callback` type qualifier

z/OS only

The keyword `__callback` is a qualifier that can be applied only to a function pointer type. The qualifier instructs the compiler to generate extra code in the call sites to assist the call, and thus allows the function pointer to point to either XPLINK or non-XPLINK functions. Under normal circumstances, a non-XPLINK function pointer is incompatible with XPLINK compilation units.

The keyword can appear in the declarator part of a function pointer declaration, wherever a cv-qualifier can appear. For example,

```
int (*__callback foo)(int);
```

declares `foo` to be a function pointer that might point to non-XPLINK functions. `foo` will then have fewer restrictions on what it can reference and can thus be used with XPLINK compilation units.

XPLINK and non-XPLINK compilation units cannot be statically bound; the two linkages can be mixed only across DLL boundaries. Moreover, a function pointer that points to a non-XPLINK function cannot be used in XPLINK DLLs unless the pointer is passed across the boundary explicitly as a function argument. The `__callback` qualifier relaxes the latter restriction, at the expense of extra code sequences in the call site.

Semantically, the `__callback` keyword is a language extension that has a single effect: to instruct the compiler to generate assistance code. It does not take part in type definition. The keyword also has no effect on the following:

- Type (such as in overload resolution).
- Name mangling.
- Allocation of the pointer object in memory.

It is the responsibility of the programmer to make sure that the function pointer is appropriately `__callback`-qualified for all call sites that require it.

End of z/OS only

The `const` type qualifier

The `const` qualifier explicitly declares a data object as something that cannot be changed. Its value is set at initialization. You cannot use `const` data objects in expressions requiring a modifiable lvalue. For example, a `const` data object cannot appear on the lefthand side of an assignment statement.

C only

A `const` object cannot be used in constant expressions. A global `const` object without an explicit storage class is considered `extern` by default.

C++ only

In C++, all `const` declarations must have initializers, except those referencing externally defined constants. A `const` object can appear in a constant expression if it is an integer and it is initialized to a constant. The following example demonstrates this:

```
const int k = 10;
int ary[k];    /* allowed in C++, not legal in C */
```

In C++ a global `const` object without an explicit storage class is considered `static` by default, with internal linkage.

```
const int k = 12; /* Different meanings in C and C++ */

static const int k2 = 120; /* Same meaning in C and C++ */
extern const int k3 = 121; /* Same meaning in C and C++ */
```

Because its linkage is assumed to be internal, a `const` object can be more easily defined in header files in C++ than in C.

End of C++ only

An item can be both `const` and `volatile`. In this case the item cannot be legitimately modified by its own program but can be modified by some asynchronous process.

Related information

- “The `#define` directive” on page 373
- “The `this` pointer (C++ only)” on page 257

The `__far` type qualifier (C only)

z/OS only

When the `METAL` option is in effect, you can use the `__far` keyword to qualify a pointer type so that it can access additional data spaces in access-register (AR) mode. The upper half of the pointer contains the access-list-entry token (ALET), which identifies the secondary virtual address space you want to access. The lower half the pointer is the offset within the secondary virtual address space. The size of a `__far`-qualified pointer is increased to 8 bytes in 31-bit mode and 16 bytes in 64-bit mode. In 31-bit mode, the upper 4 bytes contain the ALET, and the lower 4 bytes is the address within the data space. In 64-bit mode, bytes 0-3 are unused, bytes 4-7 are the ALET, and bytes 8-15 are the address within the data space.

The `__far` keyword must appear in the declarator part of a pointer declaration, wherever a `cv`-qualifier can be used. For example,

```
int * __far p;
```

declares `p` to be a `__far` pointer to `int`.

`__far` pointers can appear in global scope and function scope, in simple assignment and in implicit assignment via function parameter passing. However, if they are used inside a function in operations that access the data space, such as

dereferencing, the function must be in AR mode (that is, with the ARMODE compiler option in effect, or qualified with the `armode` function attribute).

A normal pointer can be converted to a `__far` pointer explicitly through typecasting or implicitly through assignment. The ALET of the `__far` pointer is set to zero. A `__far` pointer can be explicitly converted to a normal pointer through typecasting; the normal pointer keeps the offset of the `__far` pointer and the ALET is lost. A `__far` pointer cannot be implicitly converted to a normal pointer.

Pointer arithmetic is supported for `__far` pointers, with the ALET part being ignored. If the two ALETs are different, the results may have no meaning.

Two `__far` pointers can be compared for equality and inequality using the `==` and `!=` operators. The whole pointer is compared. To compare for equality of the offset only, use the built-in function to extract the offset and then compare. To compare for equality of the ALET only, use the built-in function to extract the ALET and then compare. For more information on the set of built-in functions that operate on `__far` pointers, see *z/OS XL C/C++ Programming Guide*.

Two `__far` pointers can be compared using the `>`, `<`, `>=`, and `<=` relational operators. The ALET parts of the pointers are ignored in this operation. There is no ordering between two `__far` pointers if their ALETs are different, and between a NULL pointer and any `__far` pointers. The result is meaningless if they are compared using relational operators.

When a `__far` pointer and a normal pointer are involved in an operation, the normal pointer is implicitly converted to `__far` before the operation. There is unspecified behavior if the ALETs are different. For example:

```
int * __far p;
int * __far q;
ptrdiff_t chunk;
...

if (p == q) {
    p = p + 1024;
}

if (p < q) {
    chunk = q - p;
}
else {
    chunk = p - q;
}
```

The result of the `&` (address) operator is a normal pointer, except for the following cases:

- If the operand of `&` is the result of an indirection operator (`*`), the type of `&` is the same as the operand of the indirection operator.
- If the operand of `&` is the result of the arrow operator (`->`, structure member access), the type of `&` is the same as the left operand of the arrow operator.

For example:

```
int * __far p;
int * __far q;
...

q = &*(p+2); // result of & is a __far pointer; the ALET is the same as p.

struct S {
```

```

|         int b;
|     } * __far r;
|     ...
|     q = & r->b; // result of & is a __far pointer; the ALET is the same as r.

```

Related information

- “The `armode | noarmode` function attribute (C only)” on page 204
- The METAL and ARMODE compiler options in the *z/OS XL C/C++ User’s Guide*.

End of z/OS only

The `__ptr32` type qualifier

z/OS only

The keyword `__ptr32` is a qualifier that can be applied to a pointer type to constrain its size to 32 bits. This language extension is provided to facilitate porting structures with pointer members from 31- to 64-bit mode. The qualifier is accepted and ignored in 31-bit mode.

The size of a pointer type doubles to 64 bits in 64-bit mode. Doubling the size of a pointer changes the layout of a structure that contains pointer members. If the object referenced by a pointer member resides within a 31-bit addressing space, constraining the pointer to 32 bits can reduce some of the unexpected effects of moving to 64-bit mode.

The `__ptr32` keyword can appear in the declarator part of a pointer declaration, wherever a cv-qualifier can be used. For example,

```
int * __ptr32 p;
```

declares `p` to be a 32-bit pointer to `int`.

```
int * __ptr32 *q;
```

declares `q` to be a 64-bit pointer to a 32-bit pointer to `int`.

```
int * __ptr32 const r;
```

declares `r` to be a const 32-bit pointer.

Pointers with external linkage must be `__ptr32`-qualified consistently across all compilation units. If a pointer is declared 31-bit in one compilation unit and 64-bit in another, the behavior is undefined.

Assignment of 32-bit and 64-bit pointers to each other is permitted. The compiler generates an implicit conversion or truncates without emitting a diagnostic.

Note: The terms *31-bit mode* and *32-bit mode* are used interchangeably when there is no ambiguity. The term *32-bit mode* is commonly used in the industry to refer to a class of machines, to which z/OS in 31-bit mode belongs. Strictly speaking, *31-bit mode* refers to the addressing mode of the architecture, and *32 bits* refers to the size of the pointer type. In z/OS 31-bit addressing mode, the size of a pointer is four bytes. However, the high-order bit is reserved for system use, and is not used to form the address. The addressing range in this mode is therefore 2 gigabytes. In 64-bit mode, the size of a pointer is eight bytes, and all 64 bits participate in addressing. When a `__ptr32` pointer is dereferenced, a 64-bit address is formed by filling

the 33 missing high-order bits with zeros. The program using that address should make sure it is valid within the address space of the application.

End of z/OS only

The restrict type qualifier

A pointer is the address of a location in memory. More than one pointer can access the same chunk of memory and modify it during the course of a program. The `restrict` (or `__restrict` or `__restrict__`)¹ type qualifier is an indication to the compiler that, if the memory addressed by the `restrict`-qualified pointer is modified, no other pointer will access that same memory. The compiler may choose to optimize code involving `restrict`-qualified pointers in a way that might otherwise result in incorrect behavior. It is the responsibility of the programmer to ensure that `restrict`-qualified pointers are used as they were intended to be used. Otherwise, undefined behavior may result.

If a particular chunk of memory is not modified, it can be aliased through more than one restricted pointer. The following example shows restricted pointers as parameters of `foo()`, and how an unmodified object can be aliased through two restricted pointers.

```
void foo(int n, int * restrict a, int * restrict b, int * restrict c)
{
    int i;
    for (i = 0; i < n; i++)
        a[i] = b[i] + c[i];
}
```

Assignments between restricted pointers are limited, and no distinction is made between a function call and an equivalent nested block.

```
{
    int * restrict x;
    int * restrict y;
    x = y; // undefined
    {
        int * restrict x1 = x; // okay
        int * restrict y1 = y; // okay
        x = y1; // undefined
    }
}
```

In nested blocks containing restricted pointers, only assignments of restricted pointers from outer to inner blocks are allowed. The exception is when the block in which the restricted pointer is declared finishes execution. At that point in the program, the value of the restricted pointer can be carried out of the block in which it was declared.

Notes:

1. The `restrict` qualifier is represented by the following keywords (all have the same semantics):
 - The `restrict` keyword is recognized in C, under compilation with **c99** or the `LANGLVL(STDC99)` or `LANGLVL(EXTC99)` options, and in C++ under the `LANGLVL(EXTENDED)` or `KEYWORD (RESTRICT)` options.  The `__restrict` and `__restrict__` keywords are recognized in both C, at all language levels, and C++, at `LANGLVL(EXTENDED)`.

The volatile type qualifier

The `volatile` qualifier declares a data object that can have its value changed in ways outside the control or detection of the compiler (such as a variable updated by the system clock or by another program). This prevents the compiler from optimizing code referring to the object by storing the object's value in a register and re-reading it from there, rather than from memory, where it may have changed.

Accessing any lvalue expression that is `volatile`-qualified produces a side effect. A side effect means that the state of the execution environment changes.

References to an object of type "pointer to `volatile`" may be optimized, but no optimization can occur to references to the object to which it points. An explicit cast must be used to assign a value of type "pointer to `volatile T`" to an object of type "pointer to `T`". The following shows valid uses of `volatile` objects.

```
volatile int * pvol;  
int *ptr;  
pvol = ptr;           /* Legal */  
ptr = (int *)pvol;   /* Explicit cast required */
```

C A signal-handling function may store a value in a variable of type `sig_atomic_t`, provided that the variable is declared `volatile`. This is an exception to the rule that a signal-handling function may not access variables with static storage duration.

An item can be both `const` and `volatile`. In this case the item cannot be legitimately modified by its own program but can be modified by some asynchronous process.

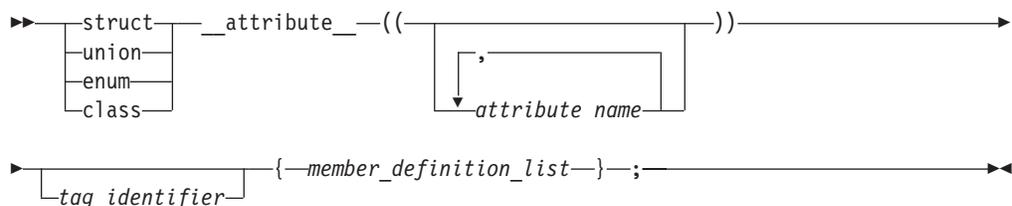
Type attributes

IBM extension

Type attributes are language extensions that allow you to use named attributes to specify special properties of user-defined types. Type attributes apply to the definitions of user-defined types, such as structures, unions, enumerations, classes. Any objects that are declared as having that type will have the attribute applied to them.

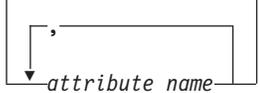
A type attribute is specified with the keyword `__attribute__` followed by the attribute name and any additional arguments the attribute name requires. Although there are variations, the syntax of a type attribute is of the general form:

Type attribute syntax — aggregate types



Type attribute syntax — typedef declarations

```
►► typedef type_declaration type_name
|
|
| ► __attribute__((attribute_name));
```



For unsupported attribute names, the z/OS XL C/C++ compiler issues diagnostics and ignores the attribute specification. Multiple attribute names can be specified in the same attribute specification.

The `armode` type attribute is supported.

Related information

- “Variable attributes” on page 100
- “Function attributes” on page 203

The `armode` | `noarmode` type attribute (C only)

For use with the METAL compiler option, the `armode` type attribute allows you to define a typedef of function or function pointer type as operating in access-register (AR) mode. AR mode allows a C function to access multiple additional data spaces, and manipulate more data in memory.

armode function attribute syntax

```
►► __attribute__((armode | noarmode))
```

Functions in AR mode can call functions not in AR mode, and vice versa.

The following example declares a typedef of function pointer `foo` that is in AR mode, and then declares `bar` as a function that passes function pointer `foo` as a parameter:

```
typedef void (*foo) (int) __attribute__((armode));
void bar (foo);
```

The attribute overrides the default setting of the `ARMODE` compiler option for the specified type. Note that this attribute is only supported when the METAL compiler option is in effect.

Related information

- “The `armode` | `noarmode` function attribute (C only)” on page 204
- “The `__far` type qualifier (C only)” on page 70
- The `ARMODE` and `METAL` options in the *z/OS XL C/C++ User's Guide*

End of IBM extension

Chapter 4. Declarators

This section continues the discussion of data declarations and includes the following topics:

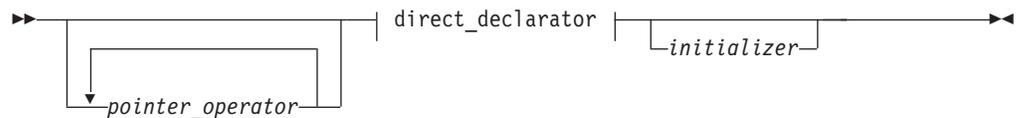
- “Overview of declarators”
- “Type names” on page 79
- “Pointers” on page 80
- “Arrays” on page 84
- “References (C++ only)” on page 87
- “Initializers” on page 88
- “Variable attributes” on page 100

Overview of declarators

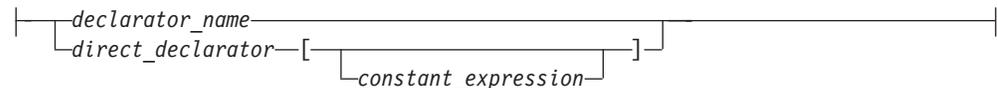
A *declarator* designates a data object or function. A declarator can also include an initialization. Declarators appear in most data definitions and declarations and in some type definitions.

For data declarations, a declarator has the form:

Declarator syntax

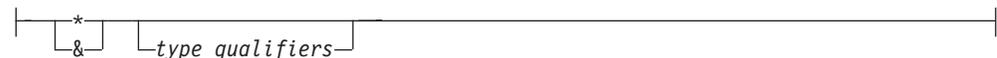


Direct declarator:



C only

Pointer operator:



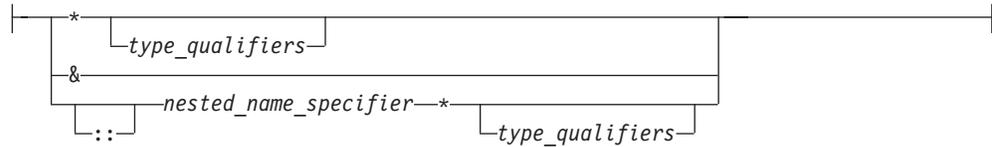
Declarator name:



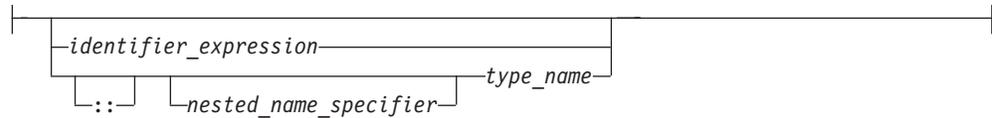
End of C only

C++ only

Pointer operator:



Declarator name:



End of C++ only

The *type_qualifiers* represent one or a combination of `const` and `volatile`.

C++ A *nested_name_specifier* is a qualified identifier expression. An *identifier_expression* can be a qualified or unqualified identifier.

Initializers are discussed in “Initializers” on page 88.

The following are known as *derived declarator* types, and are therefore discussed in this section:

- Pointers
- Arrays
- References (C++ only)

z/OS only

z/OS XL C/C++ includes two additional qualifiers, which are described in “Declarator qualifiers” on page 99.

End of z/OS only

IBM In addition, for compatibility with GNU C and C++, z/OS XL C/C++ allows you to use *variable attributes* to modify the properties of data objects. As they are normally specified as part of the declarator in a declaration, they are described in this section, in “Variable attributes” on page 100.

Related information

- “Type qualifiers” on page 67

Examples of declarators

The following table indicates the declarators within the declarations:

Declaration	Declarator	Description
<code>int owner;</code>	<code>owner</code>	<code>owner</code> is an integer data object.

Declaration	Declarator	Description
<code>int *node;</code>	<code>*node</code>	node is a pointer to an integer data object.
<code>int names[126];</code>	<code>names[126]</code>	names is an array of 126 integer elements.
<code>volatile int min;</code>	<code>min</code>	min is a volatile integer.
<code>int * volatile volume;</code>	<code>* volatile volume</code>	volume is a volatile pointer to an integer.
<code>volatile int * next;</code>	<code>*next</code>	next is a pointer to a volatile integer.
<code>volatile int * sequence[5];</code>	<code>*sequence[5]</code>	sequence is an array of five pointers to volatile integer data objects.
<code>extern const volatile int clock;</code>	<code>clock</code>	clock is a constant and volatile integer with static storage duration and external linkage.
<code>int * __far p;</code>	<code>* __far p</code>	p is a __far pointer to an integer

Related information

- “Type qualifiers” on page 67
- “Array subscripting operator []” on page 138
- “Scope resolution operator :: (C++ only)” on page 116
- “Function declarators” on page 199

Type names

A *type name*, is required in several contexts as something that you must specify without declaring an object; for example, when writing an explicit cast expression or when applying the `sizeof` operator to a type. Syntactically, the name of a data type is the same as a declaration of a function or object of that type, but without the identifier.

To read or write a type name correctly, put an “imaginary” identifier within the syntax, splitting the type name into simpler components. For example, `int` is a type specifier, and it always appears to the left of the identifier in a declaration. An imaginary identifier is unnecessary in this simple case. However, `int *[5]` (an array of 5 pointers to `int`) is also the name of a type. The type specifier `int *` always appears to the left of the identifier, and the array subscripting operator always appears to the right. In this case, an imaginary identifier is helpful in distinguishing the type specifier.

As a general rule, the identifier in a declaration always appears to the left of the subscripting and function call operators, and to the right of a type specifier, type qualifier, or indirection operator. Only the subscripting, function call, and indirection operators may appear in a type name declaration. They bind according to normal operator precedence, which is that the indirection operator is of lower precedence than either the subscripting or function call operators, which have equal ranking in the order of precedence. Parentheses may be used to control the binding of the indirection operator.

It is possible to have a type name within a type name. For example, in a function type, the parameter type syntax nests within the function type name. The same rules of thumb still apply, recursively.

The following constructions illustrate applications of the type naming rules.

Table 17. Type names

Syntax	Description
<code>int *[5]</code>	array of 5 pointers to <code>int</code>
<code>int (*)[5]</code>	pointer to an array of 5 integers
<code>int (*)[*]</code>	pointer to a variable length array of an unspecified number of integers
<code>int *()</code>	function with no parameter specification returning a pointer to <code>int</code>
<code>int *(void)</code>	function with no parameters returning an <code>int</code>
<code>int (*const [])(unsigned int, ...)</code>	array of an unspecified number of constant pointers to functions returning an <code>int</code> . Each function takes one parameter of type <code>unsigned int</code> and an unspecified number of other parameters.

The compiler turns any function designator into a pointer to the function. This behavior simplifies the syntax of function calls.

```
int foo(float); /* foo is a function designator */
int (*p)(float); /* p is a pointer to a function */
p=&foo; /* legal, but redundant */
p=foo; /* legal because the compiler turns foo into a function pointer */
```

C++ In C++, the keywords `typename` and `class`, which are interchangeable, indicate the name of the type.

Related information

- “Operator precedence and associativity” on page 156
- “Examples of expressions and precedence” on page 159
- “The `typename` keyword (C++ only)” on page 349
- “Parenthesized expressions ()” on page 115

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; it cannot refer to a bit field or a reference.

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 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.

C only

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

- Directly from the return value of a library function which returns a pointer
- As an address of a variable
- From constants that refer to valid addresses or from the NULL constant
- Received as a parameter from another C function
- Directly from a call to a service in the z/OS Language Environment that allocates storage, such as CEEGTST

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

End of C only

Note that the placement of the type qualifiers `volatile` and `const` affects the semantics of a pointer declaration. If either of the qualifiers appears before the `*`, the declarator describes a pointer to a type-qualified object. If either of the qualifiers appears between the `*` and the identifier, the declarator describes a type-qualified pointer.

The following table provides examples of pointer declarations.

Table 18. Pointer declarations

Declaration	Description
<code>long *pcoat;</code>	<code>pcoat</code> is a pointer to an object having type <code>long</code>
<code>extern short * const pvolt;</code>	<code>pvolt</code> is a constant pointer to an object having type <code>short</code>
<code>extern int volatile *pnut;</code>	<code>pnut</code> is a pointer to an <code>int</code> object having the <code>volatile</code> qualifier
<code>float * volatile psoup;</code>	<code>psoup</code> is a <code>volatile</code> pointer to an object having type <code>float</code>
<code>enum bird *pfowl;</code>	<code>pfowl</code> is a pointer to an enumeration object of type <code>bird</code>
<code>char (*pvish)(void);</code>	<code>pvish</code> is a pointer to a function that takes no parameters and returns a <code>char</code>

Related information

- “Type qualifiers” on page 67
- “Initialization of pointers” on page 94
- “Compatibility of pointers (C only)” on page 83
- “Pointer conversions” on page 107
- “Address operator `&`” on page 121
- “Indirection operator `*`” on page 122
- “Pointers to functions” on page 214

Pointer arithmetic

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

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

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

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

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

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

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

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

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

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

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

Related information

- “Increment operator ++” on page 118
- “Arrays” on page 84
- “Decrement operator --” on page 119
- Chapter 6, “Expressions and operators,” on page 111

Type-based aliasing

The compiler follows the type-based aliasing rule in the C and C++ standards when the ANSIALIAS option is in effect (which it is by default). This rule, also known as the ANSI aliasing rule, states that a pointer can only be dereferenced to an object of the same type or a compatible type.¹ The common coding practice of casting a pointer to an incompatible type and then dereferencing it violates this rule. (Note

1. The C Standard states that an object shall have its stored value accessed only by an lvalue that has one of the following types:

- the declared type of the object,
- a qualified version of the declared type of the object,
- a type that is the signed or unsigned type corresponding to the declared type of the object,
- a type that is the signed or unsigned type corresponding to a qualified version of the declared type of the object,
- an aggregate or union type that includes one of the aforementioned types among its members (including, recursively, a member of a subaggregate or contained union), or
- a character type

that char pointers are an exception to this rule.) Refer to the description of the ANSIALIAS option in the *z/OS XL C/C++ User's Guide* for additional information.

The compiler uses the type-based aliasing information to perform optimizations to the generated code. Contravening the type-based aliasing rule can lead to unexpected behavior, as demonstrated in the following example:

```
int *p;
double d = 0.0;

int *faa(double *g);          /* cast operator inside the function */

void foo(double f) {
    p = faa(&f);              /* turning &f into a int ptr */
    f += 1.0;                 /* compiler may discard this statement */
    printf("f=%x\n", *p);
}

int *faa(double *g) { return (int*) g; } /* questionable cast; */
                                        /* the function can be in */
                                        /* another translation unit */

int main() {
    foo(d);
}
```

In the above `printf` statement, `*p` cannot be dereferenced to a `double` under the ANSI aliasing rule. The compiler determines that the result of `f += 1.0;` is never used subsequently. Thus, the optimizer may discard the statement from the generated code. If you compile the above example with optimization enabled, the `printf` statement may output 0 (zero).

Related information

- “The `reinterpret_cast` operator (C++ only)” on page 146

Compatibility of pointers (C only)

Two pointer types with the same type qualifiers are compatible if they point to objects of compatible types. The composite type for two compatible pointer types is the similarly qualified pointer to the composite type.

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 C++ standard states that if a program attempts to access the stored value of an object through an lvalue of other than one of the following types, the behavior is undefined:

- the dynamic type of the object,
- a cv-qualified version of the dynamic type of the object,
- a type that is the signed or unsigned type corresponding to the dynamic type of the object,
- a type that is the signed or unsigned type corresponding to a cv-qualified version of the dynamic type of the object,
- an aggregate or union type that includes one of the aforementioned types among its members (including, recursively, a member of a subaggregate or contained union),
- a type that is a (possible cv-qualified) base class type of the dynamic type of the object,
- a char or unsigned char type.

The next example shows incompatible declarations for the assignment operation:

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

z/OS only

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;
    ...}

```

Related information

- “The `_Packed` qualifier (C only)” on page 99

End of z/OS only

Arrays

An *array* is a collection of objects of the same data type, allocated contiguously in memory. Individual objects in an array, called *elements*, are accessed by their position in the array. The subscripting operator (`[]`) provides the mechanics for creating an index to array elements. This form of access is called *indexing* or *subscripting*. An array facilitates the coding of repetitive tasks by allowing the statements executed on each element to be put into a loop that iterates through each element in the array.

The C and C++ languages provide limited built-in support for an array type: reading and writing individual elements. Assignment of one array to another, the comparison of two arrays for equality, returning self-knowledge of size are not supported by either language.

The type of an array is derived from the type of its elements, in what is called *array type derivation*. If array objects are of incomplete type, the array type is also considered incomplete. Array elements may not be of type `void` or of function type. However, arrays of pointers to functions are allowed.  Array elements may not be of reference type or of an abstract class type.

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

Array subscript declarator syntax



The *constant_expression* is a constant integer expression, indicating the size of the array, which must be positive.

C only

If the declaration appears in block or function scope, a nonconstant expression can be specified for the array subscript declarator, and the array is considered a *variable-length array*, as described in “Variable length arrays (C only)” on page 86.

End of C only

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.

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

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

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

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

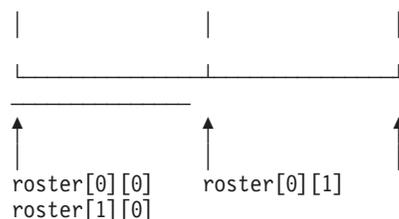
The following example defines a two-dimensional array that contains six elements of type `int`:

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

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

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

In storage, the elements of `roster` would be stored as:



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

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

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

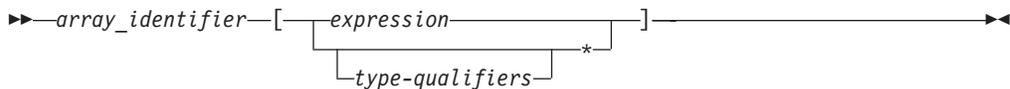
Related information

- “Array subscripting operator []” on page 138
- “Initialization of arrays” on page 95

Variable length arrays (C only)

A variable length array, which is a C99 feature, is an array of automatic storage duration whose length is determined at run time.

Variable length array declarator syntax



If the size of the array is indicated by `*` instead of an expression, the variable length array is considered to be of unspecified size. Such arrays are considered complete types, but can only be used in declarations of function prototype scope.

A variable length array and a pointer to a variable length array are considered *variably modified types*. Declarations of variably modified types must be at either block scope or function prototype scope. Array objects declared with the `extern` storage class specifier cannot be of variable length array type. Array objects declared with the `static` storage class specifier can be a pointer to a variable length array, but not an actual variable length array. The identifiers declared with a variably modified type must be ordinary identifiers and therefore cannot be members of structures or unions. A variable length array cannot be initialized.

A variable length array can be the operand of a `sizeof` expression. In this case, the operand is evaluated at run time, and the size is neither an integer constant nor a constant expression, even though the size of each instance of a variable array does not change during its lifetime.

A variable length array can be used in a `typedef` statement. The `typedef` name will have only block scope. The length of the array is fixed when the `typedef` name is defined, not each time it is used.

A function parameter can be a variable length array. The necessary size expressions must be provided in the function definition. The compiler evaluates the size expression of a variably modified parameter on entry to the function. For a function declared with a variable length array as a parameter, as in the following,

```
void f(int x, int a[][x]);
```

the size of the variable length array argument must match that of the function definition.

Related information

- “Flexible array members (C only)” on page 57

Compatibility of arrays (C only)

Two array types that are similarly qualified are compatible if the types of their elements are compatible. For example,

```
char ex1[25];  
const char ex2[25];
```

are not compatible.

The composite type of two compatible array types is an array with the composite element type. The sizes of both original types must be equivalent if they are known. If the size of only one of the original array types is known, then the composite type has that size. For example:

```
char ex3[];  
char ex4[42];
```

The composite type of `ex3` and `ex4` is `char[42]`. If one of the original types is a variable length array, the composite type is that type.

Related information

- “External linkage” on page 8

References (C++ only)

A *reference* is an alias or an alternative name for an object. All operations applied to a reference act on the object to which the reference refers. The address of a reference is the address of the aliased object.

A reference type is defined by placing the reference modifier `&` after the type specifier. You must initialize all references except function parameters when they are defined. Once defined, a reference cannot be reassigned because it is an alias to its target. What happens when you try to reassign a reference turns out to be the assignment of a new value to the target.

Because arguments of a function are passed 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 or needs to return more than one value, the argument must be *passed by reference* (as opposed to being *passed by value*). Passing arguments by reference can be done using either references or pointers. Unlike C, C++ does not force you to use pointers if you want to pass arguments by reference. The syntax of using a reference is somewhat simpler than that of using a pointer. Passing an object by reference enables the function to change the object being referred to without creating a copy of the object within the scope of the function. Only the address of the actual original object is put on the stack, not the entire object.

For example:

```
int f(int&);
int main()
{
    extern int i;
    f(i);
}
```

You cannot tell from the function call `f(i)` that the argument is being passed by reference.

References to NULL are not allowed.

Related information

- “Initialization of references (C++ only)” on page 98
- “Pointers” on page 80
- “Reference conversions (C++ only)” on page 109
- “Address operator &” on page 121
- “Pass by reference” on page 209

Initializers

An *initializer* is an optional part of a data declaration that specifies an initial value of a data object. The initializers that are legal for a particular declaration depend on the type and storage class of the object to be initialized.

The initializer consists of the = symbol followed by an initial *expression* or a brace-enclosed list of initial expressions separated by commas. Individual expressions must be separated by commas, and groups of expressions can be enclosed in braces and separated by commas. Braces (`{ }`) are optional if the initializer for a character string is a string literal. The number of initializers must not be greater than the number of elements to be initialized. The initial expression evaluates to the first value of the data object.

To assign a value to an arithmetic or pointer type, 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;
```

You initialize a variable of character type with a character literal (consisting of one character) or with an expression that evaluates to an integer.

 You can initialize variables at namespace scope with nonconstant expressions.  You cannot initialize variables at global scope with nonconstant expressions.

“Initialization and storage classes” on page 89 discusses the rules for initialization according to the storage class of variables.

“Designated initializers for aggregate types (C only)” on page 90 describes designated initializers, which are a C99 feature that can be used to initialize arrays, structures, and unions.

The following sections discuss initializations for derived types:

- “Initialization of structures and unions” on page 92

- “Initialization of pointers” on page 94
- “Initialization of arrays” on page 95
- “Initialization of references (C++ only)” on page 98

Related information

- “Using class objects (C++ only)” on page 242

Initialization and storage classes

Initialization of automatic variables

You can initialize any `auto` variable except function parameters. If you do not explicitly initialize an automatic object, its value is indeterminate. If you provide an initial value, the expression representing the initial value can be any valid C or C++ expression. The object is then set to that initial value each time the program block that contains the object’s definition is entered.

Note that if you use the `goto` statement to jump into the middle of a block, automatic variables within that block are not initialized.

Related information

- “The auto storage class specifier” on page 44

Initialization of static variables

You initialize a `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 explicitly initialize a `static` (or `external`) variable, it will have a value of zero of the appropriate type, unless it is a pointer, in which case it will be initialized to `NULL`.

A `static` variable in a block is initialized only one time, prior to program execution, whereas an `auto` variable that has an initializer is initialized every time it comes into existence.

► **C++** A `static` object of class type will use the default constructor if you do not initialize it. Automatic and register variables that are not initialized will have undefined values.

Related information

- “The static storage class specifier” on page 44

Initialization of external variables

You can initialize any object with the `extern` storage class specifier at global scope in C or at namespace scope in C++. The initializer for an `extern` object must either:

- Appear as part of the definition and the initial value must be described by a constant expression; or
- Reduce to the address of a previously declared object with `static` storage duration. You may modify this object with pointer arithmetic. (In other words, you may modify the object by adding or subtracting an integral constant expression.)

If you do not explicitly initialize an `extern` variable, its initial value is zero of the appropriate type. Initialization of an `extern` object is completed by the time the program starts running.

Related information

- “The extern storage class specifier” on page 46

Initialization of register variables

You can initialize any register object except function parameters. If you do not initialize an automatic object, its value is indeterminate. If you provide an initial value, the expression representing the initial value can be any valid C or C++ expression. The object is then set to that initial value each time the program block that contains the object’s definition is entered.

Related information

- “The register storage class specifier” on page 47

Designated initializers for aggregate types (C only)

Designated initializers, a C99 feature, are supported for aggregate types, including arrays, structures, and unions. A designated initializer, or *designator*, points out a particular element to be initialized. A *designator list* is a comma-separated list of one or more designators. A designator list followed by an equal sign constitutes a *designation*.

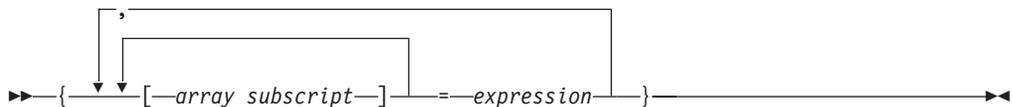
Designated initializers allow for the following flexibility:

- Elements within an aggregate can be initialized in any order.
- The initializer list can omit elements that are declared anywhere in the aggregate, rather than only at the end. Elements that are omitted are initialized as if they are static objects: arithmetic types are initialized to 0; pointers are initialized to NULL.
- Where inconsistent or incomplete bracketing of initializers for multi-dimensional arrays or nested aggregates may be difficult to understand, designators can more clearly identify the element or member to be initialized.

Designator list syntax for structures and unions



Designator list syntax for arrays



In the following example, the designator is `.any_member` and the designated initializer is `.any_member = 13`:

```
union { /* ... */ } caw = { .any_member = 13 };
```

The following example shows how the second and third members `b` and `c` of structure variable `k1m` are initialized with designated initializers:

```
struct xyz {
    int a;
    int b;
    int c;
} k1m = { .a = 99, .c = 100 };
```

In the following example, the third and second elements of the one-dimensional array `aa` are initialized to 3 and 6, respectively:

```
int aa[4] = { [2] = 3, [1] = 6 };
```

The following example initializes the first four and last four elements, while omitting the middle four:

```
static short grid[3][4] = { [0][0]=8, [0][1]=6,
                           [0][2]=4, [0][3]=1,
                           [2][0]=9, [2][1]=3,
                           [2][2]=1, [2][3]=1 };
```

The omitted four elements of `grid` are initialized to zero:

Element	Value	Element	Value
<code>grid[0][0]</code>	8	<code>grid[1][2]</code>	0
<code>grid[0][1]</code>	6	<code>grid[1][3]</code>	0
<code>grid[0][2]</code>	4	<code>grid[2][0]</code>	9
<code>grid[0][3]</code>	1	<code>grid[2][1]</code>	3
<code>grid[1][0]</code>	0	<code>grid[2][2]</code>	1
<code>grid[1][1]</code>	0	<code>grid[2][3]</code>	1

Designated initializers can be combined with regular initializers, as in the following example:

```
int a[10] = {2, 4, [8]=9, 10}
```

In this example, `a[0]` is initialized to 2, `a[1]` is initialized to 4, `a[2]` to `a[7]` are initialized to 0, and `a[9]` is initialized to 10.

In the following example, a single designator is used to "allocate" space from both ends of an array:

```
int a[MAX] = {
    1, 3, 5, 7, 9, [MAX-5] = 8, 6, 4, 2, 0
};
```

The designated initializer, `[MAX-5] = 8`, means that the array element at subscript `MAX-5` should be initialized to the value 8. If `MAX` is 15, `a[5]` through `a[9]` will be initialized to zero. If `MAX` is 7, `a[2]` through `a[4]` will first have the values 5, 7, and 9, respectively, which are overridden by the values 8, 6, and 4. In other words, if `MAX` is 7, the initialization would be the same as if the declaration had been written:

```
int a[MAX] = {
    1, 3, 8, 6, 4, 2, 0
};
```

You can also use designators to represent members of nested structures. For example:

```
struct a {
    struct b {
        int c;
        int d;
    } e;
    float f;
} g = {.e.c = 3};
```

initializes member `c` of structure variable `e`, which is a member of structure variable `g`, to the value of 3.

Related information

- “Initialization of structures and unions”
- “Initialization of arrays” on page 95

Initialization of structures and unions

An initializer for a structure is a brace-enclosed comma-separated list of values, and for a union, a brace-enclosed single value. The initializer is preceded by an equal sign (=).

C99 and C++ allow the initializer for an automatic member variable of a union or structure type to be a constant or non-constant expression.

C++ The initializer for a static member variable of a union or structure type must be a constant expression or string literal. See “Static data members (C++ only)” on page 261 for more information.

There are two ways to specify initializers for structures and unions:

- With C89-style initializers, structure members must be initialized in the order declared, and only the first member of a union can be initialized.
- **C** Using *designated* initializers, a C99 feature which allows you to *name* members to be initialized, structure members can be initialized in any order, and any (single) member of a union can be initialized. Designated initializers are described in detail in “Designated initializers for aggregate types (C only)” on page 90.

Using C89-style initialization, 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"};
```

C only

Using a designated initializer in the same example, the following initializes the second union member `age` :

```
union {
    char birthday[9];
    int age;
    float weight;
} people = { .age = 14 };
```

End of C only

The following definition shows a completely initialized structure:

```
struct address {
    int street_no;
    char *street_name;
    char *city;
    char *prov;
    char *postal_code;
```

```

};
static struct address perm_address =
    { 3, "Savona Dr.", "Dundas", "Ontario", "L4B 2A1"};

```

The values of perm_address are:

Member	Value
perm_address.street_no	3
perm_address.street_name	address of string "Savona Dr."
perm_address.city	address of string "Dundas"
perm_address.prov	address of string "Ontario"
perm_address.postal_code	address of string "L4B 2A1"

Unnamed structure or union members do not participate in initialization and have indeterminate value after initialization. Therefore, in the following example, the bit field is not initialized, and the initializer 3 is applied to member b:

```

struct {
    int a;
    int :10;
    int b;
} w = { 2, 3 };

```

You do not have to initialize all members of a structure or union; the initial value of uninitialized structure members depends on the storage class associated with the structure or union variable. In a structure declared as static, any members that are not initialized are implicitly initialized to zero of the appropriate type; the members of a structure with automatic storage have no default initialization. The default initializer for a union with static storage is the default for the first component; a union with automatic storage has no default initialization.

The following definition shows a partially initialized structure:

```

struct address {
    int street_no;
    char *street_name;
    char *city;
    char *prov;
    char *postal_code;
};
struct address temp_address =
    { 44, "Knyvet Ave.", "Hamilton", "Ontario" };

```

The values of temp_address are:

Member	Value
temp_address.street_no	44
temp_address.street_name	address of string "Knyvet Ave."
temp_address.city	address of string "Hamilton"
temp_address.prov	address of string "Ontario"
temp_address.postal_code	Depends on the storage class of the temp_address variable; if it is static, the value would be NULL.

C only

To initialize only the third and fourth members of the `temp_address` variable, you could use a designated initializer list, as follows:

```
struct address {
    int street_no;
    char *street_name;
    char *city;
    char *prov;
    char *postal_code;
};
struct address temp_address =
    { .city = "Hamilton", .prov = "Ontario" };
```

End of C only

Related information

- “Structure and union variable declarations” on page 59
- “Explicit initialization with constructors (C++ only)” on page 302
- “Assignment operators” on page 128

Initialization of enumerations

The initializer for an enumeration variable contains the = symbol followed by an expression *enumeration_constant*.

 In C++, the initializer must have the same type as the associated enumeration type.

The first line of the following example declares the enumeration `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;
```

Related information

- “Enumeration variable declarations” on page 63

Initialization of pointers

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

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

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

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

is equivalent to:

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

You can assign the address of the first character in a string constant to a pointer by specifying the string constant in the initializer. The following example defines the pointer variable `string` and the string constant "abcd". The pointer `string` is initialized to point to the character `a` in the string "abcd".

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

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

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

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

Related information

- "Pointers" on page 80

Initialization of arrays

The initializer for an array is a comma-separated list of constant expressions enclosed in braces (`{ }`). The initializer is preceded by an equal sign (`=`). You do not need to initialize all elements in an array. If an array is partially initialized, elements that are not initialized receive the value 0 of the appropriate type. The same applies to elements of arrays with static storage duration. (All file-scope variables and function-scope variables declared with the `static` keyword have static storage duration.)

There are two ways to specify initializers for arrays:

- With C89-style initializers, array elements must be initialized in subscript order.
-  Using *designated* initializers, which allow you to specify the values of the subscript elements to be initialized, array elements can be initialized in any order. Designated initializers are described in detail in "Designated initializers for aggregate types (C only)" on page 90.

Using C89-style initializers, the following definition shows a completely initialized one-dimensional array:

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

The array `number` contains the following values: `number[0]` is 5, `number[1]` is 7; `number[2]` is 2. When you have an expression in the subscript declarator defining the number of elements (in this case 3), you cannot have more initializers than the number of elements in the array.

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

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

The values of `number1[0]` and `number1[1]` are the same as in the previous definition, but `number1[2]` is 0.

C only

The following definition shows how you can use designated initializers to skip over elements of the array that you don't want to initialize explicitly:

```
static int number[3] = { [0] = 5, [2] = 7 };
```

The array `number` contains the following values: `number[0]` is 5; `number[1]` is implicitly initialized to 0; `number[2]` is 7.

End of C only

Instead of an expression in the subscript declarator defining 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, because no size was specified and there are five initializers.

Initialization of character arrays

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 surrounding the constant are optional)

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

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 name3[3]="Jan";
```

 When you initialize an array of characters with a string, the number of characters in the string — including the terminating `'\0'` — must not exceed the number of elements in the array.

Initialization of multidimensional arrays

You can initialize a multidimensional array using any of the following techniques:

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

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

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

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

- Using nested braces to initialize dimensions and elements in a dimension selectively. In the following example, only the first eight elements of the array `grid` are explicitly initialized. The remaining four elements that are not explicitly initialized are automatically initialized to zero.

```
static short grid[3][4] = {8, 6, 4, 1, 9, 3, 1, 1};
```

The initial values of `grid` are:

Element	Value	Element	Value
<code>grid[0][0]</code>	8	<code>grid[1][2]</code>	1
<code>grid[0][1]</code>	6	<code>grid[1][3]</code>	1
<code>grid[0][2]</code>	4	<code>grid[2][0]</code>	0
<code>grid[0][3]</code>	1	<code>grid[2][1]</code>	0
<code>grid[1][0]</code>	9	<code>grid[2][2]</code>	0
<code>grid[1][1]</code>	3	<code>grid[2][3]</code>	0

C only

- Using *designated* initializers. The following example uses designated initializers to explicitly initialize only the last four elements of the array. The first eight elements that are not explicitly initialized are automatically initialized to zero.

```
static short grid[3][4] = { [2][0] = 8, [2][1] = 6,
                           [2][2] = 4, [2][3] = 1 };
```

The initial values of `grid` are:

Element	Value	Element	Value
<code>grid[0][0]</code>	0	<code>grid[1][2]</code>	0
<code>grid[0][1]</code>	0	<code>grid[1][3]</code>	0
<code>grid[0][2]</code>	0	<code>grid[2][0]</code>	8

Element	Value	Element	Value
grid[0] [3]	0	grid[2] [1]	6
grid[1] [0]	0	grid[2] [2]	4
grid[1] [1]	0	grid[2] [3]	1

End of C only

Related information

- “Arrays” on page 84
- “Designated initializers for aggregate types (C only)” on page 90

Initialization of references (C++ only)

The object that you use to initialize a reference must be of the same type as the reference, or it must be of a type that is convertible to the reference type. If you initialize a reference to a constant using an object that requires conversion, a temporary object is created. In the following example, a temporary object of type float is created:

```
int i;
const float& f = i; // reference to a constant float
```

When you initialize a reference with an object, you *bind* that reference to that object.

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 referred to.

A reference can be declared without an initializer:

- When it is used in an parameter declaration
- In the declaration of a return type for a function call
- In the declaration of class member within its class declaration
- When the extern specifier is explicitly used

You cannot have references to any of the following:

- Other references
- Bit fields

- Arrays of references
- Pointers to references

Direct binding

Suppose a reference *r* of type *T* is initialized by an expression *e* of type *U*.

The reference *r* is *bound directly* to *e* if the following statements are true:

- Expression *e* is an lvalue
- *T* is the same type as *U*, or *T* is a base class of *U*
- *T* has the same, or more, `const` or `volatile` qualifiers than *U*

The reference *r* is also bound directly to *e* if *e* can be implicitly converted to a type such that the previous list of statements is true.

Related information

- “References (C++ only)” on page 87
- “Pass by reference” on page 209

Declarator qualifiers

z/OS only

z/OS XL C/C++ includes two additional qualifiers that are given in declarator specifications:

-  `_Packed`
-  `_Export`

The `_Packed` qualifier (C only)

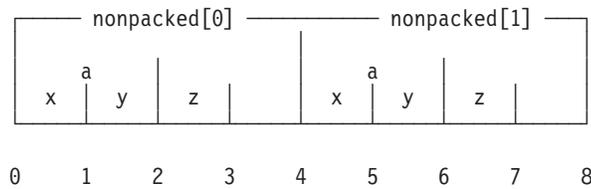
The z/OS XL C compiler aligns structure and union members according to their natural byte boundaries and ends the structure or union on its natural boundary. However, since the alignment of a structure or union is that of the member with the largest alignment requirement, the compiler may add padding to elements whose byte boundaries are smaller than this requirement. You can use the `_Packed` qualifier to remove padding between members of structures or unions. Packed and nonpacked structures and unions have different storage layouts.

Consider the following example:

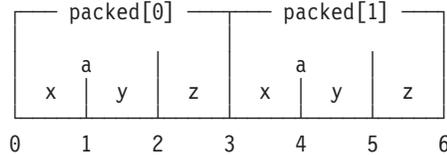
```
union uu{
    short    a;
    struct {
        char x;
        char y;
        char z;
    } b;
};

union uu          nonpacked[2];
_Packed union uu packed[2];
```

In the array of unions `nonpacked`, since the largest alignment requirement among the union members is that of `short a`, namely, 2 bytes, one byte of padding is added at the end of each union in the array to enforce this requirement:



In the array of unions packed, each union has a length of only 3 bytes, as opposed to the 4 bytes of the previous case:



Note: The compiler aligns pointers on their natural boundaries, 4 bytes, even in packed structures and unions.

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 nested structure or union.

Related information

- “Compatibility of structures, unions, and enumerations (C only)” on page 64
- “`#pragma pack`” on page 435

The `_Export` qualifier (C++ only)

You can use the `_Export` keyword with a function name or external variable to declare that it is to be exported (made available to other modules). The `_Export` keyword must immediately precede the object name. For more information, see “The `_Export` function specifier (C++ only)” on page 197.

Related information

- “External linkage” on page 8
- “`#pragma export`” on page 409

End of z/OS only

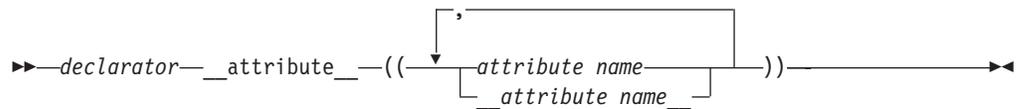
Variable attributes

IBM extension

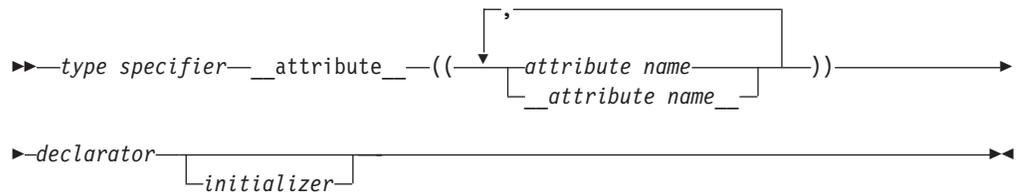
A variable attribute is a language extension that allows you to use a named attribute to specify special properties of variables. Currently, only the variable attribute `aligned` is supported on the z/OS platform.

A variable attribute is specified with the keyword `__attribute__` followed by the attribute name and any additional arguments the attribute name requires. A variable `__attribute__` specification is included in the declaration of a variable, and can be placed before or after the declarator. Although there are variations, the syntax generally takes either of the following forms:

Variable attribute syntax: post-declarator



Variable attribute syntax: pre-declarator



The *attribute name* can be specified with or without leading and trailing double underscore characters; however, using the double underscore reduces the likelihood of a name conflict with a macro of the same name. For unsupported attribute names, the z/OS XL C/C++ compiler issues diagnostics and ignores the attribute specification. Multiple attribute names can be specified in the same attribute specification.

In a comma-separated list of declarators on a single declaration line, if a variable attribute appears before all the declarators, it applies to all declarators in the declaration. If the attribute appears after a declarator, it only applies to the immediately preceding declarator. For example:

```
struct A {  
  
    int b __attribute__((aligned));           /* typical placement of variable */  
                                           /* attribute */  
  
    int __attribute__((aligned))__ c = 10; /* variable attribute can also be */  
                                           /* placed here */  
  
    int d, e, f __attribute__((aligned));    /* attribute applies to f only */  
  
    int g __attribute__((aligned)), h, i;    /* attribute applies to g only */  
  
    int __attribute__((aligned)) j, k, l;    /* attribute applies to j, k, and l */  
  
};
```

The aligned variable attribute

The aligned variable attribute allows you to override the default alignment mode to specify a minimum alignment value, expressed as a number of bytes, for any of the following:

- a non-aggregate variable
- an aggregate variable (such as a structure, class, or union)
- selected member variables

The attribute is typically used to increase the alignment of the given variable.

aligned variable attribute syntax

►► `__attribute__((aligned __aligned__ (alignment_factor)))` ►►

The *alignment_factor* is the number of bytes, specified as a constant expression that evaluates to a positive power of 2. On the z/OS platform, the maximum supported value is 8 bytes in 32-bit mode, and 16 bytes in 64-bit mode. If you omit the alignment factor (and its enclosing parentheses), the compiler automatically uses the platform maximum. If you specify an alignment factor greater than the maximum, the attribute specification is ignored, and the compiler simply uses the default alignment in effect.

When you apply the aligned attribute to a bit field structure member variable, the attribute specification is applied to the bit field *container*. If the default alignment of the container is greater than the alignment factor, the default alignment is used.

In the following example, the structures `first_address` and `second_address` are set to an alignment of 16 bytes:

```
struct address {
    int street_no;
    char *street_name;
    char *city;
    char *prov;
    char *postal_code;
} first_address __attribute__((__aligned__(16))) ;

struct address second_address __attribute__((__aligned__(16))) ;
```

In the following example, only the members `first_address.prov` and `first_address.postal_code` are set to an alignment of 16 bytes:

```
struct address {
    int street_no;
    char *street_name;
    char *city;
    char *prov __attribute__((__aligned__(16))) ;
    char *postal_code __attribute__((__aligned__(16))) ;
} first_address ;
```

End of IBM extension

Chapter 5. Type conversions

An expression of a given type is *implicitly converted* in the following situations:

- The expression is used as an operand of an arithmetic or logical operation.
- The expression is used as a condition in an `if` statement or an iteration statement (such as a `for` loop). The expression will be converted to a Boolean (or an integer in C89).
- The expression is used in a `switch` statement. The expression will be converted to an integral type.
- The expression is used as an initialization. This includes the following:
 - An assignment is made to an lvalue that has a different type than the assigned value.
 - A function is provided an argument value that has a different type than the parameter.
 - The value specified in the return statement of a function has a different type from the defined return type for the function.

You can perform *explicit* type conversions using a `cast` expression, as described in “Cast expressions” on page 143. The following sections discuss the conversions that are allowed by either implicit or explicit conversion, and the rules governing type promotions:

- “Arithmetic conversions and promotions”
- “Lvalue-to-rvalue conversions” on page 107
- “Pointer conversions” on page 107
- “Reference conversions (C++ only)” on page 109
- “Qualification conversions (C++ only)” on page 109
- “Function argument conversions” on page 110

Related information

- “User-defined conversions (C++ only)” on page 311
- “Conversion constructors (C++ only)” on page 312
- “Conversion functions (C++ only)” on page 314
- “The `switch` statement” on page 166
- “The `if` statement” on page 164
- “The return statement” on page 175

Arithmetic conversions and promotions

The following sections discuss the rules for the standard conversions for arithmetic types:

- “Integral conversions” on page 104
- “Floating-point conversions” on page 104
- 105
- “Boolean conversions” on page 104
-  “Packed decimal conversions (C only)” on page 105

If two operands in an expression have different types, they are subject to the rules of the *usual arithmetic conversions*, as described in “Integral and floating-point promotions” on page 106.

Integral conversions

Unsigned integer to unsigned integer or signed integer to signed integer

If the types are identical, there is no change. If the types are of a different size, and the value can be represented by the new type, the value is not changed; if the value cannot be represented by the new type, truncation or sign shifting will occur.

Signed integer to unsigned integer

The resulting value is the smallest unsigned integer type congruent to the source integer. If the value cannot be represented by the new type, truncation or sign shifting will occur.

Unsigned integer to signed integer

If the signed type is large enough to hold the original value, there is no change. If the value can be represented by the new type, the value is not changed; if the value cannot be represented by the new type, truncation or sign shifting will occur.

Signed and unsigned character types to integer

If the original value can be represented by `int`, it is represented as `int`. If the value cannot be represented by `int`, it is promoted to unsigned `int`.

Wide character type `wchar_t` to integer

If the original value can be represented by `int`, it is represented as `int`. If the value cannot be represented by `int`, it is promoted to the smallest type that can hold it: unsigned `int`, `long`, or unsigned `long`.

Signed and unsigned integer bit field to integer

If the original value can be represented by `int`, it is represented as `int`. If the value cannot be represented by `int`, it is promoted to unsigned `int`.

Enumeration type to integer

If the original value can be represented by `int`, it is represented as `int`. If the value cannot be represented by `int`, it is promoted to the smallest type that can hold it: unsigned `int`, `long`, or unsigned `long`. Note that an enumerated type can be converted to an integral type, but an integral type cannot be converted to an enumeration.

Boolean conversions

Boolean to integer

C If the Boolean value is 0, the result is an `int` with a value of 0. If the Boolean value is 1, the result is an `int` with a value of 1. **C++** If the Boolean value is `false`, the result is an `int` with a value of 0. If the Boolean value is `true`, the result is an `int` with a value of 1.

Scalar to Boolean

C If the scalar value is equal to 0, the Boolean value is 0; otherwise the Boolean value is 1. **C++** A zero, null pointer, or null member pointer value is converted to `false`. All other values are converted to `true`.

Floating-point conversions

The standard rule for converting between real floating-point types (binary to binary, decimal to decimal and decimal to binary) is as follows:

If the value being converted can be represented exactly in the new type, it is unchanged. If the value being converted is in the range of values that can be

represented but cannot be represented exactly, the result is rounded, according to the current compile-time or runtime rounding mode in effect. If the value being converted is outside the range of values that can be represented, the result is dependant on the rounding mode.

Integer to floating-point (binary or decimal)

If the value being converted can be represented exactly in the new type, it is unchanged. If the value being converted is in the range of values that can be represented but cannot be represented exactly, the result is correctly rounded. If the value being converted is outside the range of values that can be represented, the result is quiet NaN.

Floating-point (binary or decimal) to integer

The fractional part is discarded (i.e., the value is truncated toward zero). If the value of the integral part cannot be represented by the integer type, the result is one of the following:

- If the integer type is unsigned, the result is the largest representable number if the floating-point number is positive, or 0 otherwise.
- If the integer type is signed, the result is the most negative or positive representable number according to the sign of the floating-point number.

Complex conversions (C only)

Complex to complex

If the types are identical, there is no change. If the types are of a different size, and the value can be represented by the new type, the value is not changed; if the value cannot be represented by the new type, both real and imaginary parts are converted according to the standard conversion rule given above.

Complex to real (binary)

The imaginary part of the complex value is discarded. If necessary, the value of the real part is converted according to the standard conversion rule given above.

Complex to real (decimal)

The imaginary part of the complex value is discarded. The value of the real part is converted from binary to decimal floating-point, according to the standard conversion rule given above.

Real (binary) to complex

The source value is used as the real part of the complex value, and converted, if necessary, according to the standard conversion rule given above. The value of the imaginary part is zero.

Real (decimal) to complex

The source value is converted from decimal to binary floating-point, according to the standard conversion rule given above, and used as the real part of the complex value. The value of the imaginary part is zero.

Packed decimal conversions (C only)

z/OS only

Packed decimal to long long integer

The fractional part is discarded.

Long long integer to packed decimal

The resulting size is `decimal(20,0)`.

Complex to packed decimal

Only the floating value of the real part is used.

Packed decimal to complex

The real part of the complex type is converted, and the imaginary part is 0.

Packed decimal to decimal floating-point

If the number of significant digits in the packed decimal value exceeds the precision of the target, the result is rounded to the target precision using the current decimal floating-point rounding mode.

Decimal floating-point to packed decimal

Before conversion, the decimal floating-point value is rounded or truncated to match the fractional precision of the resulting type, if necessary. If the value being converted represents infinity or NaN, or if non-zero digits are truncated from the left end of the result, the result is undefined.

End of z/OS only

Integral and floating-point promotions

When different arithmetic types are used as operands in certain types of expressions, standard conversions known as *usual arithmetic conversions* are applied. These conversions are applied according to the rank of the arithmetic type: the operand with a type of lower rank is converted to the type of the operand with a higher rank. This is known as integral or floating point *promotion*.

For example, when the values of two different integral types are added together, both values are first converted to the same type: when a short int value and an int value are added together, the short int value is converted to the int type. Chapter 6, “Expressions and operators,” on page 111 provides a list of the operators and expressions that participate in the usual arithmetic conversions.

The ranking of arithmetic types, listed from highest to lowest, is as follows:

Table 19. Conversion rankings for floating-point types

Operand type
long double or  long double _Complex
double or  double _Complex
float or  float _Complex

Table 20. Conversion rankings for decimal floating-point types

Operand type
 _Decimal128
 _Decimal64
 _Decimal32

Table 21. Conversion rankings for integer types

Operand type
unsigned long long or unsigned long long int

Table 21. Conversion rankings for integer types (continued)

Operand type
long long or long long int
unsigned long int
long int ¹
unsigned int ¹
int and enumerated types
short int
char, signed char and unsigned char
Boolean

Notes:

1. If one operand has unsigned int type and the other operand has long int type but the value of the unsigned int cannot be represented in a long int, both operands are converted to unsigned long int.

Related information

- “Integral types” on page 49
- “Boolean types” on page 50
- “Floating-point types” on page 51
- “Character types” on page 54
- “Enumerations” on page 61
- “Binary expressions” on page 127

Lvalue-to-rvalue conversions

If an lvalue appears in a situation in which the compiler expects an rvalue, the compiler converts the lvalue to an rvalue. The following table lists exceptions to this:

Situation before conversion	Resulting behavior
The lvalue is a function type.	
The lvalue is an array.	
The type of the lvalue is an incomplete type.	compile-time error
The lvalue refers to an uninitialized object.	undefined behavior
The lvalue refers to an object not of the type of the rvalue, nor of a type derived from the type of the rvalue.	undefined behavior
 The lvalue is a nonclass type, qualified by either const or volatile.	The type after conversion is not qualified by either const or volatile.

Related information

- “Lvalues and rvalues” on page 111

Pointer conversions

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

C only

Conversions that involve pointers must use an explicit type cast. The exceptions to this rule are the allowable assignment conversions for C pointers. In the following table, a const-qualified lvalue cannot be used as a left operand of the assignment.

Table 22. Legal assignment conversions for C pointers

Left operand type	Permitted right operand types
pointer to (object) T	<ul style="list-style-type: none">the constant 0a pointer to a type compatible with Ta pointer to void (void*)
pointer to (function) F	<ul style="list-style-type: none">the constant 0a pointer to a function compatible with F

The referenced type of the left operand must have the same qualifiers as the right operand. An object pointer may be an incomplete type if the other pointer has type void*.

End of C only

Zero constant to null pointer

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

 C++ A constant expression that evaluates to zero can also be converted to the null pointer to a member.

Array to pointer

An lvalue or rvalue with type "array of *N*," where *N* is the type of a single element of the array, to *N**. The result is a pointer to the initial element of the array. A conversion cannot be performed if the expression is used as the operand of the & (address) operator or the sizeof operator.

Function to pointer

An lvalue that is a function can be converted to an rvalue that is a pointer to a function of the same type, except when the expression is used as the operand of the & (address) operator, the () (function call) operator, or the sizeof operator.

Related information

- "Pointers" on page 80
- "Integer constant expressions" on page 113
- "Arrays" on page 84
- "Pointers to functions" on page 214
- "Pointers to members (C++ only)" on page 256
- "Pointer conversions" on page 290

Conversion to void*

C pointers are not necessarily the same size as type int. Pointer arguments given to functions should be explicitly cast to ensure that the correct type expected by the function is being passed. The generic object pointer in C is void*, but there is no generic function pointer.

Any pointer to an object, optionally type-qualified, can be converted to `void*`, keeping the same `const` or `volatile` qualifications.

C only

The allowable assignment conversions involving `void*` as the left operand are shown in the following table.

Table 23. Legal assignment conversions in C for `void`*

Left operand type	Permitted right operand types
<code>(void*)</code>	<ul style="list-style-type: none">• The constant 0.• A pointer to an object. The object may be of incomplete type.• <code>(void*)</code>

End of C only

C++ only

Pointers to functions cannot be converted to the type `void*` with a standard conversion: this can be accomplished explicitly, provided that a `void*` has sufficient bits to hold it.

End of C++ only

Related information

- “The void type” on page 54

Reference conversions (C++ only)

A reference conversion can be performed wherever a reference initialization occurs, including reference initialization done in argument passing and function return values. A reference to a class can be converted 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.

Reference conversion is allowed if the corresponding pointer conversion is allowed.

Related information

- “References (C++ only)” on page 87
- “Initialization of references (C++ only)” on page 98
- “Function calls” on page 207
- “Function return values” on page 199

Qualification conversions (C++ only)

An type-qualified rvalue of any type, containing zero or more `const` or `volatile` qualifications, can be converted to an rvalue of type-qualified type where the second rvalue contains more `const` or `volatile` qualifications than the first rvalue.

An rvalue of type pointer to member of a class can be converted to an rvalue of type pointer to member of a class if the second rvalue contains more `const` or `volatile` qualifications than the first rvalue.

Related information

- “Type qualifiers” on page 67

Function argument conversions

When a function is called, if a function declaration is present and includes declared argument types, the compiler performs type checking. The compiler compares the data types provided by the calling function with the data types that the called function expects and performs necessary type conversions. For example, when function `funct` is called, argument `f` is converted to a `double`, and argument `c` is converted to an `int`:

```
char * funct (double d, int i);
    /* ... */
int main(void)
{
    float f;
    char c;
    funct(f, c) /* f is converted to a double, c is converted to an int */
    return 0;
}
```

If no function declaration is visible when a function is called, or when an expression appears as an argument in the variable part of a prototype argument list, the compiler performs default argument promotions or converts the value of the expression before passing any arguments to the function. The automatic conversions consist of the following:

- Integral and floating-point values are promoted.
- Arrays or functions are converted to pointers.
-  Non-static class member functions are converted to pointers to members.

Related information

- “Integral and floating-point promotions” on page 106
- “Function call expressions” on page 116
- “Function calls” on page 207

Chapter 6. Expressions and operators

Expressions are sequences of operators, operands, and punctuators that specify a computation. The evaluation of expressions is based on the operators that the expressions contain and the context in which they are used. An expression can result in a value and can produce *side effects*. A side effect is a change in the state of the execution environment.

The following sections describe these types of expressions:

- “Lvalues and rvalues”
- “Primary expressions” on page 112
- “Function call expressions” on page 116
- “Member expressions” on page 117
- “Unary expressions” on page 118
- “Binary expressions” on page 127
- “Conditional expressions” on page 141
- “Compound literal expressions (C only)” on page 151
- “Cast expressions” on page 143
- “new expressions (C++ only)” on page 151
- “delete expressions (C++ only)” on page 155
- “throw expressions (C++ only)” on page 156

“Operator precedence and associativity” on page 156 provides tables listing the precedence of all the operators described in the various sections listed above.

 C++ operators can be defined to behave differently when applied to operands of class type. This is called operator *overloading*, and is described in “Overloading operators (C++ only)” on page 227.

Lvalues and rvalues

An *object* is a region of storage that can be examined and stored into. An *lvalue* is an expression that refers to such an object. An lvalue does not necessarily permit modification of the object it designates. For example, a `const` object is an lvalue that cannot be modified. The term *modifiable lvalue* is used to emphasize that the lvalue allows the designated object to be changed as well as examined. The following object types are lvalues, but not modifiable lvalues:

- An array type
- An incomplete type
- A `const`-qualified type
- A structure or union type with one of its members qualified as a `const` type

Because these lvalues are not modifiable, they cannot appear on the left side of an assignment statement.

The term *rvalue* refers to a data value that is stored at some address in memory. An rvalue is an expression that cannot have a value assigned to it. Both a literal constant and a variable can serve as an rvalue. When an lvalue appears in a context that requires an rvalue, the lvalue is implicitly converted to an rvalue. The reverse, however, is not true: an rvalue cannot be converted to an lvalue. Rvalues always have complete types or the void type.

C C defines a *function designator* as an expression that has function type. A function designator is distinct from an object type or an lvalue. It can be the name of a function or the result of dereferencing a function pointer. The C language also differentiates between its treatment of a function pointer and an object pointer.

C++ On the other hand, in C++, a function call that returns a reference is an lvalue. Otherwise, a function call is an rvalue expression. In C++, every expression produces an lvalue, an rvalue, or no value.

In both C and C++, certain operators require lvalues for some of their operands. The table below lists these operators and additional constraints on their usage.

Operator	Requirement
& (unary)	Operand must be an lvalue.
++ --	Operand must be an lvalue. This applies to both prefix and postfix forms.
= += -= *= %= <<= >>= &= ^= =	Left operand must be an lvalue.

For example, all assignment operators evaluate their right operand and assign that value to their left operand. The left operand must be a modifiable lvalue or a reference to a modifiable object.

The address operator (&) requires an lvalue as an operand while the increment (++) and the decrement (--) operators require a modifiable lvalue as an operand. The following example shows expressions and their corresponding lvalues.

Expression	Lvalue
x = 42	x
*ptr = newvalue	*ptr
a++	a
C++ int& f()	The function call to f()

Related information

- “Arrays” on page 84
- “Lvalue-to-rvalue conversions” on page 107

Primary expressions

Primary expressions fall into the following general categories:

- Names (identifiers)
- Literals (constants)
- Integer constant expressions
- Identifier expressions (C++ only)
- Parenthesized expressions ()
- **C++** The `this` pointer (described in “The `this` pointer (C++ only)” on page 257)
- **C++** Names qualified by the scope resolution operator (`::`)

Names

The value of a name depends on its type, which is determined by how that name is declared. The following table shows whether a name is an lvalue expression.

Table 24. Primary expressions: Names

Name declared as	Evaluates to	Is an lvalue?
Variable of arithmetic, pointer, enumeration, structure, or union type	An object of that type	yes
Enumeration constant	The associated integer value	no
Array	That array. In contexts subject to conversions, a pointer to the first object in the array, except where the name is used as the argument to the <code>sizeof</code> operator.	 no  yes
Function	That function. In contexts subject to conversions, a pointer to that function, except where the name is used as the argument to the <code>sizeof</code> operator, or as the function in a function call expression.	 no  yes

As an expression, a name may not refer to a label, typedef name, structure member, union member, structure tag, union tag, or enumeration tag. Names used for these purposes reside in a namespace that is separate from that of names used in expressions. However, some of these names may be referred to within expressions by means of special constructs: for example, the dot or arrow operators may be used to refer to structure and union members; typedef names may be used in casts or as an argument to the `sizeof` operator.

Literals

A literal is a numeric constant or string literal. When a literal is evaluated as an expression, its value is a constant. A lexical constant is never an lvalue. However, a string literal is an lvalue.

Related information

- “Literals” on page 19
- “The this pointer (C++ only)” on page 257

Integer constant expressions

An *integer compile-time constant* is a value that is determined during compilation and cannot be changed at run time. An *integer compile-time constant expression* is an expression that is composed of constants and evaluated to a constant.

An integer constant expression is an expression that is composed of only the following:

- literals
- enumerators

- `const` variables
- `static` data members of integral or enumeration types
- casts to integral types
- `sizeof` expressions, where the operand is not a variable length array

The `sizeof` operator applied to a variable length array type is evaluated at run time, and therefore is not a constant expression.

You must use an integer constant expression in the following situations:

- 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 enumeration constant.
- In a bit-field width specifier.
- In the preprocessor `#if` statement. (Enumeration constants, address constants, and `sizeof` cannot be specified in a preprocessor `#if` statement.)

Related information

- “The `sizeof` operator” on page 123

Identifier expressions (C++ only)

An identifier expression, or *id-expression*, is a restricted form of primary expression. Syntactically, an *id-expression* requires a higher level of complexity than a simple identifier to provide a name for all of the language elements of C++.

An *id-expression* can be either a qualified or unqualified identifier. It can also appear after the dot and arrow operators.

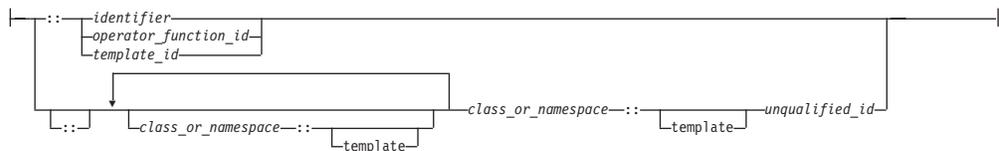
Identifier expression syntax



unqualified_id:



qualified_id:

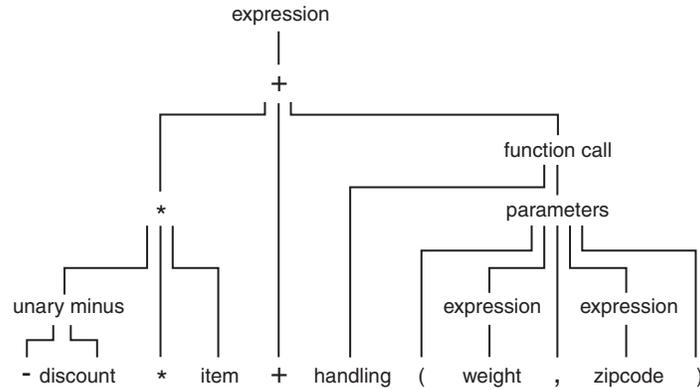


Related information

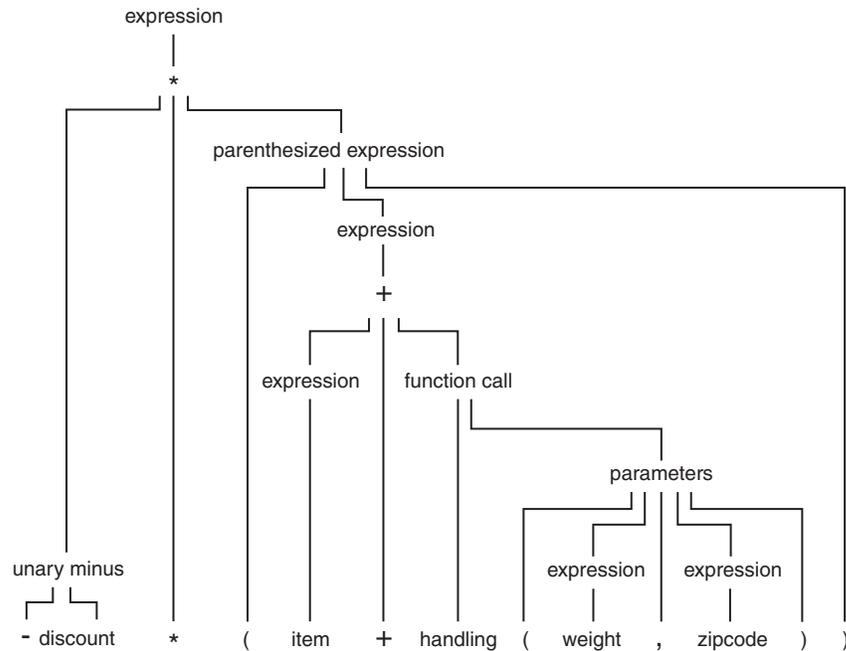
- “Identifiers” on page 15
- Chapter 4, “Declarators,” on page 77

Parenthesized expressions ()

Use parentheses to explicitly force the order of expression evaluation. The following expression does not use parentheses to group operands and operators. The parentheses surrounding `weight`, `zipcode` are used to form a function call. Note how the compiler groups the operands and operators in the expression according to the rules for operator precedence and associativity:



The following expression is similar to the previous expression, but it contains parentheses that change how the operands and operators are grouped:



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);`

Related information

- “Operator precedence and associativity” on page 156

Scope resolution operator :: (C++ only)

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 a namespace scope or global scope name is hidden by an explicit declaration of the same name in a block or class. For example:

```
int count = 0;

int main(void) {
    int count = 0;
    ::count = 1; // set global count to 1
    count = 2;   // set local count to 2
    return 0;
}
```

The declaration of `count` declared in the `main` function hides the integer named `count` declared in global namespace scope. The statement `::count = 1` accesses the variable named `count` declared in global namespace scope.

You can also use the class scope operator to qualify class names or class member names. If a class member name is hidden, you can use it by qualifying it with its class name and the class scope operator.

In the following example, the declaration of the variable `X` hides the class type `X`, but you can still use the static class member `count` by qualifying it with the class type `X` and the scope resolution operator.

```
#include <iostream>
using namespace std;

class X
{
public:
    static int count;
};
int X::count = 10;           // define static data member

int main ()
{
    int X = 0;               // hides class type X
    cout << X::count << endl; // use static member of class X
}
```

Related information

- “Scope of class names (C++ only)” on page 245
- Chapter 9, “Namespaces (C++ only),” on page 217

Function call expressions

A *function call* is an expression containing the function name followed by the function call operator, `()`. If the function has been defined to receive parameters, the values that are to be sent into the function are listed inside the parentheses of the function call operator. The argument list can contain any number of expressions separated by commas. It can also be empty.

The type of a function call expression is the return type of the function. This type can either be a complete type, a reference type, or the type `void`.  A function call expression is always an rvalue.  A function call is an lvalue if and only if the type of the function is a reference.

Here are some examples of the function call operator:

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

The order of evaluation for function call arguments is not specified. In the following example:

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

the argument `batch.process--` might be evaluated last, causing the last two arguments to be passed with the same value.

Related information

- “Function argument conversions” on page 110
- “Function calls” on page 207

Member expressions

Member expressions indicate members of classes, structures, or unions. The member operators are:

- Dot operator `.`
- Arrow operator `->`

Dot operator `.`

The `.` (dot) operator is used to access class, structure, or union members. The member is specified by a postfix expression, followed by a `.` (dot) operator, followed by a possibly qualified identifier or a pseudo-destructor name. (A *pseudo-destructor* is a destructor of a nonclass type.) The postfix 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 postfix expression and the name are lvalues, the expression value is also an lvalue. If the postfix expression is type-qualified, the same type qualifiers will apply to the designated member in the resulting expression.

Related information

- “Access to structure and union members” on page 60
- “Pseudo-destructors (C++ only)” on page 310

Arrow operator `->`

The `->` (arrow) operator is used to access class, structure or union members using a pointer. A postfix expression, followed by an `->` (arrow) operator, followed by a possibly qualified identifier or a pseudo-destructor name, designates a member of the object to which the pointer points. (A *pseudo-destructor* is a destructor of a nonclass type.) The postfix expression must be a pointer to 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 name is an lvalue, the expression value is also an lvalue. If the expression is a pointer to a qualified type, the same type-qualifiers will apply to the designated member in the resulting expression.

Related information

- “Pointers” on page 80
- “Access to structure and union members” on page 60
- Chapter 12, “Class members and friends (C++ only),” on page 251
- “Pseudo-constructors (C++ only)” on page 310

Unary expressions

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

The supported unary operators are:

- Increment operator ++
- Decrement operator --
- Unary plus operator +
- Unary minus operator -
- Logical negation operator !
- Bitwise negation operator ~
- Address operator &
- Indirection operator *
-  typeid
- sizeof
-  typeof
-   digitsof and precisionof
-   __real__ and __imag__

All unary operators have the same precedence and have right-to-left associativity, as shown in Table 28 on page 157.

As indicated in the descriptions of the operators, the usual arithmetic conversions are performed on the operands of most unary expressions.

Related information

- “Pointer arithmetic” on page 81
- “Lvalues and rvalues” on page 111
- “Arithmetic conversions and promotions” on page 103

Increment operator ++

The ++ (increment) operator adds 1 to the value of a scalar operand, or if the operand is a pointer, increments the operand by the size of the object to which it points. The operand receives the result of the increment operation. The operand must be a modifiable lvalue of arithmetic or pointer type.

You can put the ++ before or after the operand. If it appears before the operand, the operand is incremented. The incremented value is then used in the expression. If you put the ++ after the operand, the value of the operand is used in the expression *before* the operand is incremented. For example:

```
play = ++play1 + play2++;
```

is similar to the following expressions; play2 is altered before play:

```

int temp, temp1, temp2;

temp1 = play1 + 1;
temp2 = play2;
play1 = temp1;
temp = temp1 + temp2;
play2 = play2 + 1;
play = temp;

```

The result has the same type as the operand after integral promotion.

The usual arithmetic conversions on the operand are performed.

IBM extension

C The increment operator has been extended to handle complex types. The operator works in the same manner as it does on a real type, except that only the real part of the operand is incremented, and the imaginary part is unchanged.

End of IBM extension

Decrement operator --

The -- (decrement) operator subtracts 1 from the value of a scalar operand, or if the operand is a pointer, decreases the operand by the size of the object to which it points. The operand receives the result of the decrement operation. The operand must be a modifiable lvalue.

You can put the -- before or after the operand. If it appears before the operand, the operand is decremented, and the decremented value is used in the expression. If the -- appears after the operand, the current value of the operand is used in the expression and the operand is decremented.

For example:

```
play = --play1 + play2--;
```

is similar to the following expressions; play2 is altered before play:

```

int temp, temp1, temp2;

temp1 = play1 - 1;
temp2 = play2;
play1 = temp1;
temp = temp1 + temp2;
play2 = play2 - 1;
play = temp;

```

The result has the same type as the operand after integral promotion, but is not an lvalue.

The usual arithmetic conversions are performed on the operand.

IBM extension

C The decrement operator has been extended to handle complex types, for compatibility with GNU C. The operator works in the same manner as it does on a real type, except that only the real part of the operand is decremented, and the

imaginary part is unchanged.

End of IBM extension

Unary plus operator +

The + (unary plus) operator maintains the value of the operand. The operand can have any arithmetic type or pointer type. The result is not an lvalue.

The result has the same type as the operand after integral promotion.

Note: Any plus sign in front of a constant is not part of the constant.

Unary minus operator -

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 after integral promotion.

Note: Any minus sign in front of a constant is not part of the constant.

Logical negation operator !

The ! (logical negation) operator determines whether the operand evaluates to 0 (false) or nonzero (true).

C The expression yields the value 1 (true) if the operand evaluates to 0, and yields the value 0 (false) if the operand evaluates to a nonzero value.

C++ The expression yields the value true if the operand evaluates to false (0), and yields the value false if the operand evaluates to true (nonzero). The operand is implicitly converted to `bool`, and the type of the result is `bool`.

The following two expressions are equivalent:

```
!right;  
right == 0;
```

Related information

- “Boolean types” on page 50

Bitwise negation operator ~

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 `x` represents the decimal value 5. The 16-bit binary representation of `x` is:
0000000000000101

The expression `~x` yields the following result (represented here as a 16-bit binary number):

```
1111111111111010
```

Note that the `~` character can be represented by the trigraph `??~`.

The 16-bit binary representation of `~0` is:

```
1111111111111111
```

IBM extension

C The bitwise negation operator has been extended to handle complex types. With a complex type, the operator computes the complex conjugate of the operand by reversing the sign of the imaginary part.

End of IBM extension

Related information

- “Trigraph sequences” on page 35

Address operator &

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 having 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 and has the same type as the member. The result is not an lvalue.

If `p_to_y` is defined as a pointer to an `int` and `y` as an `int`, the following expression assigns the address of the variable `y` to the pointer `p_to_y`:

```
p_to_y = &y;
```

C++ only

The ampersand symbol `&` is used in C++ as a reference declarator in addition to being the address operator. The meanings are related but not identical.

```
int target;  
int &rTarg = target; // rTarg is a reference to an integer.  
                // The reference is initialized to refer to target.  
void f(int*& p);    // p is a reference to a pointer
```

If you take the address of a reference, it returns the address of its target. Using the previous declarations, `&rTarg` is the same memory address as `&target`.

You may take the address of a register variable.

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.

End of C++ only

Related information

- “Indirection operator `**`” on page 122

- “Pointers” on page 80
- “References (C++ only)” on page 87

Indirection operator *

The * (indirection) operator determines the value referred to by the pointer-type operand. The operand cannot be a pointer to an incomplete type. If the operand points to an object, the operation yields an lvalue referring to that object. If the operand points to a function, the result is a function designator in C or, in C++, an lvalue referring to the object to which the operand points. Arrays and functions are converted 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.

If `p_to_y` is defined as a pointer to an `int` and `y` as an `int`, the expressions:

```
p_to_y = &y;
*p_to_y = 3;
```

cause the variable `y` to receive the value 3.

Related information

- “Arrays” on page 84
- “Pointers” on page 80

The typeid operator (C++ only)

The `typeid` operator provides a program with the ability to retrieve the actual derived type of the object referred to by a pointer or a reference. This operator, along with the `dynamic_cast` operator, are provided for runtime type identification (RTTI) support in C++.

typeid operator syntax

►► typeid (*expr*) ◄◄

└── type-name ─┘

The `typeid` operator returns an lvalue of type `const std::type_info` that represents the type of expression `expr`. You must include the standard template library header `<typeinfo>` to use the `typeid` operator.

If `expr` is a reference or a dereferenced pointer to a polymorphic class, `typeid` will return a `type_info` object that represents the object that the reference or pointer denotes at run time. If it is not a polymorphic class, `typeid` will return a `type_info` object that represents the type of the reference or dereferenced pointer. The following example demonstrates this:

```
#include <iostream>
#include <typeinfo>
using namespace std;

struct A { virtual ~A() { } };
struct B : A { };
```

```

struct C { };
struct D : C { };

int main() {
    B bobj;
    A* ap = &bobj;
    A& ar = bobj;
    cout << "ap: " << typeid(*ap).name() << endl;
    cout << "ar: " << typeid(ar).name() << endl;

    D dobj;
    C* cp = &dobj;
    C& cr = dobj;
    cout << "cp: " << typeid(*cp).name() << endl;
    cout << "cr: " << typeid(cr).name() << endl;
}

```

The following is the output of the above example:

```

ap: B
ar: B
cp: C
cr: C

```

Classes A and B are polymorphic; classes C and D are not. Although `cp` and `cr` refer to an object of type D, `typeid(*cp)` and `typeid(cr)` return objects that represent class C.

Lvalue-to-rvalue, array-to-pointer, and function-to-pointer conversions will not be applied to *expr*. For example, the output of the following example will be `int [10]`, not `int *`:

```

#include <iostream>
#include <typeinfo>
using namespace std;

int main() {
    int myArray[10];
    cout << typeid(myArray).name() << endl;
}

```

If *expr* is a class type, that class must be completely defined.

The `typeid` operator ignores top-level `const` or `volatile` qualifiers.

Related information

- “Type names” on page 79

The sizeof operator

The `sizeof` operator yields the size in bytes of the operand, which can be an expression or the parenthesized name of a type.

sizeof operator syntax

```

▶▶ sizeof expr
           (—type-name—)

```

The result for either kind of operand is not an lvalue, but a constant integer value. The type of the result is the unsigned integral type `size_t` defined in the header file `stddef.h`.

Except in preprocessor directives, you can use a `sizeof` expression wherever an integral constant is required. One of the most common uses 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 can use the `sizeof` operator to determine the size that a data type represents. For example:

```
sizeof(int);
```

The `sizeof` operator applied to a type name yields the amount of memory that would be used by an object of that type, including any internal or trailing padding.

z/OS only

Using the `sizeof` operator with a fixed-point decimal type results in the total number of bytes that are occupied by the decimal type. z/OS XL 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));
```

End of z/OS only

For compound types, results are as follows:

Operand	Result
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, and 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.

The `sizeof` operator may not be applied to:

- A bit field
- A function type
- An undefined structure or class
- An incomplete type (such as `void`)

The `sizeof` operator applied to an expression yields the same result as if it had been applied to only the name of the type of the expression. At compile time, the compiler analyzes the expression to determine its type. None of the usual type conversions that occur in the type analysis of the expression are directly attributable to the `sizeof` operator. However, if the operand contains operators that perform conversions, the compiler does take these conversions into consideration in determining the type. For example, the second line of the following sample causes the usual arithmetic conversions to be performed. Assuming that a `short` uses 2 bytes of storage and an `int` uses 4 bytes,

```
short x; ... sizeof (x)          /* the value of sizeof operator is 2 */
short x; ... sizeof (x + 1)     /* value is 4, result of addition is type int */
```

The result of the expression `x + 1` has type `int` and is equivalent to `sizeof(int)`. The value is also 4 if `x` has type `char`, `short`, or `int` or any enumeration type.

Related information

- “Type names” on page 79
- “Integer constant expressions” on page 113
- “Arrays” on page 84
- “References (C++ only)” on page 87

The typeof operator

IBM extension

The `typeof` operator returns the type of its argument, which can be an expression or a type. The language feature provides a way to derive the type from an expression. Given an expression `e`, `__typeof__(e)` can be used anywhere a type name is needed, for example in a declaration or in a cast. The alternate spelling of the keyword, `__typeof`, is recommended.

typeof operator syntax

```

  >>  [ __typeof__ ] ( [ expr ] )
      [  typeof   ] [ type-name ]

```

A `typeof` construct itself is not an expression, but the name of a type. A `typeof` construct behaves like a type name defined using `typedef`, although the syntax resembles that of `sizeof`.

The following examples illustrate its basic syntax. For an expression `e`:

```
int e;
__typeof__(e + 1) j; /* the same as declaring int j; */
e = (__typeof__(e)) f; /* the same as casting e = (int) f; */
```

Using a `typeof` construct is equivalent to declaring a `typedef` name. Given

```
int T[2];
int i[2];
```

you can write

```
__typeof__(i) a; /* all three constructs have the same meaning */
__typeof__(int[2]) a;
__typeof__(T) a;
```

The behavior of the code is as if you had declared `int a[2];`.

For a bit field, `typeof` represents the underlying type of the bit field. For example, `int m:2;`, the `typeof(m)` is `int`. Since the bit field property is not reserved, `n in typeof(m) n;` is the same as `int n,` but not `int n:2.`

The `typeof` operator can be nested inside `sizeof` and itself. The following declarations of `arr` as an array of pointers to `int` are equivalent:

```
int *arr[10]; /* traditional C declaration */
__typeof__(__typeof__(int *)[10]) a; /* equivalent declaration */
```

The `typeof` operator can be useful in macro definitions where expression `e` is a parameter. For example,

```
#define SWAP(a,b) { __typeof__(a) temp; temp = a; a = b; b = temp; }
```

Notes:

1. The `typeof` and `__typeof__` keywords are supported as follows:
 - The `__typeof__` keyword is recognized in C under `LANGLVL(EXTC89|EXTC99|EXTENDED)`, and in C++ under the `LANGLVL(EXTENDED)`.
 -  The `typeof` keyword is only recognized when the `KEYWORD(TYPEOF)` compiler option is in effect.

Related information

- “Type names” on page 79
- “typedef definitions” on page 65

End of IBM extension

The `digitsof` and `precisionof` operators (C only)

z/OS only

The `digitsof` and `precisionof` operators yield information about fixed-point 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,

$$\begin{aligned} \text{digitsof}(\text{decimal}(n,p)) &= n \\ \text{precisionof}(\text{decimal}(n,p)) &= p \end{aligned}$$

The results of the `digitsof` and `precisionof` operators are integer constants.

Related information

- “Fixed-point decimal literals” on page 25
- “Fixed-point decimal types (C only)” on page 53

End of z/OS only

The `__real__` and `__imag__` operators (C only)

IBM extension

z/OS XL C/C++ extends the C99 standards to support the unary operators `__real__` and `__imag__`. These operators provide the ability to extract the real and imaginary parts of a complex type. These extensions have been implemented to ease the porting applications developed with GNU C.

`__real__` and `__imag__` operator syntax

► `__real__` `__imag__` (`—var_identifier—`) ◀

The `var_identifier` is the name of a previously declared complex variable. The

`__real__` operator returns the real part of the complex variable, while the `__imag__` operator returns the imaginary part of the variable. If the operand of these operators is an lvalue, the resulting expression can be used in any context where lvalues are allowed. They are especially useful in initializations of complex variables, and as arguments to calls to library functions such as `printf` and `scanf` that have no format specifiers for complex types. For example:

```
float _Complex myvar;  
__imag__(myvar) = 2.0f;  
__real__(myvar) = 3.0f;
```

initializes the imaginary part of the complex variable `myvar` to $2.0i$ and the real part to 3.0 , and

```
printf("myvar = %f + %f * i\n", __real__(myvar), __imag__(myvar));
```

prints:

```
myvar = 2.000000 + 3.000000 * i
```

Related information

- “Complex literals (C only)” on page 25
- “Complex floating-point types (C only)” on page 52

End of IBM extension

Binary expressions

A *binary expression* contains two operands separated by one operator. The supported binary operators are:

- “Assignment operators” on page 128
- Multiplication operator `*`
- Division operator `/`
- Remainder operator `%`
- Addition operator `+`
- Subtraction operator `-`
- Bitwise left and right shift operators `<<` `>>`
- Relational operators `<` `>` `<=` `>=`
- Equality and inequality operators `==` `!=`
- Bitwise AND operator `&`
- Bitwise exclusive OR operator `^`
- Bitwise inclusive OR operator `|`
- Logical AND operator `&&`
- Logical OR operator `||`
- Array subscripting operator `[]`
- “Comma operator `,`” on page 139
- Pointer to member operators `.*` `->*` (C++ only)

All binary operators have left-to-right associativity, but not all binary operators have the same precedence. The ranking and precedence rules for binary operators is summarized in Table 29 on page 158.

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 descriptions of the operators, the usual arithmetic conversions are performed on the operands of most binary expressions.

Related information

- “Lvalues and rvalues” on page 111
- “Arithmetic conversions and promotions” on page 103

Assignment operators

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

- Simple assignment operator =
- Compound assignment operators

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.

 The result of an assignment expression is not an lvalue.  The result of an assignment expression is an lvalue.

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

Related information

- “Lvalues and rvalues” on page 111
- “Pointers” on page 80
- “Type qualifiers” on page 67

Simple assignment operator =

The simple assignment operator has the following form:

lvalue = *expr*

The operator stores the value of the right operand *expr* in the object designated by the left operand *lvalue*.

The left operand must be a modifiable lvalue. The type of an assignment operation is the type of the left operand.

If the left operand is not a class type, the right operand is implicitly converted to the type of the left operand. This converted type will not be qualified by `const` or `volatile`.

If the left operand is a class type, that type must be complete. The copy assignment operator of the left operand will be called.

If the left operand is an object of reference type, the compiler will assign the value of the right operand to the object denoted by the reference.

z/OS only

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, z/OS XL 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 “Initialization of pointers” on page 94 for more information on assignments with pointers.

End of z/OS only

Compound assignment operators

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 store the result of that operation into the left operand, which must be a modifiable lvalue.

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><<=</code> , <code>>>=</code> , <code>&=</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 using 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>

Operator	Example	Equivalent expression
/=	time /= hours	time = time / hours
%=	allowance %= 1000	allowance = allowance % 1000
<<=	result <<= num	result = result << num
>>=	form >>= 1	form = form >> 1
&=	mask &= 2	mask = mask & 2
^=	test ^= pre_test	test = test ^ pre_test
=	flag = 0N	flag = flag 0N

Although the equivalent expression column shows the left operands (from the example column) twice, it is in effect evaluated only once.

C++ In addition to the table of operand types, an expression is implicitly converted to the cv-unqualified type of the left operand if it is not of class type. However, if the left operand is of class type, the class becomes complete, and assignment to objects of the class behaves as a copy assignment operation. Compound expressions and conditional expressions are lvalues in C++, which allows them to be a left operand in a compound assignment expression.

Multiplication operator *

The * (multiplication) operator yields the product of its operands. The operands must have an arithmetic or enumeration type. The result is not an lvalue. The usual arithmetic conversions on the operands are performed.

Because the multiplication operator has both associative and commutative properties, the compiler can rearrange the operands in an expression that contains more than one multiplication operator. For example, the expression:

```
sites * number * cost
```

can be interpreted in any of the following ways:

```
(sites * number) * cost
sites * (number * cost)
(cost * sites) * number
```

Division operator /

The / (division) operator yields the algebraic quotient of its operands. If both operands are integers, any fractional part (remainder) is discarded. Throwing away the fractional part is often called *truncation toward zero*. The operands must have an arithmetic or enumeration type. The right operand may not be zero: the result is undefined if the right operand evaluates to 0. For example, expression `7 / 4` yields the value 1 (rather than 1.75 or 2). The result is not an lvalue.

If either operand is negative, the result is rounded towards zero.

The usual arithmetic conversions on the operands are performed.

Remainder operator %

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 or enumeration 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 usual arithmetic conversions on the operands are performed.

Addition operator +

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 or enumeration type.

When both operands have an arithmetic type, the usual arithmetic conversions on the operands are performed. The result has the type produced by the conversions on the operands and is not an lvalue.

A pointer to an object in an array can be added to a value having integral type. The result is a pointer of the same type as the pointer operand. The result refers to another element in the array, offset from the original element by the amount of the integral value treated as a subscript. If the resulting pointer points to storage outside the array, other than the first location outside the array, the result is undefined. A pointer to one element past the end of an array cannot be used to access the memory content at that address. The compiler does not provide boundary checking on the pointers. For example, after the addition, `ptr` points to the third element of the array:

```
int array[5];
int *ptr;
ptr = array + 2;
```

Related information

- “Pointer arithmetic” on page 81
- “Pointer conversions” on page 107

Subtraction operator -

The - (subtraction) operator yields the difference of its operands. Both operands must have an arithmetic or enumeration type, or the left operand must have a pointer type and the right operand must have the same pointer type or an integral or enumeration type. You cannot subtract a pointer from an integral value.

When both operands have an arithmetic type, the usual arithmetic conversions on the operands are performed. The result has the type produced by the conversions on the operands and is not an lvalue.

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 elements in the same array, the result is the number of objects separating the two addresses. The number is of type `ptrdiff_t`, which is defined in the header file `stddef.h`. Behavior is undefined if the pointers do not refer to objects in the same array.

Related information

- “Pointer arithmetic” on page 81
- “Pointer conversions” on page 107

Bitwise left and right shift operators << >>

The bitwise shift operators move the bit values of a binary object. The left operand specifies the value to be shifted. The right operand specifies the number of positions that the bits in the value are to be shifted. The result is not an lvalue. Both operands have the same precedence and are left-to-right associative.

Operator	Usage
<<	Indicates the bits are to be shifted to the left.
>>	Indicates the bits are to be shifted to the right.

Each operand must have an integral or enumeration type. The compiler performs integral promotions on the operands, and then the right operand is converted to type `int`. The result has the same type as the left operand (after the arithmetic conversions).

The right operand should not have a negative value or a value that is greater than or equal to the width in bits of the expression being shifted. The result of bitwise shifts on such values is unpredictable.

If the right operand has the value 0, the result is the value of the left operand (after the usual arithmetic conversions).

The << operator fills vacated bits with zeros. For example, if `left_op` has the value 4019, the bit pattern (in 16-bit format) of `left_op` is:

0000111110110011

The expression `left_op << 3` yields:

0111110110011000

The expression `left_op >> 3` yields:

000000011110110

Relational operators < > <= >=

The relational operators compare two operands and determine the validity of a relationship. The following table describes the four 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.

Operator	Usage
<=	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 or enumeration types or be pointers to the same type.

C The type of the result is `int` and has the values 1 if the specified relationship is true, and 0 if false. **C++** The type of the result is `bool` and has the values `true` or `false`.

The result is not an lvalue.

If the operands have arithmetic types, the usual arithmetic conversions on the operands are performed.

When the operands are pointers, the result is determined by the locations of the objects to which the pointers refer. If the pointers do not refer to objects in the same array, the result is not defined.

A pointer can be compared to a constant expression that evaluates to 0. You can also compare a pointer to a pointer of type `void*`. The pointer is converted to a pointer of type `void*`.

If two pointers refer to the same object, they are considered equal. If two pointers refer to nonstatic members of the same object, the pointer to the object declared later is greater, provided that they are not separated by an access specifier; otherwise the comparison is undefined. 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 is greater.

You can only compare members of the same object with relational operators.

Relational operators have left-to-right associativity. For example, the expression:

```
a < b <= c
```

is interpreted as:

```
(a < b) <= c
```

If the value of `a` is less than the value of `b`, the first relationship yields 1 in C, or `true` in C++. The compiler then compares the value `true` (or 1) with the value of `c` (integral promotions are carried out if needed).

Equality and inequality operators `==` `!=`

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. The following table describes the two equality

operators:

Operator	Usage
==	Indicates whether the value of the left operand is equal to the value of the right operand.
!=	Indicates whether the value of the left operand is not equal to the value of the right operand.

Both operands must have arithmetic or enumeration 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 a null pointer.

 The type of the result is `int` and has the values 1 if the specified relationship is true, and 0 if false.  The type of the result is `bool` and has the values `true` or `false`.

If the operands have arithmetic types, the usual arithmetic conversions on the operands are performed.

If the operands are pointers, the result is determined by the locations of the objects to which the pointers refer.

If one operand is a pointer and the other operand is an integer having the value 0, the `==` expression is true only if the pointer operand evaluates to `NULL`. The `!=` operator evaluates to true if the pointer operand does not evaluate to `NULL`.

You can also use the equality operators to compare pointers to members that are of the same type but do not belong to the same object. The following expressions contain examples of equality and relational operators:

```
time < max_time == status < complete
letter != EOF
```

Note: The equality operator (`==`) should not be confused with the assignment (`=`) operator.

For example,

```
if (x == 3)
```

evaluates to true (or 1) if `x` is equal to three. Equality tests like this should be coded with spaces between the operator and the operands to prevent unintentional assignments.

```
while
```

```
if (x = 3)
```

is taken to be true because `(x = 3)` evaluates to a nonzero value (3). The expression also assigns the value 3 to `x`.

Related information

- “Simple assignment operator `=`” on page 128

Bitwise AND operator &

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, the corresponding bit of the result is set to 1. Otherwise, it sets the corresponding result bit to 0.

Both operands must have an integral or enumeration type. The usual arithmetic conversions on each operand are performed. 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	000000001011100
bit pattern of b	000000000101110
bit pattern of a & b	000000000001100

Note: The bitwise AND (&) should not be confused with the logical AND. (&&) operator. For example,

```
1 & 4 evaluates to 0
while
1 && 4 evaluates to true
```

Bitwise exclusive OR operator ^

The bitwise exclusive OR operator (in EBCDIC, the ^ symbol is represented by the \neg symbol) compares each bit of its first operand to the corresponding bit of the second operand. If both bits are 1's or both bits are 0's, the corresponding bit of the result is set to 0. Otherwise, it sets the corresponding result bit to 1.

Both operands must have an integral or enumeration type. The usual arithmetic conversions on each operand are performed. The result has the same type as the converted operands and is not an lvalue.

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 the ^ character can be represented 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 a	000000001011100
bit pattern of b	000000000101110
bit pattern of a ^ b	000000001110010

Related information

- “Trigraph sequences” on page 35

Bitwise inclusive OR operator |

The `|` (bitwise inclusive OR) operator compares the values (in binary format) of each operand and yields a value whose bit pattern shows which bits in either of the operands has the value 1. If both of the bits are 0, the result of that bit is 0; otherwise, the result is 1.

Both operands must have an integral or enumeration type. The usual arithmetic conversions on each operand are performed. The result has the same type as the converted operands and is not an lvalue.

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 the `|` character can be represented 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>	000000001011100
bit pattern of <code>b</code>	000000000101110
bit pattern of <code>a b</code>	000000001111110

Note: The bitwise OR (`|`) should not be confused with the logical OR (`||`) operator. For example,

```
1 | 4 evaluates to 5
while
1 || 4 evaluates to true
```

Related information

- “Trigraph sequences” on page 35

Logical AND operator &&

The `&&` (logical AND) operator indicates whether both operands are true.

C If both operands have nonzero values, the result has the value 1. Otherwise, the result has the value 0. The type of the result is `int`. Both operands must have a arithmetic or pointer type. The usual arithmetic conversions on each operand are performed.

C++ If both operands have values of `true`, the result has the value `true`. Otherwise, the result has the value `false`. Both operands are implicitly converted to `bool` and the result type is `bool`.

Unlike the `&` (bitwise AND) operator, the `&&` operator guarantees left-to-right evaluation of the operands. If the left operand evaluates to 0 (or `false`), the right operand is not evaluated.

The following examples show how the expressions that contain the logical AND operator are evaluated:

Expression	Result
1 && 0	false or 0
1 && 4	true or 1
0 && 0	false or 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 (or false).

Note: The logical AND (&&) should not be confused with the bitwise AND (&) operator. For example:

```

1 && 4 evaluates to 1 (or true)
while
1 & 4 evaluates to 0

```

Logical OR operator ||

The `||` (logical OR) operator indicates whether either operand is true.

C If either of the operands has a nonzero value, the result has the value 1. Otherwise, the result has the value 0. The type of the result is `int`. Both operands must have a arithmetic or pointer type. The usual arithmetic conversions on each operand are performed.

C++ If either operand has a value of `true`, the result has the value `true`. Otherwise, the result has the value `false`. Both operands are implicitly converted to `bool` and the result type is `bool`.

Unlike the `|` (bitwise inclusive OR) operator, the `||` operator guarantees left-to-right evaluation of the operands. If the left operand has a nonzero (or true) value, the right operand is not evaluated.

The following examples show how expressions that contain the logical OR operator are evaluated:

Expression	Result
1 0	true or 1
1 4	true or 1
0 0	false or 0

The following example uses the logical OR operator to conditionally increment `y`:
`++x || ++y;`

The expression `++y` is not evaluated when the expression `++x` evaluates to a nonzero (or true) quantity.

Note: The logical OR (||) should not be confused with the bitwise OR (|) operator.
For example:

```
1 || 4 evaluates to 1 (or true)
while
1 | 4 evaluates to 5
```

Array subscripting operator []

A postfix expression followed by an expression in [] (brackets) specifies an element of an array. The expression within the brackets is referred to as a *subscript*. The first element of an array has the subscript zero.

By definition, the expression `a[b]` is equivalent to the expression `*((a) + (b))`, and, because addition is associative, it is also equivalent to `b[a]`. Between expressions `a` and `b`, one must be a pointer to a type `T`, and the other must have integral or enumeration type. The result of an array subscript is an lvalue. The following example demonstrates this:

```
#include <stdio.h>

int main(void) {
    int a[3] = { 10, 20, 30 };
    printf("a[0] = %d\n", a[0]);
    printf("a[1] = %d\n", 1[a]);
    printf("a[2] = %d\n", *(2 + a));
    return 0;
}
```

The following is the output of the above example:

```
a[0] = 10
a[1] = 20
a[2] = 30
```

 The above restrictions on the types of expressions required by the subscript operator, as well as the relationship between the subscript operator and pointer arithmetic, do not apply if you overload operator[] of a class.

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 right-most 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 < 4; ++first)
{
    for (second = 0; second < 3; ++second)
    {
        for (third = 0; third < 6; ++third)
        {
            code[first][second][third] =
            100;
        }
    }
}
```

C only

C99 allows array subscripting on arrays that are not lvalues. However, using the address of a non-lvalue as an array subscript is still not allowed. The following example is valid in C99:

```
struct trio{int a[3];};
struct trio f();
foo (int index)
{
    return f().a[index];
}
```

End of C only

Related information

- “Pointers” on page 80
- “Integral types” on page 49
- “Lvalues and rvalues” on page 111
- “Arrays” on page 84
- “Overloading subscripting (C++ only)” on page 234
- “Pointer arithmetic” on page 81

Comma operator ,

A *comma expression* contains two operands of any type separated by a comma and has left-to-right associativity. The left operand is fully evaluated, possibly producing side effects, and its value, if there is one, is discarded. The right operand is then evaluated. The type and value of the result of a comma expression are those of its right operand, after the usual unary conversions.

C only

The result of a comma expression is not an lvalue.

End of C only

C++ only

In C++, the result is an lvalue if the right operand is an lvalue. The following statements are equivalent:

```
r = (a,b,...,c);
a; b; r = c;
```

The difference is that the comma operator may be suitable for expression contexts, such as loop control expressions.

Similarly, the address of a compound expression can be taken if the right operand is an lvalue.

```
&(a, b)
a, &b
```

End of C++ only

Any number of expressions separated by commas can form a single expression because the comma operator is associative. The use of the comma operator guarantees that the subexpressions will be evaluated in left-to-right order, and the value of the last becomes the value of the entire expression. 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);
```

A sequence point occurs after the evaluation of the first operand. The value of `delta` is discarded. Similarly, in the following example, the value of the expression: `intensity++, shade * increment, rotate(direction);`

is the value of the expression:

```
rotate(direction)
```

In some contexts where the comma character is used, parentheses are required to avoid ambiguity. For example, the function

```
f(a, (t = 3, t + 2), c);
```

has only three arguments: the value of `a`, the value 5, and the value of `c`. Other contexts in which parentheses are required are in field-length expressions in structure and union declarator lists, enumeration value expressions in enumeration declarator lists, and initialization expressions in declarations and initializers.

In the previous example, the comma is used to separate the argument expressions in a function invocation. In this context, its use does not guarantee the order of evaluation (left to right) of the function arguments.

The primary use of the comma operator is to produce side effects in the following situations:

- Calling a function
- Entering or repeating an iteration loop
- Testing a condition
- Other situations where a side effect is required but the result of the expression is not immediately needed

The following table gives some examples of the uses of the comma operator.

Statement	Effects
<code>for (i=0; i<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>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>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.

Pointer to member operators `.*` `->*` (C++ only)

There are two pointer to member operators: `.*` and `->*`.

The `.*` operator is used to dereference pointers to class members. The first operand must be of 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`.

The `->*` operator is also used to 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, resulting 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.

Related information

- “Class member lists (C++ only)” on page 251
- “Pointers to members (C++ only)” on page 256

Conditional expressions

A *conditional expression* is a compound expression that contains a condition that is implicitly converted to type `bool` in C++(`operand1`), an expression to be evaluated if the condition evaluates to true (`operand2`), and an expression to be evaluated if the condition has the value false (`operand3`).

The conditional expression contains one two-part operator. The `?` symbol follows the condition, and the `:` symbol appears between the two action expressions. All expressions that occur between the `?` and `:` are treated as one expression.

The first operand must have a scalar type. The type of the second and third operands must be one of the following:

- An arithmetic type
- A compatible pointer, structure, or union type
- void

The second and third operands can also be a pointer or a null pointer constant.

Two 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.

The first operand is evaluated, and its value determines whether the second or third operand is evaluated:

- If the value is true, the second operand is evaluated.
- If the value is false, the third operand is evaluated.

The result is the value of the second or third operand.

If the second and third expressions evaluate to arithmetic types, the usual arithmetic conversions are performed on the values. The types of the second and third operands determine the type of the result as shown in the following tables.

Conditional expressions have right-to-left associativity with respect to their first and third operands. The leftmost operand is evaluated first, and then only one of the remaining two operands is evaluated. The following expressions are equivalent:

```
a ? b : c ? d : e ? f : g
a ? b : (c ? d : (e ? f : g))
```

Types in conditional C expressions

C only

In C, a conditional expression is not an lvalue, nor is its result.

Table 25. Types of operands and results in 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

End of C only

Types in conditional C++ expressions

C++ only

In C++, a conditional expression is a valid lvalue if its type is not void, and its result is an lvalue.

Table 26. Types of operands and results in C++ conditional 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;
```

The following is an equivalent statement:

```
if (y > z)
    x = y;
else
    x = z;
```

The following expression calls the function `printf`, which receives the value of the variable *c*, if *c* evaluates to a digit. Otherwise, `printf` receives the character constant 'x'.

```
printf(" c = %c\n", isdigit(c) ? c : 'x');
```

If the last operand of a conditional expression contains an assignment operator, use parentheses to ensure the expression evaluates properly. For example, the `=` operator has higher precedence than the `?:` operator in the following expression:

```
int i,j,k;
(i == 7) ? j ++ : k = j;
```

The compiler will interpret this expression 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`.

To assign the value of *j* to *k* when `i == 7` is false, enclose the last operand in parentheses:

```
int i,j,k;
(i == 7) ? j ++ : (k = j);
```

Cast expressions

A cast operator is used for *explicit type conversions*. It converts the value of an expression to a specified type.

The following cast operators are supported:

- “Cast operator `()`” on page 144
- “The `static_cast` operator (C++ only)” on page 145
- “The `reinterpret_cast` operator (C++ only)” on page 146
- “The `const_cast` operator (C++ only)” on page 148
- “The `dynamic_cast` operator (C++ only)” on page 149

Cast operator ()

Cast expression syntax

▶—(*type*)—*expression*—▶

C The result of this operation is not an lvalue. **C++** The result of this operation is an lvalue if *type* is a reference; in all other cases, the result is an rvalue.

The following demonstrates the use of the cast operator to dynamically create an integer array of size 10:

```
#include <stdlib.h>

int main(void) {
    int* myArray = (int*) malloc(10 * sizeof(int));
    free(myArray);
    return 0;
}
```

The `malloc` library function returns a `void` pointer that points to memory that will hold an object of the size of its argument. The statement `int* myArray = (int*) malloc(10 * sizeof(int))` does the following:

- Creates a `void` pointer that points to memory that can hold ten integers.
- Converts that `void` pointer into an integer pointer with the use of the cast operator.
- Assigns that integer pointer to `myArray`. Because a name of an array is the same as a pointer to the initial element of the array, `myArray` is an array of ten integers stored in the memory created by the call to `malloc()`.

C++ only

In C++ you can also use the following in cast expressions:

- Function-style casts
- C++ conversion operators, such as `static_cast`.

Function-style notation converts the value of *expression* to the type *type*:

expression(*type*)

The following example shows the same value cast with a C-style cast, the C++ function-style cast, and a C++ cast operator:

```
#include <iostream>
using namespace std;

int main() {
    float num = 98.76;
    int x1 = (int) num;
    int x2 = int(num);
    int x3 = static_cast<int>(num);

    cout << "x1 = " << x1 << endl;
    cout << "x2 = " << x2 << endl;
    cout << "x3 = " << x3 << endl;
}
```

The following is the output of the above example:

```
x1 = 98
x2 = 98
x3 = 98
```

The integer `x1` is assigned a value in which `num` has been explicitly converted to an `int` with the C-style cast. The integer `x2` is assigned a value that has been converted with the function-style cast. The integer `x3` is assigned a value that has been converted with the `static_cast` operator.

A cast is a valid lvalue if its operand is an lvalue. In the following simple assignment expression, the right-hand side is first converted to the specified type, then to the type of the inner left-hand side expression, and the result is stored. The value is converted back to the specified type, and becomes the value of the assignment. In the following example, `i` is of type `char *`.

```
(int)i = 8 // This is equivalent to the following expression
(int)(i = (char*) (int)(8))
```

For compound assignment operation applied to a cast, the arithmetic operator of the compound assignment is performed using the type resulting from the cast, and then proceeds as in the case of simple assignment. The following expressions are equivalent. Again, `i` is of type `char *`.

```
(int)i += 8 // This is equivalent to the following expression
(int)(i = (char*) (int)((int)i = 8))
```

For C++, the operand of a cast expression can have class type. If the operand has class type, it can be cast to any type for which the class has a user-defined conversion function. Casts 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.

End of C++ only

Related information

- “Type names” on page 79
- “Conversion functions (C++ only)” on page 314
- “Conversion constructors (C++ only)” on page 312
- “Lvalues and rvalues” on page 111

The `static_cast` operator (C++ only)

The *static_cast operator* converts a given expression to a specified type.

static_cast operator syntax

```
► static_cast<Type>(—expression—) ◀
```

The following is an example of the `static_cast` operator.

```
#include <iostream>
using namespace std;

int main() {
    int j = 41;
    int v = 4;
    float m = j/v;
```

```

float d = static_cast<float>(j)/v;
cout << "m = " << m << endl;
cout << "d = " << d << endl;
}

```

The following is the output of the above example:

```

m = 10
d = 10.25

```

In this example, `m = j/v;` produces an answer of type `int` because both `j` and `v` are integers. Conversely, `d = static_cast<float>(j)/v;` produces an answer of type `float`. The `static_cast` operator converts variable `j` to a type `float`. This allows the compiler to generate a division with an answer of type `float`. All `static_cast` operators resolve at compile time and do not remove any `const` or `volatile` modifiers.

Applying the `static_cast` operator to a null pointer will convert it to a null pointer value of the target type.

You can explicitly convert a pointer of a type `A` to a pointer of a type `B` if `A` is a base class of `B`. If `A` is not a base class of `B`, a compiler error will result.

You may cast an lvalue of a type `A` to a type `B` if the following are true:

- `A` is a base class of `B`
- You are able to convert a pointer of type `A` to a pointer of type `B`
- The type `B` has the same or greater `const` or `volatile` qualifiers than type `A`
- `A` is not a virtual base class of `B`

The result is an lvalue of type `B`.

A pointer to member type can be explicitly converted into a different pointer to member type if both types are pointers to members of the same class. This form of explicit conversion may also take place if the pointer to member types are from separate classes, however one of the class types must be derived from the other.

Related information

- “User-defined conversions (C++ only)” on page 311
- “Type-based aliasing” on page 82

The `reinterpret_cast` operator (C++ only)

A *`reinterpret_cast` operator* handles conversions between unrelated types.

`reinterpret_cast` operator syntax

►► `reinterpret_cast` ◀◀ *Type* ▶▶ (*expression*) ◀◀

The `reinterpret_cast` operator produces a value of a new type that has the same bit pattern as its argument. You cannot cast away a `const` or `volatile` qualification. You can explicitly perform the following conversions:

- A pointer to any integral type large enough to hold it
- A value of integral or enumeration type to a pointer
- A pointer to a function to a pointer to a function of a different type
- A pointer to an object to a pointer to an object of a different type

- A pointer to a member to a pointer to a member of a different class or type, if the types of the members are both function types or object types

A null pointer value is converted to the null pointer value of the destination type.

Given an lvalue expression of type T and an object x, the following two conversions are synonymous:

- `reinterpret_cast<T&>(x)`
- `*reinterpret_cast<T*>(&x)`

C++ also supports C-style casts. The two styles of explicit casts have different syntax but the same semantics, and either way of reinterpreting one type of pointer as an incompatible type of pointer is usually invalid. The `reinterpret_cast` operator, as well as the other named cast operators, is more easily spotted than C-style casts, and highlights the paradox of a strongly typed language that allows explicit casts.

The C++ compiler detects and quietly fixes most but not all violations. It is important to remember that even though a program compiles, its source code may not be completely correct. On some platforms, performance optimizations are predicated on strict adherence to standard aliasing rules. Although the C++ compiler tries to help with type-based aliasing violations, it cannot detect all possible cases.

The following example violates the aliasing rule, but will execute as expected when compiled unoptimized in C++ or in K&R C or with `NOANSIALIAS`. It will also successfully compile optimized in C++ with `ANSIALIAS`, but will not necessarily execute as expected. The offending line 7 causes an old or uninitialized value for x to be printed.

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

The next code example contains an incorrect cast that the compiler cannot even detect because the cast is across two different files.

```
1 /* separately compiled file 1 */
2     extern float f;
3     extern int * int_pointer_to_f = (int *) &f; /* suspicious cast */
4
5 /* separately compiled file 2 */
6     extern float f;
7     extern int * int_pointer_to_f;
8     f = 1.0;
9     int i = *int_pointer_to_f;           /* no suspicious cast but wrong */
```

In line 8, there is no way for the compiler to know that `f = 1.0` is storing into the same object that `int i = *int_pointer_to_f` is loading from.

Related information

- “User-defined conversions (C++ only)” on page 311
- “Type-based aliasing” on page 82

The `const_cast` operator (C++ only)

A *const_cast operator* is used to add or remove a `const` or `volatile` modifier to or from a type.

`const_cast` operator syntax

►► `const_cast` ◀◀ *Type* ◀◀ (*expression*) ◀◀ ◀◀

Type and the type of *expression* may only differ with respect to their `const` and `volatile` qualifiers. Their cast is resolved at compile time. A single `const_cast` expression may add or remove any number of `const` or `volatile` modifiers.

The result of a `const_cast` expression is an rvalue unless *Type* is a reference type. In this case, the result is an lvalue.

Types can not be defined within `const_cast`.

The following demonstrates the use of the `const_cast` operator:

```
#include <iostream>
using namespace std;

void f(int* p) {
    cout << *p << endl;
}

int main(void) {
    const int a = 10;
    const int* b = &a;

    // Function f() expects int*, not const int*
    // f(b);
    int* c = const_cast<int*>(b);
    f(c);

    // Lvalue is const
    // *b = 20;

    // Undefined behavior
    // *c = 30;

    int a1 = 40;
    const int* b1 = &a1;
    int* c1 = const_cast<int*>(b1);

    // Integer a1, the object referred to by c1, has
    // not been declared const
    *c1 = 50;

    return 0;
}
```

The compiler will not allow the function call `f(b)`. Function `f()` expects a pointer to an `int`, not a `const int`. The statement `int* c = const_cast<int*>(b)` returns a pointer `c` that refers to `a` without the `const` qualification of `a`. This process of using `const_cast` to remove the `const` qualification of an object is called *casting away constness*. Consequently the compiler will allow the function call `f(c)`.

The compiler would not allow the assignment `*b = 20` because `b` points to an object of type `const int`. The compiler will allow the `*c = 30`, but the behavior of this

statement is undefined. If you cast away the constness of an object that has been explicitly declared as `const`, and attempt to modify it, the results are undefined.

However, if you cast away the constness of an object that has not been explicitly declared as `const`, you can modify it safely. In the above example, the object referred to by `b1` has not been declared `const`, but you cannot modify this object through `b1`. You may cast away the constness of `b1` and modify the value to which it refers.

Related information

- “Type qualifiers” on page 67
- “Type-based aliasing” on page 82

The `dynamic_cast` operator (C++ only)

The *dynamic_cast* operator performs type conversions at run time. The `dynamic_cast` operator guarantees the conversion of a pointer to a base class to a pointer to a derived class, or the conversion of an lvalue referring to a base class to a reference to a derived class. A program can thereby use a class hierarchy safely. This operator and the `typeid` operator provide runtime type information (RTTI) support in C++.

The expression `dynamic_cast<T>(v)` converts the expression `v` to type `T`. Type `T` must be a pointer or reference to a complete class type or a pointer to void. If `T` is a pointer and the `dynamic_cast` operator fails, the operator returns a null pointer of type `T`. If `T` is a reference and the `dynamic_cast` operator fails, the operator throws the exception `std::bad_cast`. You can find this class in the standard library header `<typeinfo>`.

If `T` is a void pointer, then `dynamic_cast` will return the starting address of the object pointed to by `v`. The following example demonstrates this:

```
#include <iostream>
using namespace std;

struct A {
    virtual ~A() { };
};

struct B : A { };

int main() {
    B bobj;
    A* ap = &bobj;
    void* vp = dynamic_cast<void*>(ap);
    cout << "Address of vp : " << vp << endl;
    cout << "Address of bobj: " << &bobj << endl;
}
```

The output of this example will be similar to the following. Both `vp` and `&bobj` will refer to the same address:

```
Address of vp : 12FF6C
Address of bobj: 12FF6C
```

The primary purpose for the `dynamic_cast` operator is to perform type-safe *downcasts*. A downcast is the conversion of a pointer or reference to a class `A` to pointer or reference to a class `B`, where class `A` is a base class of `B`. The problem with downcasts is that a pointer of type `A*` can and must point to any object of a class that has been derived from `A`. The `dynamic_cast` operator ensures that if you

convert a pointer of class A to a pointer of a class B, the object that A points to belongs to class B or a class derived from B.

The following example demonstrates the use of the `dynamic_cast` operator:

```
#include <iostream>
using namespace std;

struct A {
    virtual void f() { cout << "Class A" << endl; }
};

struct B : A {
    virtual void f() { cout << "Class B" << endl; }
};

struct C : A {
    virtual void f() { cout << "Class C" << endl; }
};

void f(A* arg) {
    B* bp = dynamic_cast<B*>(arg);
    C* cp = dynamic_cast<C*>(arg);

    if (bp)
        bp->f();
    else if (cp)
        cp->f();
    else
        arg->f();
};

int main() {
    A aobj;
    C cobj;
    A* ap = &cobj;
    A* ap2 = &aobj;
    f(ap);
    f(ap2);
}
```

The following is the output of the above example:

```
Class C
Class A
```

The function `f()` determines whether the pointer `arg` points to an object of type A, B, or C. The function does this by trying to convert `arg` to a pointer of type B, then to a pointer of type C, with the `dynamic_cast` operator. If the `dynamic_cast` operator succeeds, it returns a pointer that points to the object denoted by `arg`. If `dynamic_cast` fails, it returns `0`.

You may perform downcasts with the `dynamic_cast` operator only on polymorphic classes. In the above example, all the classes are polymorphic because class A has a virtual function. The `dynamic_cast` operator uses the runtime type information generated from polymorphic classes.

Related information

- “Derivation (C++ only)” on page 275
- “User-defined conversions (C++ only)” on page 311
- “Type-based aliasing” on page 82

Compound literal expressions (C only)

A *compound literal* is a postfix expression that provides an unnamed object whose value is given by an initializer list. The C99 language feature allows you to pass parameters to functions without the need for temporary variables. It is useful for specifying constants of an aggregate type (arrays, structures, and unions) when only one instance of such types is needed.

The syntax for a compound literal resembles that of a cast expression. However, a compound literal is an lvalue, while the result of a cast expression is not. Furthermore, a cast can only convert to scalar types or void, whereas a compound literal results in an object of the specified type.

Compound literal syntax

►► (—*type_name*—) { —*initializer_list*— } ◀◀

The *type_name* can be any data type, and user-defined types. It can be an array of unknown size, but not a variable length array. If the type is an array of unknown size, the size is determined by the initializer list.

The following example passes a constant structure variable of type point containing two integer members to the function drawline:

```
drawline((struct point){6,7});
```

If the compound literal occurs outside the body of a function, the initializer list must consist of constant expressions, and the unnamed object has static storage duration. If the compound literal occurs within the body of a function, the initializer list need not consist of constant expressions, and the unnamed object has automatic storage duration.

Related information

- “String literals” on page 27

new expressions (C++ only)

The new operator provides dynamic storage allocation.

new operator syntax

►► [::] new [(—*argument_list*—)] [(—*type*—)] ◀◀

►► [([—*initial_value*—])] ◀◀

If you prefix new with the scope resolution operator (::), the global operator new() is used. If you specify an *argument_list*, the overloaded new operator that corresponds to that *argument_list* is used. The *type* is an existing built-in or user-defined type. A *new_type* is a type that has not already been defined and can include type specifiers and declarators.

An allocation expression containing the new operator is used to find storage in free store for the object being created. The *new expression* returns a pointer to the object created and can be used to initialize the object. If the object is an array, a pointer to the initial element is returned.

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 object being created 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 an existing *type* is used. You can create an array with zero bounds with the new operator. For example:

```
char * c = new char[0];
```

In this case, a pointer to a unique object is returned.

An object created with operator new() or operator new[] () exists until the operator delete() or operator delete[] () is called to deallocate the object's memory. A delete operator or a destructor will not be implicitly called for an object created with a new that has not been explicitly deallocated before the end of the program.

If parentheses are used within a new type, parentheses should also surround the new type to prevent syntax errors.

In the following example, storage is allocated for an array of pointers to functions:

```
void f();
void g();

int main(void)
{
    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()
    return (0);
}
```

However, the second use of new causes an erroneous binding of q = (new void) (*[3])().

The type of the object being created cannot contain class declarations, enumeration declarations, or const or volatile types. It can contain pointers to const or volatile objects.

For example, const char* is allowed, but char* const is not.

Related information

- “Allocation and deallocation functions (C++ only)” on page 210

Placement syntax

Arguments specifying an allocated storage location can be supplied to `new` by using the *argument_list*, also called the *placement syntax*. If placement arguments are used, a declaration of operator `new()` or operator `new[]()` with these arguments must exist. For example:

```
#include <new>
using namespace std;

class X
{
public:
    void* operator new(size_t,int, int){ /* ... */ }
};

// ...

int main ()
{
    X* ptr = new(1,2) X;
}
```

The placement syntax is commonly used to invoke the global placement `new` function. The global placement `new` function initializes an object or objects at the location specified by the placement argument in the placement `new` expression. This location must address storage that has previously been allocated by some other means, because the global placement `new` function does not itself allocate memory. In the following example, no new memory is allocated by the calls `new(whole) X(8);`, `new(seg2) X(9);`, or `new(seg3) X(10);`. Instead, the constructors `X(8)`, `X(9)`, and `X(10)` are called to reinitialize the memory allocated to the buffer `whole`.

Because placement `new` does not allocate memory, you should not use `delete` to deallocate objects created with the placement syntax. You can only delete the entire memory pool (`delete whole`). In the example, you can keep the memory buffer but destroy the object stored in it by explicitly calling a destructor.

```
#include <new>
class X
{
public:
    X(int n): id(n){ }
    ~X(){ }
private:
    int id;
    // ...
};

int main()
{
    char* whole = new char[ 3 * sizeof(X) ]; // a 3-part buffer
    X * p1 = new(whole) X(8); // fill the front
    char* seg2 = &whole[ sizeof(X) ]; // mark second segment
    X * p2 = new(seg2) X(9); // fill second segment
    char* seg3 = &whole[ 2 * sizeof(X) ]; // mark third segment
    X * p3 = new(seg3) X(10); // fill third segment

    p2->~X(); // clear only middle segment, but keep the buffer
    // ...
    return 0;
}
```

The placement new syntax can also be used for passing parameters to an allocation routine rather than to a constructor.

Related information

- “delete expressions (C++ only)” on page 155
- “Scope resolution operator :: (C++ only)” on page 116
- “Overview of constructors and destructors (C++ only)” on page 299

Initialization of objects created with the new operator

You can initialize objects 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 does not have a default constructor, the new initializer must be provided when any object of that class is allocated. The arguments of the new initializer must match the arguments of a constructor.

You cannot specify an initializer for arrays. You can initialize an array of class objects only if the class has a default constructor. The constructor is called to initialize each array element (class object).

Initialization using the new initializer is performed only if new successfully allocates storage.

Related information

- “Overview of constructors and destructors (C++ only)” on page 299

Handling new allocation 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 std::set_new_handler() function is declared in the header <new>. Use it to call a new handler you have defined or the default new handler.

Your new handler must perform one of the following:

- obtain more storage for memory allocation, then return
- throw an exception of type std::bad_alloc or a class derived from std::bad_alloc
- call either abort() or exit()

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` throws an exception of type `std::bad_alloc`.

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>
#include <new>
#include <cstdlib>
using namespace std;

void no_storage()
{
    std::cerr << "Operator new failed: no storage is
    available.\n";
    std::exit(1);
}

int main(void)
{
    std::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.
```

delete expressions (C++ only)

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.

delete operator syntax

►►  `delete object_pointer`

The operand of `delete` must be a pointer returned by `new`, and cannot be a pointer to constant. 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`.

delete[] operator syntax

►►  `delete [] array`

The result of deleting an array object with `delete` is undefined, as is deleting an individual object with `delete[]`. The array dimensions do not need to be specified with `delete[]`.

The result of any attempt to access a deleted object or array is undefined.

If a destructor has been defined 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 if:

- The class has no operator `delete()`.
- The object is of a nonclass type.
- The object is deleted with the `::delete` expression.

The global `::operator delete[]()` is used if:

- The class has no operator `delete[]()`
- The object is of a nonclass type
- 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[]()`.

Related information

- “Overview of constructors and destructors (C++ only)” on page 299
- “The void type” on page 54

throw expressions (C++ only)

A *throw* expression is used to throw exceptions to C++ exception handlers. A throw expression is of type `void`.

Related information

- Chapter 16, “Exception handling (C++ only),” on page 353
- “The void type” on page 54

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. An operator’s precedence is meaningful only if other operators with higher or lower precedence are present. Expressions with higher-precedence operators are evaluated first. The grouping of operands can be forced by using parentheses.

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

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

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

In the expression

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

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

The following tables list the C and C++ language operators in order of precedence and show the direction of associativity for each operator. Operators that have the same rank have the same precedence.

Table 27. Precedence and associativity of postfix operators

Rank	Right associative?	Operator function	Usage
1	yes	▶ C++ global scope resolution	:: <i>name_or_qualified name</i>
1		▶ C++ class or namespace scope resolution	<i>class_or_namespace :: member</i>
2		member selection	<i>object . member</i>
2		member selection	<i>pointer -> member</i>
2		subscripting	<i>pointer [expr]</i>
2		function call	<i>expr (expr_list)</i>
2		value construction	<i>type (expr_list)</i>
2		postfix increment	<i>lvalue ++</i>
2		postfix decrement	<i>lvalue --</i>
2	yes	▶ C++ type identification	<i>typeid (type)</i>
2	yes	▶ C++ type identification at run time	<i>typeid (expr)</i>
2	yes	▶ C++ conversion checked at compile time	<i>static_cast < type > (expr)</i>
2	yes	▶ C++ conversion checked at run time	<i>dynamic_cast < type > (expr)</i>
2	yes	▶ C++ unchecked conversion	<i>reinterpret_cast < type > (expr)</i>
2	yes	▶ C++ const conversion	<i>const_cast < type > (expr)</i>

Table 28. Precedence and associativity of unary operators

Rank	Right associative?	Operator function	Usage
3	yes	size of object in bytes	<i>sizeof expr</i>
3	yes	size of type in bytes	<i>sizeof (type)</i>
3	yes	prefix increment	<i>++ lvalue</i>
3	yes	prefix decrement	<i>-- lvalue</i>
3	yes	bitwise negation	<i>~ expr</i>
3	yes	not	<i>! expr</i>
3	yes	unary minus	<i>- expr</i>

Table 28. Precedence and associativity of unary operators (continued)

Rank	Right associative?	Operator function	Usage
3	yes	unary plus	+ <i>expr</i>
3	yes	address of	& <i>lvalue</i>
3	yes	indirection or dereference	* <i>expr</i>
3	yes	 create (allocate memory)	<i>new type</i>
3	yes	 create (allocate and initialize memory)	<i>new type (expr_list) type</i>
3	yes	 create (placement)	<i>new type (expr_list) type (expr_list)</i>
3	yes	 destroy (deallocate memory)	<i>delete pointer</i>
3	yes	 destroy array	<i>delete [] pointer</i>
3	yes	type conversion (cast)	(<i>type</i>) <i>expr</i>

Table 29. Precedence and associativity of binary operators

Rank	Right associative?	Operator function	Usage
4		 member selection	<i>object .* ptr_to_member</i>
4		 member selection	<i>object ->* ptr_to_member</i>
5		multiplication	<i>expr * expr</i>
5		division	<i>expr / expr</i>
5		modulo (remainder)	<i>expr % expr</i>
6		binary addition	<i>expr + expr</i>
6		binary subtraction	<i>expr - expr</i>
7		bitwise shift left	<i>expr << expr</i>
7		bitwise shift right	<i>expr >> expr</i>
8		less than	<i>expr < expr</i>
8		less than or equal to	<i>expr <= expr</i>
8		greater than	<i>expr > expr</i>
8		greater than or equal to	<i>expr >= expr</i>
9		equal	<i>expr == expr</i>
9		not equal	<i>expr != expr</i>
10		bitwise AND	<i>expr & expr</i>
11		bitwise exclusive OR	<i>expr ^ expr</i>
12		bitwise inclusive OR	<i>expr expr</i>
13		logical AND	<i>expr && expr</i>
14		logical inclusive OR	<i>expr expr</i>
15		conditional expression	<i>expr ? expr : expr</i>

Table 29. Precedence and associativity of binary operators (continued)

Rank	Right associative?	Operator function	Usage
16	yes	simple assignment	<i>lvalue = expr</i>
16	yes	multiply and assign	<i>lvalue *= expr</i>
16	yes	divide and assign	<i>lvalue /= expr</i>
16	yes	modulo and assign	<i>lvalue %= expr</i>
16	yes	add and assign	<i>lvalue += expr</i>
16	yes	subtract and assign	<i>lvalue -= expr</i>
16	yes	shift left and assign	<i>lvalue <<= expr</i>
16	yes	shift right and assign	<i>lvalue >>= expr</i>
16	yes	bitwise AND and assign	<i>lvalue &= expr</i>
16	yes	bitwise exclusive OR and assign	<i>lvalue ^= expr</i>
16	yes	bitwise inclusive OR and assign	<i>lvalue = expr</i>
17	yes	 throw expression	throw <i>expr</i>
18		comma (sequencing)	<i>expr , expr</i>

Examples of expressions and precedence

The parentheses in the following expressions explicitly show how the compiler groups operands and operators.

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

If parentheses did not appear in these expressions, the operands and operators would be grouped in the same manner as indicated by the parentheses. For example, the following expressions produce the same output.

```
total = (4+(5*3));
total = 4+5*3;
```

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

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

in the following ways (as indicated by parentheses):

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

The grouping of operands and operators does not affect the result unless one ordering causes an overflow and another does not. For example, if `price = 32767`, `prov_tax = -42`, and `city_tax = 32767`, and all three of these variables have been declared as integers, the third statement `total = ((price + city_tax) + prov_tax)` will cause an integer overflow and the rest will not.

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 depending on the order in which the functions are called.

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

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

The order of evaluation for function call arguments or for the operands of binary operators is not specified. Therefore, the following expressions are ambiguous:

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

If `y` has the value of 1 before the first statement, it is not known whether or not the value of 1 or 2 is passed to `func1()`. In the second statement, if `i` has the value of 1 before the expression is evaluated, it is not known whether `x[1]` or `x[2]` is passed as the second argument to `func2()`.

Chapter 7. Statements

A statement, the smallest independent computational unit, specifies an action to be performed. In most cases, statements are executed in sequence. The following is a summary of the statements available in C and C++:

- Labeled statements
- Expression statements
- Block statements
- Selection statements
- Iteration statements
- Jump statements
- Declaration statements
-  try blocks
- Null statement

Related information

- Chapter 3, “Data objects and declarations,” on page 39
- “Function declarations” on page 183
- “try blocks (C++ only)” on page 353

Labeled statements

There are three kinds of labels: identifier, case, and default.

Labeled statement syntax

►► *identifier* : *statement* ◀◀

The label consists of the *identifier* and the colon (:) character.

 A label name must be unique within the function in which it appears.

 In C++, an identifier label may only be used as the target of a `goto` statement. A `goto` statement can use a label before its definition. Identifier labels have their own namespace; you do not have to worry about identifier labels conflicting with other identifiers. However, you may not redeclare a label within a function.

Case and default label statements only appear in `switch` statements. These labels are accessible only within the closest enclosing `switch` statement.

case statement syntax

►► *case* *constant_expression* : *statement* ◀◀

default statement syntax

►► *default* : *statement* ◀◀

The following are examples of labels:

```
comment_complete : ;          /* null statement label */
test_for_null : if (NULL == pointer)
```

Related information

- “The goto statement” on page 177
- “The switch statement” on page 166

Expression statements

An *expression statement* contains an expression. The expression can be null.

Expression statement syntax



An expression statement evaluates *expression*, then discards the value of the expression. An expression statement without an expression is a null statement.

The following are examples of statements:

```
printf("Account Number: \n");          /* call to the printf */
marks = dollars * exch_rate;          /* assignment to marks */
(difference < 0) ? ++losses : ++gain; /* conditional increment */
```

Related information

- Chapter 6, “Expressions and operators,” on page 111

Resolution of ambiguous statements

C++ only

The C++ syntax does not disambiguate between expression statements and declaration statements. The ambiguity arises when an expression statement has a function-style cast as its left-most subexpression. (Note that, because C does not support function-style casts, this ambiguity does not occur in C programs.) If the statement can be interpreted both as a declaration and as an expression, the statement is interpreted as a declaration statement.

Note: The ambiguity is resolved only on a syntactic level. The disambiguation does not use the meaning of the names, except to assess whether or not they are type names.

The following expressions disambiguate into expression statements because the ambiguous subexpression is followed by an assignment or an operator. `type_spec` in the expressions can be any type specifier:

```
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 statements are interpreted 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

```

Related information

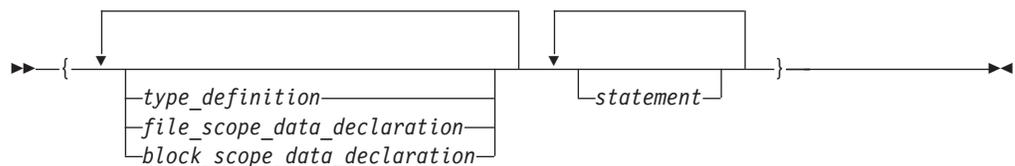
- Chapter 3, “Data objects and declarations,” on page 39
- Chapter 6, “Expressions and operators,” on page 111
- “Function call expressions” on page 116

End of C++ only

Block statements

A *block statement*, or *compound statement*, lets you group any number of data definitions, declarations, and statements into one statement. All definitions, declarations, and statements enclosed within a single set of braces are treated as a single statement. You can use a block wherever a single statement is allowed.

Block statement syntax



A block defines a local scope. If a data object is usable within a block and its identifier is not redefined, all nested blocks can use that data object.

Example of blocks

The following program shows how the values of data objects change in nested blocks:

```

/**
** This example shows how data objects change in nested blocks.
**/
#include <stdio.h>

int main(void)
{
    int x = 1;           /* Initialize x to 1 */
    int y = 3;

    if (y > 0)

```

```

    {
        int x = 2;           /* Initialize x to 2 */
        printf("second x = %4d\n", x);
    }
    printf("first x = %4d\n", x);

    return(0);
}

```

The program produces the following output:

```

second x =    2
first x =    1

```

Two variables named `x` are defined in `main`. The first definition of `x` retains storage while `main` is running. However, because the second definition of `x` occurs within a nested block, `printf("second x = %4d\n", x);` recognizes `x` as the variable defined on the previous line. Because `printf("first x = %4d\n", x);` is not part of the nested block, `x` is recognized as the first definition of `x`.

Selection statements

Selection statements consist of the following types of statements:

- The `if` statement
- The `switch` statement

The `if` statement

An `if` statement is a selection statement that allows more than one possible flow of control.

C++ An *if statement* lets you conditionally process a statement when the specified test expression, implicitly converted to `bool`, evaluates to `true`. If the implicit conversion to `bool` fails the program is ill-formed.

C In C, an *if statement* lets you conditionally process a statement when the specified test expression evaluates to a nonzero value. The test expression must be of arithmetic or pointer type.

You can optionally specify an `else` clause on the `if` statement. If the test expression evaluates to `false` (or in C, a zero value) and an `else` clause exists, the statement associated with the `else` clause runs. If the test expression evaluates to `true`, the statement following the expression runs and the `else` clause is ignored.

if statement syntax

```

▶▶ 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.

A single statement following any selection statements (`if`, `switch`) is treated as a compound statement containing the original statement. As a result any variables declared on that statement will be out of scope after the `if` statement. For example:

```

if (x)
int i;

```

is equivalent to:

```
if (x)
{ int i; }
```

Variable `i` is visible only within the `if` statement. The same rule applies to the `else` part of the `if` statement.

Examples of if statements

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, braces might be needed to force a particular `else` clause to associate with the correct `if` statement. In this example, omitting the braces would cause the `else` clause to associate with the nested `if` statement.

```
if (kegs > 0) {
    if (furlongs > kegs)
        fxph = furlongs/kegs;
}
else
    fxph = 0;
```

The following example shows an `if` statement nested within an `else` clause. This example tests multiple conditions. The tests are made in order of their appearance. If one test evaluates to a nonzero value, a statement runs and the entire `if` statement ends.

```
if (value > 0)
    ++increase;
else if (value == 0)
    ++break_even;
else
    ++decrease;
```

Related information

- “Boolean types” on page 50

The switch statement

A *switch statement* is a selection statement that 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 or enumeration value. The body of the switch statement contains *case clauses* that consist of

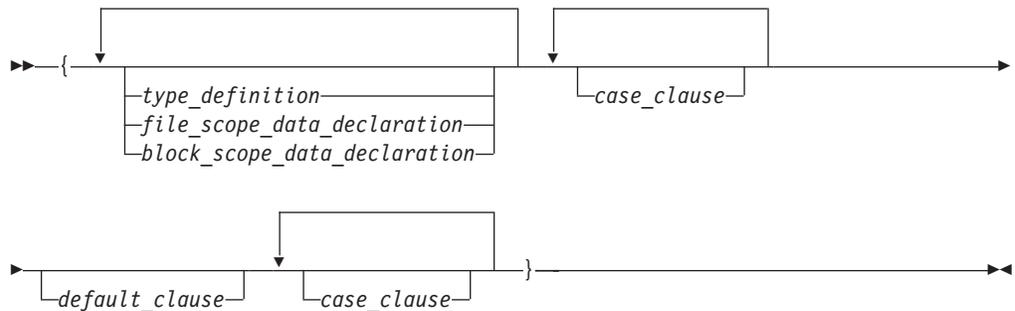
- A case label
- An optional default label
- A case expression
- A list of statements.

If the value of the switch expression equals the value of one of the case expressions, the statements following that case expression are processed. If not, the default label statements, if any, are processed.

switch statement syntax

→ switch (*expression*) *switch_body* →

The *switch body* is enclosed in braces and can contain definitions, declarations, *case clauses*, and a *default clause*. Each case clause and default clause can contain statements.



Note: An initializer within a *type_definition*, *file_scope_data_declaration* or *block_scope_data_declaration* is ignored.

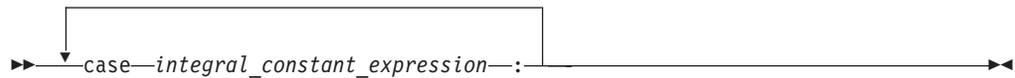
A *case clause* contains a *case label* followed by any number of statements. A case clause has the form:

Case clause syntax



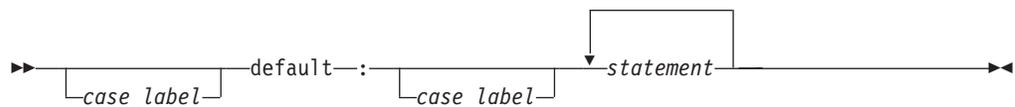
A *case label* contains the word case followed by an integral constant expression and a colon. The value of each integral constant expression must represent a different value; you cannot have duplicate case labels. Anywhere you can put one case label, you can put multiple case labels. A case label has the form:

case label syntax



A *default clause* contains a default label followed by one or more statements. You can put a case label on either side of the default label. A switch statement can have only one default label. A *default clause* has the form:

Default clause statement



The switch statement passes control to the statement following one of the labels or to the statement following the switch body. The value of the expression that precedes the switch body determines which statement receives control. This expression is called the *switch expression*.

The value of the switch expression is compared with the value of the expression in each case label. If a matching value is found, control is passed to the statement following the case label that contains the matching value. If there is no matching value but there is a default label in the switch body, control passes to the default labelled statement. If no matching value is found, and there is no default label anywhere in the switch body, no part of the switch body is processed.

When control passes to a statement in the switch body, control only leaves the switch body when a break statement is encountered or the last statement in the switch body is processed.

If necessary, an integral promotion is performed on the controlling expression, and all expressions in the case statements are converted to the same type as the controlling expression. The switch expression can also be of class type if there is a single conversion to integral or enumeration type.

Restrictions on switch statements

You can put data definitions at the beginning of the switch body, but the compiler does not initialize auto and register variables at the beginning of a switch body. You can have declarations in the body of the switch statement.

You cannot use a switch statement to jump over initializations.

C When the scope of an identifier with a variably modified type includes a case or default label of a switch statement, the entire switch statement is considered to be within the scope of that identifier. That is, the declaration of the identifier must precede the switch statement.

C++ In C++, you cannot transfer control over a declaration containing an explicit or implicit initializer unless the declaration is located in an inner block that is completely bypassed by the transfer of control. All declarations within the body of a switch statement that contain initializers must be contained in an inner block.

Examples of switch statements

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;

printf("Enter an arithmetic operator\n");
scanf("%c",&key);

switch (key)
{
    case '+':
        add();
        break;

    case '-':
        subtract();
        break;

    case '*':
        multiply();
        break;

    case '/':
        divide();
        break;

    default:
        printf("invalid key\n");
        break;
}
```

If the switch expression matches a case expression, the statements following the case expression are processed until a break statement is encountered or the end of the switch body is reached. In the following example, break statements are not present. If the value of text[i] is equal to 'A', all three counters are incremented. If the value of text[i] is equal to 'a', lettera and total are increased. Only total is increased if text[i] is not equal to 'A' or 'a'.

```
char text[100];
int capa, lettera, total;

// ...

for (i=0; i<sizeof(text); i++) {

    switch (text[i])
    {
        case 'A':
            capa++;
        case 'a':
            lettera++;
        default:
            total++;
    }
}
```

The following switch statement performs the same statements for more than one case label:

CCNRAB1

```
/**
 ** 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);

    /* 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, control passes to the statement:

```
printf("month %d is a spring month\n",
month);
```

The break statement passes control to the statement following the switch body.

Related information

- “Case and Default Labels” on page 161

- “The break statement” on page 173

Iteration statements

Iteration statements consist of the following types of statements:

- The while statement
- The do statement
- The for statement
-

Related information

- “Boolean types” on page 50

The while statement

A *while statement* repeatedly runs the body of a loop until the controlling expression evaluates to `false` (or `0` in C).

while statement syntax

▶▶ `while`—(`expression`)—`statement`◀◀

C The *expression* must be of arithmetic or pointer type. **C++** The expression must be convertible to `bool`.

The expression is evaluated to determine whether or not to process the body of the loop. If the expression evaluates to `false`, the body of the loop never runs. If the expression does not evaluate to `false`, the loop body is processed. After the body has run, control passes back to the expression. Further processing depends on the value of the condition.

A `break`, `return`, or `goto` statement can cause a `while` statement to end, even when the condition does not evaluate to `false`.

C++ A `throw` expression also can cause a `while` statement to end prior to the condition being evaluated.

In the following example, `item[index]` triples and is printed out, as long as the value of the expression `++index` is less than `MAX_INDEX`. When `++index` evaluates to `MAX_INDEX`, the `while` statement ends.

CCNRAA7

```
/**
 ** 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);
}

```

The do statement

A *do statement* repeatedly runs a statement until the test expression evaluates to false (or 0 in C). Because of the order of processing, the statement is run at least once.

do statement syntax

► `do statement while (expression);` ◀

► **C** The *expression* must be of arithmetic or pointer type. ► **C++** The controlling *expression* must be convertible to type `bool`.

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 false, the statement runs again. When the `while` clause evaluates to false, the statement ends.

A `break`, `return`, or `goto` statement can cause the processing of a `do` statement to end, even when the `while` clause does not evaluate to false.

► **C++** A `throw` expression also can cause a `while` statement to end prior to the condition being evaluated.

The following example keeps incrementing `i` while `i` is less than 5:

```

#include <stdio.h>

int main(void) {
    int i = 0;
    do {
        i++;
        printf("Value of i: %d\n", i);
    }
    while (i < 5);
    return 0;
}

```

The following is the output of the above example:

```

Value of i: 1
Value of i: 2
Value of i: 3
Value of i: 4
Value of i: 5

```

The for statement

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 (the *condition*)

- Evaluate an expression after each iteration of the statement (often used to increment for each iteration)
- Repeatedly process the statement if the controlling part does not evaluate to false (or 0 in C).

for statement syntax

```

▶▶ for ( [expression1] ; [expression2] ; [expression3] )
▶▶ statement

```

expression1 is the *initialization expression*. It is evaluated only before the *statement* is processed for the first time. You can use this expression to initialize a variable. You can also use this expression to declare a variable, provided that the variable is not declared as static (it must be automatic and may also be declared as register). If you declare a variable in this expression, or anywhere else in *statement*, that variable goes out of scope at the end of the for loop. 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 *conditional expression*. It is evaluated before each iteration of the *statement*. C *expression2* must be of arithmetic or pointer type. C++ *expression3* must be convertible to type bool.

If it evaluates to false (or 0 in C), the statement is not processed and control moves to the next statement following the for statement. If *expression2* does not evaluate to false, the statement is processed. If you omit *expression2*, it is as if the expression had been replaced by true, and the for statement is not terminated by failure of this condition.

expression3 is evaluated after each iteration of the *statement*. This expression is often used for incrementing, decrementing, or assigning to 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 false. If you omit *expression2*, you must use a break, return, or goto statement to end the for statement.

Examples of for statements

The following for statement prints the value of count 20 times. The for statement initially sets the value of count to 1. After each iteration of the statement, count is incremented.

```

int 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.

```

int count = 1;
while (count <= 20)
{
    printf("count = %d\n", count);
    count++;
}

```

The following for statement does not contain an initialization expression:

```
for (; index > 10; --index)
{
    list[index] = var1 + var2;
    printf("list[%d] = %d\n", index,
        list[index]);
}
```

The following for statement will continue running until scanf receives the letter e:

```
for (;;)
{
    scanf("%c", &letter);
    if (letter == '\n')
        continue;
    if (letter == 'e')
        break;
    printf("You entered the letter %c\n", letter);
}
```

The following for statement contains multiple initializations and increments. The comma operator makes this construction possible. 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 both i and j to be incremented 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 having the dimensions [5][3].

```
for (row = 0; row < 5; row++)
    for (column = 0; column < 3; column++)
        printf("%d\n",
            table[row][column]);
```

The outer statement is processed as long as the value of row is less than 5. Each time the outer for statement is executed, the inner for statement sets the initial value of column to zero and the statement of the inner for statement is executed 3 times. The inner statement is executed as long as the value of column is less than 3.

Jump statements

Jump statements consist of the following types of statements:

- The break statement
- The continue statement
- The return statement
- The goto statement

The break statement

A *break statement* lets you end an *iterative* (do, for, or while) statement or a switch statement and exit from it at any point other than the logical end. A break may only appear on one of these statements.

break statement syntax

▶▶—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` statement, the `break` passes control out of the `switch` body to the next statement outside the `switch` statement.

The continue statement

A *continue statement* ends 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—;—————▶▶

A `continue` statement can only appear within the body of an iterative statement, such as `do`, `for`, or `while`.

The `continue` statement ends the processing of the action part of an iterative statement and moves control to the loop continuation portion of the statement. For example, if the iterative statement is a `for` statement, control moves to the third expression in the condition part of the statement, then to the second expression (the test) in the condition part of the statement.

Within nested statements, the `continue` statement ends only the current iteration of the `do`, `for`, or `while` statement immediately enclosing it.

Examples of continue statements

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.

CCNRAA3

```
/**
 ** This example shows a continue statement in a for statement.
 **/

#include <stdio.h>
#define SIZE 5

int main(void)
{
    int i;
    static float rates[SIZE] = { 1.45, 0.05, 1.88, 2.00, 0.75 };

    printf("Rates over 1.00\n");
    for (i = 0; i < SIZE; i++)
    {
        if (rates[i] <= 1.00) /* skip rates <= 1.00 */
            continue;
        printf("rate = %.2f\n", rates[i]);
    }
}
```

```

    }
    return(0);
}

```

The program produces the following output:

```

Rates over 1.00
rate = 1.45
rate = 1.88
rate = 2.00

```

The following example shows a `continue` statement in a nested loop. When the inner loop encounters a number in the array `strings`, that iteration of the loop ends. Processing continues with the third expression of the inner loop. The inner loop ends when the `'\0'` escape sequence is encountered.

CCNRAA4

```

/**
 ** This program counts the characters in strings that are part
 ** of an array of pointers to characters. The count excludes
 ** the digits 0 through 9.
 **/

#include <stdio.h>
#define SIZE 3

int main(void)
{
    static char *strings[SIZE] = { "ab", "c5d", "e5" };
    int i;
    int letter_count = 0;
    char *pointer;
    for (i = 0; i < SIZE; i++)          /* for each string */
        for (pointer = strings[i]; *pointer != '\0'; /* for each character */
            ++pointer)
        {
            /* if a number */
            if (*pointer >= '0' && *pointer <= '9')
                continue;
            letter_count++;
        }
    printf("letter count = %d\n", letter_count);
    return(0);
}

```

The program produces the following output:

```

letter count = 5

```

The return statement

A *return statement* ends the processing of the current function and returns control to the caller of the function.

return statement syntax

```

return [ ( [ ] ) expression ] ;

```

A value-returning function should include a return statement, containing an *expression*.  If an expression is not given on a return statement in a function declared with a non-void return type, the compiler issues a warning message.  If an expression is not given on a return statement in a function declared with a non-void return type, the compiler issues an error message.

If the data type of the expression is different from the function return type, conversion of the return value takes place as if the value of the expression were assigned to an object with the same function return type.

For a function of return type `void`, a return statement is not strictly necessary. If the end of such a function is reached without encountering a return statement, control is passed to the caller as if a return statement without an expression were encountered. In other words, an implicit return takes place upon completion of the final statement, and control automatically returns to the calling function.  If a return statement is used, it must not contain an expression.

Examples of return statements

The following are examples of return statements:

```
return;           /* Returns no value      */
return result;   /* Returns the value of result */
return 1;        /* Returns the value 1        */
return (x * x);  /* Returns the value of x * x  */
```

The following function searches through an array of integers to determine if a match exists for the variable `number`. If a match exists, the function `match` returns the value of `i`. If a match does not exist, the function `match` returns the value `-1` (negative one).

```
int match(int number, int array[ ], int n)
{
    int i;

    for (i = 0; i < n; i++)
        if (number == array[i])
            return (i);
    return(-1);
}
```

A function can contain multiple return statements. For example:

```
void copy( int *a, int *b, int c)
{
    /* Copy array a into b, assuming both arrays are the same size */

    if (!a || !b)           /* if either pointer is 0, return */
        return;

    if (a == b)             /* if both parameters refer */
        return;             /* to same array, return */

    if (c == 0)             /* nothing to copy */
        return;

    for (int i = 0; i < c; ++i) /* do the copying */
        b[i] = a[i];         /* implicit return */
}
```

In this example, the return statement is used to cause a premature termination of the function, similar to a `break` statement.

An expression appearing in a return statement is converted to the return type of the function in which the statement appears. If no implicit conversion is possible, the return statement is invalid.

Related information

- “Function return type specifiers” on page 198
- “Function return values” on page 199

The goto statement

A *goto statement* causes your program to unconditionally transfer control to the statement associated with the label specified on the *goto* statement.

goto statement syntax

```
▶▶ 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 an active block is exited using a *goto* statement, any local variables are destroyed when control is transferred from that block.

You cannot use a *goto* statement to jump over initializations.

C A *goto* statement is allowed to jump within the scope of a variable length array, but not past any declarations of objects with variably modified types.

The following example shows a *goto* statement that is used to jump out of a nested loop. This function could be written without using a *goto* statement.

CCNRAA6

```
/**
 ** 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]);
        }
    out_of_bounds:
}
```

```

    }
    return;
    out_of_bounds: printf("number must be 1 through 6\n");
}

```

Related information

- “Labeled statements” on page 161

Null statement

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.

The following example initializes the elements of the array `price`. Because the initializations occur within the for expressions, a statement is only needed to finish the for syntax; no operations are required.

```

for (i = 0; i < 3; price[i++] = 0)
;

```

A null statement can be used when a label is needed before the end of a block statement. For example:

```

void func(void) {
    if (error_detected)
        goto depart;
    /* further processing */
    depart: ; /* null statement required */
}

```

Inline assembly statements (C only)

IBM extension

When the GENASM compiler option is in effect, the compiler provides support for embedded assembly code fragments among C source statements. This extension allows C programs to invoke MVS system services directly via system-provided assembly macros.

The syntax is as follows:

asm statement syntax — statement in local scope

```

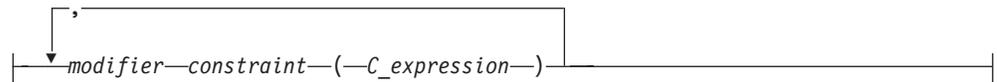
▶▶ [__asm__] [volatile]
▶(-code_format_string-: [output-:] [input-:] [clobbers-:]-)◀◀

```

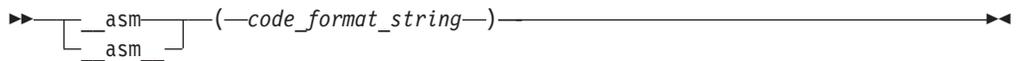
input:



output:



asm statement syntax — statement in global scope



The qualifier `volatile` instructs the compiler that the assembler instructions may update memory not listed in *output*, *input*, or *clobbers*.

The *code_format_string* is the source text of the asm instructions and is a string literal similar to a `printf` format specifier. Depending on the operands taken by the assembly instruction, the string contains zero or more modifiers, each of which corresponds to an input or output operand, separated by commas.

The *input* consists of zero, one or more input operands, separated by commas. Each operand consists of a *constraint*(*C_expression*) pair.

The *output* consists of zero, one or more output operands, separated by commas. Each operand consists of a *constraint*(*C_expression*) pair. The output operand must be constrained by the `=` or `+` modifier (described below).

The *modifier* is one of the following:

- =** Indicates that the operand is write-only for this instruction. The previous value is discarded and replaced by output data.
- +** Indicates that the operand is both read and written by the instruction.
- &** Indicates that the operand may be modified before the instruction is finished using the input operands; a register that is used as input should not be reused here.

Note: The `&` modifier is ignored in z/OS V1R9.

The *constraint* is a string literal that describes the kind of operand that is permitted, one character per constraint. The following constraints are supported:

- a** Use an address register (general purpose register except r0)
- d** Use a data register (equivalent to the `r` constraint)
- g** Use a general register, memory, or immediate operand.
- i | n** Use an immediate integer or string literal operand.
- m** Use a memory operand supported by the machine.
- o** Use a memory operand that is offsetable.

- r** Use a general register.
- s** Use a string literal operand.
- 0, 1, 2, ...** A matching constraint. Allocate the same register in output as in the corresponding input.
- I, J, K** Constant values. Fold the expression in the operand and substitute the value into the % specifier.
- XL** Use only the parameter constraints listed in this constraint. XL is an optional prefix, followed by a colon (:), to introduce any of the following parameter constraints:
 - DS** Do not generate a definition for the operand defined in the assembly statement; instead, substitute an assembly instruction to define the operand. Optionally, to specify the data size of the operand defined in the assembly statement, use a colon (:) followed by a positive integer. If you do not specify a data size, the size specified in the ASMDATASIZE option is used.
 - RP** The operand requires a register pair. Optionally, to specify the constraint for the register pair, specify a :, followed by the *register_type*, optionally followed by another : and an optional *register_pair_flag*. The *register_pair_flag* can be one of the following:
 - o** The operand needs an odd/even register pair.
 - e** The operand needs an even/odd register pair.

If you do not specify a register type, r (general purpose register) is used as the default. If you do not specify a register pair flag, e (even/odd pair) is used as the default.
 - NR** Use the named general purpose register. Use a colon (:) followed by the general purpose register name (see below for acceptable register names).

Note: The XL constraints can be used for both input and output operands, with the exception of DS, which can only be used for output operands.

The *C_expression* is a C expression whose value is used as the operand for the asm instruction. Output operands must be modifiable lvalues. The *C_expression* must be consistent with the constraint specified on it. For example, if i is specified, the operand must be an integer constant number.

Note: If pointer expressions are used in *input* or *output*, the assembly instructions should honor the ANSI aliasing rule (see “Type-based aliasing” on page 82 for more information). This means that indirect addressing using values in pointer expression operands should be consistent with the pointer types; otherwise, you must disable the ANSIALIAS option during compilation.

clobbers is a comma-separated list of register names enclosed in double quotes. These are registers that can be updated by the asm instruction. The following are valid register names:

r0 or R0 to r15 or R15
General purpose registers

Related information

- GENASM, ANSIALIAS options in the *z/OS XL C/C++ User's Guide*

Examples of inline assembly statements

In the following example:

```
__asm("x DC F'0' ":"XL:DS:4"(x));
```

The contents of the instruction `x DC F'0'` will be inserted into the assembly file to define `x` directly. The constraint `XL:DS:4` indicates that the data size for this variable is 4.

The following is an example of an assembly statement using register pair constraints:

```
__asm ( " CLCL %0, %2" : : "XL:RP:r:e"(s1), "r"(len1), "RP"(s2), "r"(len2));
```

In this example, `CLCL` is the data to be defined in the assembly code. `%0` and `%2` are the operands, which are to be substituted by the C expressions in the output/input operand fields. For the first two output operands, an even/odd register pair will be allocated; for the second two output operands, another even/odd register pair will be allocated. The `r` constraint indicates that a general purpose register is required. Within these restrictions, the compiler is free to choose any registers to substitute for `%0` and `%2`.

Restrictions on inline assembly statements

The following are restrictions on the use of inline assembly statements:

- The assembler instructions must be self-contained within an `asm` statement. The `asm` statement can only be used to generate instructions. All connections to the rest of the program must be established through the output and input operand list.
- If an `asm` statement is used to define data, it cannot contain assembly instructions for other purposes.
- Only `asm` statements that are used to define data can exist in global scope.
- Each assembly statement can define only one variable.
- You must ensure that the symbol used in the assembly statement is unique within the scope of the source file and is valid according to the assembler's requirements.
- Referencing an external symbol directly, without going through the operand list, is not supported.

Related information

- "Variables in specified registers (C only)" on page 48

End of IBM extension

Chapter 8. Functions

In the context of programming languages, the term *function* means an assemblage of statements used for computing an output value. The word is used less strictly than in mathematics, where it means a set relating input variables *uniquely* to output variables. Functions in C or C++ programs may not produce consistent outputs for all inputs, may not produce output at all, or may have side effects. Functions can be understood as user-defined operations, in which the parameters of the parameter list, if any, are the operands.

This section discusses the following topics:

- “Function declarations and definitions”
- “Function storage class specifiers” on page 188
- “Function specifiers” on page 190
- “Function return type specifiers” on page 198
- “Function declarators” on page 199
- “Function attributes” on page 203
- “The main() function” on page 205
- “Function calls” on page 207
- “Default arguments in C++ functions” on page 211
- “Pointers to functions” on page 214

Function declarations and definitions

The distinction between a function *declaration* and function *definition* is similar to that of a data declaration and definition. The declaration establishes the names and characteristics of a function but does not allocate storage for it, while the *definition* specifies the body for a function, associates an identifier with the function, and allocates storage for it. Thus, the identifiers declared in this example:

```
float square(float x);
```

do not allocate storage.

The *function definition* contains a function declaration and the body of a function. The body is a block of statements that perform the work of the function. The identifiers declared in this example allocate storage; they are both declarations and definitions.

```
float square(float x)
{ return x*x; }
```

A function can be declared several times in a program, but all declarations for a given function must be compatible; that is, the return type is the same and the parameters have the same type. However, a function can only have one definition. Declarations are typically placed in header files, while definitions appear in source files.

Function declarations

A function identifier preceded by its return type and followed by its parameter list is called a *function declaration* or *function prototype*. The prototype informs the compiler of the format and existence of a function prior to its use. The compiler

checks for mismatches between the parameters of a function call and those in the function declaration. The compiler also uses the declaration for argument type checking and argument conversions.

▶ **C++** Implicit declaration of functions is not allowed: you must explicitly declare every function before you can call it.

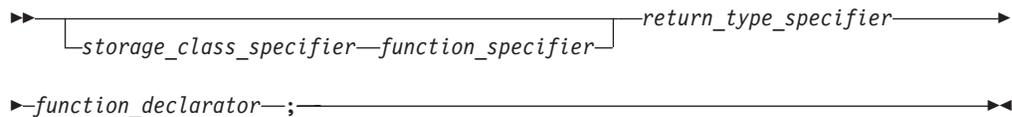
▶ **C** If a function declaration is not visible at the point at which a call to the function is made, the compiler assumes an implicit declaration of `extern int func();` However, for conformance to C99, you should explicitly prototype every function before making a call to it.

The elements of a declaration for a function are as follows:

- Function storage class specifiers, which specify linkage
- Function return type specifiers, which specify the data type of a value to be returned
- Function specifiers, which specify additional properties for functions
- Function declarators, which include function identifiers as well as lists of parameters

All function declarations have the form:

Function declaration syntax

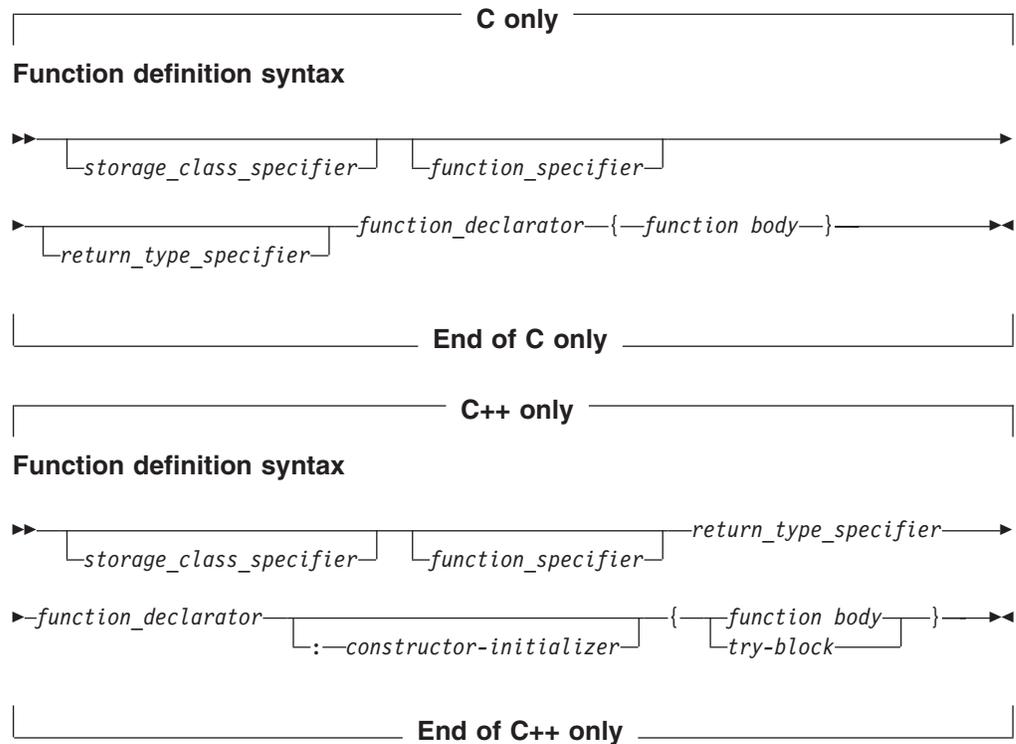


Function definitions

The elements of a function definition are as follows:

- Function storage class specifiers, which specify linkage
- Function return type specifiers, which specify the data type of a value to be returned
- Function specifiers, which specify additional properties for functions
- Function declarators, which include function identifiers as well as lists of parameters
- The *function body*, which is a braces-enclosed series of statements representing the actions that the function performs
- ▶ **C++** Constructor-initializers, which are used only in constructor functions declared in classes; they are described in “Constructors (C++ only)” on page 301.
- ▶ **C++** Try blocks, which are used in class functions; they are described in “try blocks (C++ only)” on page 353.

Function definitions take the following form:



Examples of function declarations

The following code fragments show several function declarations (or *prototypes*). The first declares a function `f` that takes two integer arguments and has a return type of void:

```
void f(int, int);
```

This 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 code fragment declares a function `f1` that 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, a typedef can be used for the complicated return type of function `f1`:

```
typedef int f1_return_type(int);
f1_return_type* f1(int);
```

The following declaration is of an external function `f2` that takes a constant integer as its first argument, can have a variable number and variable types of other arguments, and returns type `int`.

```
int extern f2(const int, ...); /* C version */
int extern f2(const int ...); // C++ version
```

Function `f6` is a const class member function of class `X`, takes no arguments, and has a return type of `int`:

```

class X
{
public:
    int f6() const;
};

```

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);

```

Examples of function definitions

The following example is a definition of the function sum:

```

int sum(int x,int y)
{
    return(x + y);
}

```

The function sum has external linkage, returns an object that has type int, and has two parameters of type int declared as x and y. The function body contains a single statement that returns the sum of x and y.

The following function set_date declares a pointer to a structure of type date as a parameter. date_ptr has the storage class specifier register.

```

void set_date(register struct date *date_ptr)
{
    date_ptr->mon = 12;
    date_ptr->day = 25;
    date_ptr->year = 87;
}

```

Compatible functions

C only

For two function types to be compatible, they must meet the following requirements:

- They must agree in the number of parameters (and use of ellipsis).
- They must have compatible return types.
- The corresponding parameters must be compatible with the type that results from the application of the default argument promotions.

The composite type of two function types is determined as follows:

- If one of the function types has a parameter type list, the composite type is a function prototype with the same parameter type list.
- If both function types have parameter type lists, the composite type of each parameter is determined as follows:
 - The composite of parameters of different rank is the type that results from the application of the default argument promotions.
 - The composite of parameters with array or function type is the adjusted type.
 - The composite of parameters with qualified type is the unqualified version of the declared type.

For example, for the following two function declarations:

```
int f(int (*)(), double (*)[3]);
int f(int (*)(char *), double (*)[]);
```

The resulting composite type would be:

```
int f(int (*)(char *), double (*)[3]);
```

If the function declarator is not part of the function declaration, the parameters may have incomplete type. The parameters may also specify variable length array types by using the `[*]` notation in their sequences of declarator specifiers. The following are examples of compatible function prototype declarators:

```
double maximum(int n, int m, double a[n][m]);
double maximum(int n, int m, double a[*][*]);
double maximum(int n, int m, double a[ ][*]);
double maximum(int n, int m, double a[ ][m]);
```

Related information

- “Compatible and composite types” on page 41

End of C only

Multiple function declarations

C++ only

All function declarations for a particular function must have the same number and type of parameters, and must have the same return type.

These return and parameter types are part of the function type, although the default arguments and exception specifications are not.

If a previous declaration of an object or function is visible in an enclosing scope, the identifier has the same linkage as the first declaration. However, a variable or function that has no linkage and later declared with a linkage specifier will have the linkage you have specified.

For the purposes of argument matching, ellipsis and linkage keywords are considered a part of the function type. They must be used consistently in all declarations of a function. If the only difference between the parameter 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` type qualifier is also part of the function type, but can only be part of a declaration or definition of a nonstatic member function.

If two function declarations match in both return type and parameter lists, then the second declaration is treated as redeclaration of the first. The following example declares the same function:

```
int foo(const string &bar);
int foo(const string &);
```

Declaring two functions differing only in return type is not valid function overloading, and is flagged as a compile-time error. For example:

```
void f();
int f();    // error, two definitions differ only in
           // return type
```

```
int g()
{
    return f();
}
```

Related information

- “Overloading functions (C++ only)” on page 225

End of C++ only

Function storage class specifiers

For a function, the storage class specifier determines the linkage of the function. By default, function definitions have external linkage, and can be called by functions defined in other files.  An exception is inline functions, which are treated by default as having internal linkage; see “Linkage of inline functions” on page 191 for more information.

A storage class specifier may be used in both function declarations and definitions. The only storage class options for functions are:

- `static`
- `extern`

The static storage class specifier

A function declared with the `static` storage class specifier has internal linkage, which means that it may be called only within the translation unit in which it is defined.

The `static` storage class specifier can be used in a function declaration only if it is at file scope. You cannot declare functions within a block as `static`.

 This use of `static` is deprecated in C++. Instead, place the function in the unnamed namespace.

Related information

- “Internal linkage” on page 7
- Chapter 9, “Namespaces (C++ only),” on page 217

The extern storage class specifier

A function that is declared with the `extern` storage class specifier has external linkage, which means that it can be called from other translation units. The keyword `extern` is optional; if you do not specify a storage class specifier, the function is assumed to have external linkage.

 In z/OS XL C++, an `extern` declaration cannot appear in class scope.

C++ only

In z/OS XL C++, you can use the `extern` keyword with arguments that specify the type of linkage.

extern function storage class specifier syntax

►—extern—“—*linkage_specification*—”—————►

where *linkage_specification* can be any of the following:

- builtin
- C
- C++
- COBOL
- FORTRAN
- OS
- OS_DOWNSTACK
- OS_NOSTACK
- OS_UPSTACK
- PLI

For an explanation of these options, see the descriptions in “#pragma linkage (C only)” on page 419.

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, such as `qsort`. Use the extern "C" linkage specification to ensure that the declared linkages are the same. 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

int main() {
    void *table;

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

While the C++ language supports overloading, other languages do not. The implications of this are:

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

```
int func(int);          // function with C++ linkage
int func(char);        // overloaded function with C++ linkage
```

By contrast, 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
```

- Only one non-C++-linkage function can have the same name as overloaded functions. For example:

```
int func(char);
int func(int);
extern "FORTRAN"{int func(int,int);}
```

However, the non-C++-linkage function cannot have the same parameters as any of the C++ functions with the same name:

```
int func(char); // first function with C++ linkage
int func(int, int); // second function with C++ linkage
extern "FORTRAN"{int func(int,int);} // not allowed since the parameter
// list is the same as the one for
// the second function with C++ linkage
// compiler will issue an error message
```

End of C++ only

Related information

- “External linkage” on page 8
- “Language linkage (C++ only)” on page 9
- “Class scope (C++ only)” on page 4
- Chapter 9, “Namespaces (C++ only),” on page 217

Function specifiers

The available function specifiers for function definitions are:

- `inline`, which instructs the compiler to expand a function definition at the point of a function call.

z/OS only

-  `__cdecl`, which sets linkage conventions for C++ function calls to C functions.
-  `_Export`, which makes function definitions available to other modules.

End of z/OS only

-  `explicit`, which can only be used for member functions of classes, and is described in “The explicit specifier (C++ only)” on page 314
-  `virtual`, which can only be used for member functions of classes, and is described in “Virtual functions (C++ only)” on page 291

The inline function specifier

An inline function is one for which the compiler copies the code from the function definition directly into the code of the calling function rather than creating a separate set of instructions in memory. Instead of transferring control to and from the function code segment, a modified copy of the function body may be substituted directly for the function call. In this way, the performance overhead of a function call is avoided. Using the `inline` specifier is only a suggestion to the compiler that an inline expansion can be performed; the compiler is free to ignore the suggestion.

▶ **C** Any function, with the exception of `main`, can be declared or defined as inline with the `inline` function specifier. Static local variables are not allowed to be defined within the body of an inline function.

▶ **C++** C++ functions implemented inside of a class declaration are automatically defined inline. Regular C++ functions and member functions declared outside of a class declaration, with the exception of `main`, can be declared or defined as inline with the `inline` function specifier. Static locals and string literals defined within the body of an inline function are treated as the same object across translation units; see “Linkage of inline functions” for details.

The following code fragment shows an inline function definition:

```
inline int add(int i, int j) { return i + j; }
```

The use of the `inline` specifier does not change the meaning of the function. However, the inline expansion of a function may not preserve the order of evaluation of the actual arguments.

The most efficient way to code an inline function is to place the inline function definition in a header file, and then include the header in any file containing a call to the function which you would like to inline.

Note: ▶ **C** To enable the `inline` function specifier in C, you must compile with `c99` or the `LANGVL(STDC99)` or `LANGVL(EXTC99)` options.

Linkage of inline functions

C only

In C, inline functions are treated by default as having *static* linkage; that is, they are only visible within a single translation unit. Therefore, in the following example, even though function `foo` is defined in exactly the same way, `foo` in file `a.c` and `foo` in file `b.c` are treated as separate functions: two function bodies are generated, and assigned two different addresses in memory:

```
// a.c

#include <stdio.h>

inline int foo(){
return 3;
}

void g() {
printf("foo called from g: return value = %d, address = %p\n", foo(), &foo);
}

// b.c

#include <stdio.h>

inline int foo(){
return 3;
}

void g();
```

```
int main() {
printf("foo called from main: return value = %d, address = %p\n", foo(), &foo);
g();
}
```

The output from the compiled program is:

```
foo called from main: return value = 3, address = 0x10000580
foo called from g: return value = 3, address = 0x10000500
```

Since inline functions are treated as having internal linkage, an inline function definition can co-exist with a regular, external definition of a function with the same name in another translation unit. However, when you call the function from the file containing the inline definition, the compiler may choose *either* the inline version defined in the same file *or* the external version defined in another file for the call; your program should not rely on the inline version being called. In the following example, the call to `foo` from function `g` could return either 6 or 3:

```
// a.c

#include <stdio.h>

inline int foo(){
return 6;
}

void g() {
printf("foo called from g: return value = %d\n", foo());
}

// b.c

#include <stdio.h>

int foo(){
return 3;
}

void g();

int main() {
printf("foo called from main: return value = %d\n", foo());
g();
}
```

Similarly, if you define a function as `extern inline`, or redeclare an `inline` function as `extern`, the function simply becomes a regular, external function and is not inlined.

End of C only

C++ only

You must define an inline function in exactly the same way in each translation unit in which the function is used or called. Furthermore, if a function is defined as `inline`, but never used or called within the same translation unit, it is discarded by the compiler.

Nevertheless, in C++, inline functions are treated by default as having *external* linkage, meaning that the program behaves as if there is only one copy of the function. The function will have the same address in all translation units and each

translation unit will share any static locals and string literals. Therefore, compiling the previous example gives the following output:

```
foo called from main: return value = 3, address = 0x10000580
foo called from g: return value = 3, address = 0x10000580
```

Redefining an inline function with the same name but with a different function body is illegal; however, the compiler does not flag this as an error, but simply generates a function body for the version defined in the first file entered on the compilation command line, and discards the others. Therefore, the following example, in which inline function `foo` is defined differently in two different files, may not produce the expected results:

```
// a.C

#include <stdio.h>

inline int foo(){
return 6;
}

void g() {
printf("foo called from g: return value = %d, address = %p\n", foo(), &foo);
}

// b.C

#include <stdio.h>

inline int foo(){
return 3;
}

void g();

int main() {
printf("foo called from main: return value = %d, address = %p\n", foo(), &foo);
g();
}
```

When compiled with the command `xlc++ a.C b.C`, the output is:

```
foo called from main: return value = 6, address = 0x10001640
foo called from g: return value = 6, address = 0x10001640
```

The call to `foo` from `main` does not use the inline definition provided in `b.C`, but rather calls `foo` as a regular external function defined in `a.C`. It is your responsibility to ensure that inline function definitions with the same name match exactly across translation units, to avoid unexpected results.

Because inline functions are treated as having external linkage, any static local variables or string literals that are defined within the body of an inline function are treated as the same object across translation units. The following example demonstrates this:

```
// a.C

#include <stdio.h>

inline int foo(){
static int x = 23;
printf("address of x = %p\n", &x);
x++;
return x;
}
```

```

void g() {
printf("foo called from g: return value = %d\n", foo());
}

// b.C

#include <stdio.h>

inline int foo()
{
static int x=23;
printf("address of x = %p\n", &x);
x++;
return x;
}

void g();

int main() {
printf("foo called from main: return value = %d\n", foo());
g();
}

```

The output of this program shows that `x` in both definitions of `foo` is indeed the same object:

```

address of x = 0x10011d5c
foo called from main: return value = 24
address of x = 0x10011d5c
foo called from g: return value = 25

```

If you want to ensure that each instance of function defined as inline is treated as a separate function, you can use the `static` specifier in the function definition in each translation unit, or compile with the `-qstaticinline` option. Note, however, that static inline functions are removed from name lookup during template instantiation, and are not found.

Related information

- “The static storage class specifier” on page 188
- “The extern storage class specifier” on page 188

End of C++ only

The `__cdecl` function specifier (C++ only)

z/OS only

You can use the `__cdecl` keyword to set linkage conventions for function calls in C++ applications. 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);

```

z/OS XL C++ allows the `__cdecl` keyword on member functions and nonmember functions. These functions can be static or nonstatic. 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;}
```

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.

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
}
```

If the function is redeclared, the linkage keyword must appear in the first declaration; otherwise an error message is issued. 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 the first declaration

```

Example of __cdecl use

The following example illustrates how you can use __cdecl to pass in a C parameter list from C++ code to a C function:

```

/*-----*/
/* C++ source file */
/*-----*/
//
// C++ Application: passing a C++ function pointer to a C function
//
#include <stdio.h>

// C++ function declared with C calling convention
void __cdecl callcxx() {
    printf(" I am a C++ function\n");
}

// declare a function pointer with __cdecl linkage
void (__cdecl *p1)();

// declare an extern C function,
// accepting a __cdecl function pointer
extern "C" {
    void CALLC(void (__cdecl *pp)());
}

// assign the function pointer to a __cdecl function
int main() {
    p1 = callcxx;

// call the C function with the __cdecl function pointer
    CALLC(p1);
}

/*-----*/
/* 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
}

```

Related information

- “Language linkage (C++ only)” on page 9

End of z/OS only

The `_Export` function specifier (C++ only)

z/OS only

C++ only

Use the `_Export` keyword with a function name to declare that it is to be exported (made available to other modules). You must define the function in the same translation unit in which you use the `_Export` keyword. For example:

```
int _Export anthony(float);
```

The above statement exports the function `anthony`, if you define the function within the translation unit.

The `_Export` keyword must immediately precede the function name. If the `_Export` keyword is repeated in a declaration, z/OS XL C++ issues a warning when you specify the `INFO(GEN)` option.

If you apply the `_Export` keyword to a class, the z/OS XL C++ compiler automatically exports all members of the class, whether static, public, private, or protected. However, if you want it to apply to individual class members, then you must apply it to each member that can be referenced. The following class definitions demonstrate this.

```
class A {
public:
    int iii() {
        printf("Hi from A::iii()\n");
        aaa();
        printf("Call to A::ccc() returned %c\n", ccc());
        return 88;
    }
    static void _Export sss();
protected:
    void _Export aaa();
private:
    char _Export ccc();
};

class _Export B {
public:
    int iii() {
        printf("Hi from B::iii()\n");
        aaa();
        printf("Call to B::ccc() returned %c\n", ccc());
        return 99;
    }
    static void sss();
protected:
    void _Export aaa();
private:
    char _Export ccc();
};
```

In the example below, both `X::Print()` and `X::GetNext()` will be exported.

```
class _Export X {
public:
    ...
    void static Print();
    int GetNext();
    ...
};
```

```

void X:: static Print() {
    ...
}
int X::GetNext() {
    ...
}

```

The above examples demonstrate that you can either export specific members of a class or the entire class itself. Note that the `_Export` keyword can be applied to class tags in nested class declarations.

Related information

- “External linkage” on page 8
- “#pragma export” on page 409

End of C++ only

End of z/OS only

Function return type specifiers

The result of a function is called its *return value* and the data type of the return value is called the *return type*.

C++ Every function declaration and definition must specify a return type, whether or not it actually returns a value.

C If a function declaration does not specify a return type, the compiler assumes an implicit return type of `int`. However, for conformance to C99, you should specify a return type for every function declaration and definition, whether or not the function returns `int`.

A function may be defined to return any type of value, except an array type or a function type; these exclusions must be handled by returning a pointer to the array or function. When a function does not return a value, `void` is the type specifier in the function declaration and definition.

A function cannot be declared as returning a data object having a `volatile` or `const` type, but it can return a pointer to a `volatile` or `const` object.

A function can have a return type that is a user-defined type. For example:

```

enum count {one, two, three};
enum count counter();

```

C The user-defined type may also be defined within the function declaration.

C++ The user-defined type may not be defined within the function declaration.

```

enum count{one, two, three} counter(); // legal in C
enum count{one, two, three} counter(); // error in C++

```

C++ References can also be used as return types for functions. The reference returns the lvalue of the object to which it refers.

Related information

- “Type specifiers” on page 49

Function return values

▶ **C** If a function is defined as having a return type of `void`, it should not return a value. ▶ **C++** In C++, a function which is defined as having a return type of `void`, or is a constructor or destructor, *must* not return a value.

▶ **C** If a function is defined as having a return type other than `void`, it should return a value. Under compilation for strict C99 conformance, a function defined with a return type *must* include an expression containing the value to be returned.

▶ **C++** A function defined with a return type *must* include an expression containing the value to be returned.

When a function returns a value, the value is returned via a return statement to the caller of the function, after being implicitly converted to the return type of the function in which it is defined. 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 variable `answer` is initialized with the `int` value 30. The type of the returned expression is checked against the returned type. All standard and user-defined conversions are performed as necessary.

Each time a function is called, new copies of its variables with automatic storage are created. Because the storage for these automatic variables may be reused after the function has terminated, a pointer or reference to an automatic variable should not be returned. ▶ **C++** If a class object is returned, a temporary object may be created if the class has copy constructors or a destructor.

Related information

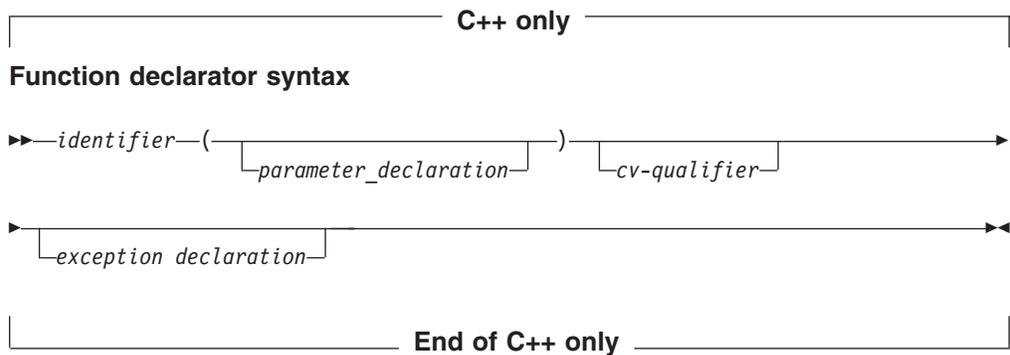
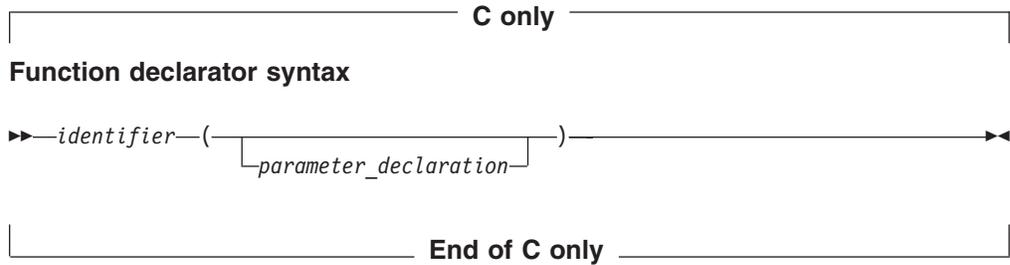
- “The return statement” on page 175
- “Overloading assignments (C++ only)” on page 232
- “Overloading subscripting (C++ only)” on page 234
- “The auto storage class specifier” on page 44

Function declarators

Function declarators consist of the following elements:

- An *identifier*, or name
- Parameter declarations, which specify the parameters that can be passed to the function in a function call
- ▶ **C++** Exception declarations, which include throw expressions; exception specifications are described in Chapter 16, “Exception handling (C++ only),” on page 353.

- C++ The type qualifiers `const` and `volatile`, which are used only in class member functions; they are described in “Constant and volatile member functions (C++ only)” on page 254.



Related information

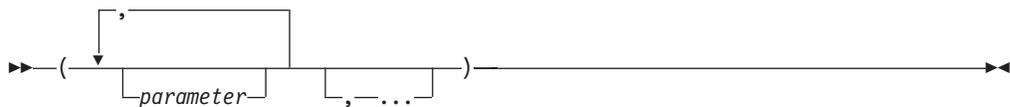
- “Default arguments in C++ functions” on page 211

Parameter declarations

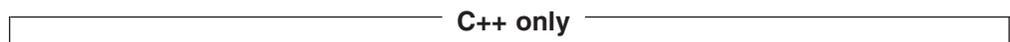
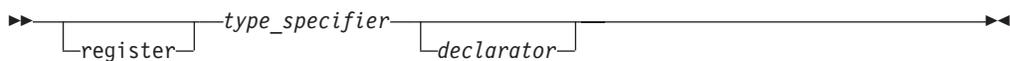
The function declarator includes the list of parameters that can be passed to the function when it is called by another function, or by itself.

- C++ In C++, the parameter list of a function is referred to as its *signature*. The name and signature of a function uniquely identify it. As the word itself suggests, the function signature is used by the compiler to distinguish among the different instances of overloaded functions.

Function parameter declaration syntax



parameter



An empty argument list in a function declaration or definition indicates a function

that takes no arguments. To explicitly indicate that a function does not take any arguments, you can declare the function in two ways: with an empty parameter list, or with the keyword `void`:

```
int f(void);  
int f();
```

End of C++ only

C only

An empty argument list in a function *definition* indicates that a function that takes no arguments. An empty argument list in a function *declaration* indicates that a function may take any number or type of arguments. Thus,

```
int f()  
{  
...  
}
```

indicates that function `f` takes no arguments. However,

```
int f();
```

simply indicates that the number and type of parameters is not known. To explicitly indicate that a function does not take any arguments, you should define the function with the keyword `void`.

End of C only

An ellipsis at the end of the parameter specifications is used to specify that a function has a variable number of parameters. The number of parameters is equal to, or greater than, the number of parameter specifications.

```
int f(int, ...);
```

C++ The comma before the ellipsis is optional. In addition, a parameter declaration is not required before the ellipsis.

C At least one parameter declaration, as well as a comma before the ellipsis, are both required in C.

Related information

- “The void type” on page 54
- “Type specifiers” on page 49
- “Type qualifiers” on page 67
- “Exception specifications (C++ only)” on page 364

Parameter types

In a function *declaration*, or prototype, the type of each parameter must be

specified. **C++** In the function *definition*, the type of each parameter must also be specified. **C** In the function *definition*, if the type of a parameter is not specified, it is assumed to be `int`.

A variable of a user-defined type may be declared in a parameter declaration, as in the following example, in which `x` is declared for the first time:

```
struct X { int i; };
void print(struct X x);
```

C The user-defined type can also be defined within the parameter declaration. **C++** The user-defined type can not be defined within the parameter declaration.

```
void print(struct X { int i; } x); // legal in C
void print(struct X { int i; } x); // error in C++
```

Parameter names

In a function *definition*, each parameter must have an identifier. In a function *declaration*, or prototype, specifying an identifier is optional. Thus, the following example is legal in a function declaration:

```
int func(int, long);
```

C++ only

The following constraints apply to the use of parameter names in function declarations:

- Two parameters cannot have the same name within a single declaration.
- If a parameter name is the same as a name outside the function, the name outside the function is hidden and cannot be used in the parameter declaration. In the following example, the third parameter name `intersects` is meant to have enumeration type `subway_line`, but this name is hidden by the name of the first parameter. The declaration of the function `subway()` causes a compile-time error because `subway_line` is not a valid type name because the first parameter name `subway_line` hides the namespace scope enum type and cannot be used again in the second parameter.

```
enum subway_line {yonge,
university, spadina, bloor};
int subway(char * subway_line, int stations,
subway_line intersects);
```

End of C++ only

Static array indices in function parameter declarations (C only)

Except in certain contexts, 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 that the array has previously been declared. An array type in the parameter list of a function is also converted to the corresponding pointer type. Information about the size of the argument array is lost when the array is accessed from within the function body.

To preserve this information, which is useful for optimization, you may declare the index of the argument array using the `static` keyword. The constant expression specifies the minimum pointer size that can be used as an assumption for optimizations. This particular usage of the `static` keyword is highly prescribed. The keyword may only appear in the outermost array type derivation and only in function parameter declarations. If the caller of the function does not abide by these restrictions, the behavior is undefined.

The following examples show how the feature can be used.

```
void foo(int arr [static 10]); /* arr points to the first of at least
                             10 ints */
void foo(int arr [const 10]); /* arr is a const pointer */
```

```

void foo(int arr [static const i]); /* arr points to at least i ints;
                                   i is computed at run time.      */
void foo(int arr [const static i]); /* alternate syntax to previous example */
void foo(int arr [const]);          /* const pointer to int             */

```

Related information

- “The static storage class specifier” on page 44
- “Arrays” on page 84
- “Array subscripting operator []” on page 138

Function attributes

IBM extension

Function attributes are extensions implemented to enhance the portability of programs developed with GNU C. Specifiable attributes for functions provide explicit ways to help the compiler optimize function calls and to instruct it to check more aspects of the code. Others provide additional functionality.

A function attribute is specified with the keyword `__attribute__` followed by the attribute name and any additional arguments the attribute name requires. A function `__attribute__` specification is included in the declaration or definition of a function. The syntax takes the following forms:

Function attribute syntax: function declaration

►► *function declarator* `__attribute__` _____►

►► `((` *attribute_name* `)` `;` _____►►

_____►

_____►

_____►

C only

Function attribute syntax: function definition

►► `__attribute__` `((` *attribute name* `)` `—function_declarator` _____►

_____►

►► `{—function body—}` _____►►

_____►

_____►

End of C only

The function attribute in a function declaration is always placed after the declarator, including the parenthesized parameter declaration:

```

/* Specify the attribute on a function prototype declaration */
void f(int i, int j) __attribute__((individual_attribute_name));
void f(int i, int j) { }

```

C only

Due to ambiguities in parsing old-style parameter declarations, a function definition must have the attribute specification *precede* the declarator:

```
int __attribute__((individual_attribute_name)) foo(int i) { }
```

End of C only

A function attribute specification using the form `__attribute_name__` (that is, the attribute name with double underscore characters leading and trailing) reduces the likelihood of a name conflict with a macro of the same name.

 Currently, only the `armode` function attribute is supported.

Related information

- “Variable attributes” on page 100

The `armode` | `noarmode` function attribute (C only)

z/OS only

For use with the METAL compiler option, the `armode` function attribute allows you to specify whether or not a given function is to operate in access-register (AR) mode. AR mode allows a C function to access multiple additional data spaces, and manipulate more data in memory.

armode function attribute syntax

```
►► __attribute__((

|          |
|----------|
| armode   |
| noarmode |

))◄◄
```

Functions in AR mode can call functions not in AR mode, and vice versa.

The following example declares the function `foo` to be in AR mode:

```
void foo() __attribute__((armode));
```

The attribute overrides the default setting of the `ARMODE` compiler option for the specified function. Note that this attribute is only supported when the METAL compiler option is in effect.

Related information

- “The `armode` | `noarmode` type attribute (C only)” on page 75
- “The `__far` type qualifier (C only)” on page 70
- The `ARMODE` and `METAL` options in the *z/OS XL C/C++ User’s Guide*

End of z/OS only

End of IBM extension

The main() function

When a program begins running, the system calls the function `main`, which marks the entry point of the program. By default, `main` has the storage class `extern`. Every program must have one function named `main`, and the following constraints apply:

- No other function in the program can be called `main`.
- `main` cannot be defined as `inline` or `static`.
-  `main` cannot be called from within a program.
-  The address of `main` cannot be taken.
-  The `main` function cannot be overloaded.

The function `main` can be defined with or without parameters, using any of the following forms:

```
int main (void)
int main ( )
int main(int argc, char *argv[])
int main (int argc, char ** argv)
```

Although any name can be given to these parameters, they are usually referred to as `argc` and `argv`. The first parameter, `argc` (argument count) is an integer that indicates how many arguments were entered on the command line when the program was started. The second parameter, `argv` (argument vector), is an array of pointers to arrays of character objects. The array objects are null-terminated strings, representing the arguments that were entered on the command line when the program was started.

The first element of the array, `argv[0]`, is a pointer to the character array that contains the program name or invocation name of the program that is being run from the command line. `argv[1]` indicates the first argument passed to the program, `argv[2]` the second argument, and so on. The following example 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]);
    printf("\n");
}
```

Invoking this program from a command line with the following:

```
backward string1 string2
```

gives the following output:

```
string2 string1
```

The arguments `argc` and `argv` would contain the following values:

Object	Value
<code>argc</code>	3
<code>argv[0]</code>	pointer to string "backward"
<code>argv[1]</code>	pointer to string "string1"
<code>argv[2]</code>	pointer to string "string2"

Object	Value
argv[3]	NULL

Note: See *z/OS XL C/C++ Programming Guide* for details about receiving the parameter list (argv) in C main, preparing your **main** function to receive parameters, and on C and C++ parameter passing considerations.

Related information

- “The extern storage class specifier” on page 46
- “The inline function specifier” on page 190
- “The static storage class specifier” on page 44
- “Function calls” on page 207

Command-line arguments

z/OS only

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

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

Under z/OS batch

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

Under IMS

argc	Returns 1
argv[0]	Is a null pointer

Under CICS

argc	Returns 1
argv[0]	Returns the transaction ID

Under TSO command

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

Under TSO call

Without the ASIS option:

argc	Returns the number of strings in the argument line
argv	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 z/OS UNIX System Services 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. z/OS XL 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')
```

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

Related information

- "Function calls"
- "Type specifiers" on page 49
- "Identifiers" on page 15
- "Block statements" on page 163

End of z/OS only

Function calls

Once a function has been declared and defined, it can be *called* from anywhere within the program: from within the `main` function, from another function, and even from itself. Calling the function involves specifying the function name, followed by the function call operator and any data values the function expects to receive. These values are the *arguments* for the parameters defined for the function, and the process just described is called *passing arguments* to the function.

► **C++** A function may not be called if it has not already been declared.

Passing arguments can be done in two ways:

- Pass by value, which copies the *value* of an argument to the corresponding parameter in the called function
- Pass by reference, which passes the *address* of an argument to the corresponding parameter in the called function

C++ only

If a class 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 object 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.
- A copy constructor cannot be generated for that class.

End of C++ only

Related information

- “Function argument conversions” on page 110
- “Function call expressions” on page 116
- “Constructors (C++ only)” on page 301

Pass by value

When you use *pass-by-value*, the compiler copies the value of an argument in a calling function to a corresponding non-pointer or non-reference parameter in the called function definition. The parameter in the called function is initialized with the value of the passed argument. As long as the parameter has not been declared as constant, the value of the parameter can be changed, but the changes are only performed within the scope of the called function only; they have no effect on the value of the argument in the calling function.

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.

```
/**
 ** This example illustrates calling a function by value
 **/

#include <stdio.h>

void func (int a, int b)
{
    a += b;
    printf("In func, a = %d    b = %d\n", a, b);
}

int main(void)
{
    int x = 5, y = 7;
    func(x, y);
    printf("In main, x = %d    y = %d\n", x, y);
    return 0;
}
```

The output of the program is:

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

Pass by reference

Passing by *reference* refers to a method of passing the address of an argument in the calling function to a corresponding parameter in the called function.

C In C, the corresponding parameter in the called function must be declared as a pointer type. **C++** In C++, the corresponding parameter can be declared as any reference type, not just a pointer type.

In this way, the value of the argument in the calling function can be modified by the called function.

The following example shows how arguments are passed by reference. In C++, the reference parameters are initialized with the actual arguments when the function is called. In C, the pointer parameters are initialized with pointer values when the function is called.

CCNX06A

```
_____ C++ only _____  
  
#include <stdio.h>  
  
void swapnum(int &i, int &j) {  
    int temp = i;  
    i = j;  
    j = temp;  
}  
  
int main(void) {  
    int a = 10;  
    int b = 20;  
  
    swapnum(a, b);  
    printf("A is %d and B is %d\n", a, b);  
    return 0;  
}  
_____ End of C++ only _____
```

```
_____ C only _____  
  
#include <stdio.h>  
  
void swapnum(int *i, int *j) {  
    int temp = *i;  
    *i = *j;  
    *j = temp;  
}  
  
int main(void) {  
    int a = 10;  
    int b = 20;  
  
    swapnum(&a, &b);  
    printf("A is %d and B is %d\n", a, b);  
    return 0;  
}  
_____ End of C only _____
```

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

C++ only

In order to modify a reference that is const-qualified, you must cast away its constness with the `const_cast` operator. The following example demonstrates this:

```
#include <iostream>
using namespace std;

void f(const int& x) {
    int* y = const_cast<int>(&x);
    (*y)++;
}

int main() {
    int a = 5;
    f(a);
    cout << a << endl;
}
```

This example outputs 6.

End of C++ only

Related information

- “References (C++ only)” on page 87
- “The `const_cast` operator (C++ only)” on page 148

Allocation and deallocation functions (C++ only)

You may define your own `new` operator or allocation function as a class member function or a global namespace function with the following restrictions:

- The first parameter must be of type `std::size_t`. It cannot have a default parameter.
- The return type must be of type `void*`.
- Your allocation function may be a template function. Neither the first parameter nor the return type may depend on a template parameter.
- If you declare your allocation function with the empty exception specification `throw()`, your allocation function must return a null pointer if your function fails. Otherwise, your function must throw an exception of type `std::bad_alloc` or a class derived from `std::bad_alloc` if your function fails.

You may define your own `delete` operator or deallocation function as a class member function or a global namespace function with the following restrictions:

- The first parameter must be of type `void*`.
- The return type must be of type `void`.
- Your deallocation function may be a template function. Neither the first parameter nor the return type may depend on a template parameter.

The following example defines replacement functions for global namespace `new` and `delete`:

```
#include <cstdio>
#include <cstdlib>

using namespace std;

void* operator new(size_t sz) {
```

```

    printf("operator new with %d bytes\n", sz);
    void* p = malloc(sz);
    if (p == 0) printf("Memory error\n");
    return p;
}

void operator delete(void* p) {
    if (p == 0) printf ("Deleting a null pointer\n");
    else {
        printf("delete object\n");
        free(p);
    }
}

struct A {
    const char* data;
    A() : data("Text String") { printf("Constructor of S\n"); }
    ~A() { printf("Destructor of S\n"); }
};

int main() {
    A* ap1 = new A;
    delete ap1;

    printf("Array of size 2:\n");
    A* ap2 = new A[2];
    delete[] ap2;
}

```

The following is the output of the above example:

```

operator
new with 40 bytes
operator new with 33 bytes
operator new with 4 bytes
Constructor of S
Destructor of S
delete object
Array of size 2:
operator new with 16 bytes
Constructor of S
Constructor of S
Destructor of S
Destructor of S
delete object

```

Related information

- “new expressions (C++ only)” on page 151

Default arguments in C++ functions

You can provide default values for function parameters. For example:

CCNX06B

```

#include <iostream>
using namespace std;

int a = 1;
int f(int a) { return a; }
int g(int x = f(a)) { return x; }

int h() {
    a = 2;
    {
        int a = 3;
    }
}

```

```

        return g();
    }
}

int main() {
    cout << h() << endl;
}

```

This example prints 2 to standard output, because the `a` referred to in the declaration of `g()` is the one at file scope, which has the value 2 when `g()` is called.

The default argument must be implicitly convertible to the parameter 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 will produce an error. The type of a function is not affected by arguments with default values.

The following example shows that default arguments are not considered part of a function's 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. Function `f` can be called without an explicit argument, but the pointer `badpointer` cannot be defined 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.

```

In this example, function `f3` has a return type `int`, and takes an `int` argument with a default value that is the value returned from function `f2`:

```

const int j = 5;
int f3( int x = f2(j) );

```

Related information

- “Pointers to functions” on page 214

Restrictions on default arguments

Of the operators, only the function call operator and the operator `new` can have default arguments when they are overloaded.

Parameters with default arguments must be the trailing parameters in the function declaration parameter 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 a default argument has been given in a declaration or definition, you cannot redefine that argument, even to the same value. However, you can add default arguments 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. Any parameters in the parameter list following a default argument value must have a default argument value specified in this or a previous declaration of the function.

You cannot use local variables in default argument expressions. For example, the 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" cannot be used here
    void h(int d=b); // Local variable "b" cannot be used here
}

```

Related information

- “Function call expressions” on page 116
- “new expressions (C++ only)” on page 151

Evaluation of default arguments

When a function defined with default arguments is called with trailing arguments missing, the default expressions are evaluated. 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

```

Default arguments are checked against the function declaration and evaluated when the function is called. The order of evaluation of default arguments is undefined. Default argument expressions cannot use other parameters of the function. For example:

```

int f(int q = 3, int r = q); // error

```

The argument `r` cannot be initialized with the value of the argument `q` because the value of `q` may not be known when it is assigned to `r`. If the above function declaration is rewritten:

```

int q=5;
int f(int q = 3, int r = q); // error

```

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 type `D` is interpreted within the function declaration as the name of an integer. The type `D` is hidden by 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, the nonstatic member `a` cannot be used as an initializer because `a` does not exist until an object of class `X` is constructed. You can use the

static member `b` as an initializer because `b` is created independently of any objects of class `X`. You can declare the member `b` after its use as a default argument because the default values are not analyzed until after the final bracket `}` of the class declaration.

```
class X
{
    int a;
    f(int z = a) ; // error
    g(int z = b) ; // valid
    static int b;
};
```

Pointers to functions

A pointer to a function points to the address of the executable code of the function. 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.

z/OS For z/OS XL C/C++, use the `__cdecl` keyword to declare a pointer to a function as a C linkage. For more information, refer to “The `__cdecl` function specifier (C++ only)” on page 194.

The type of a pointer to a function is based on both the return type and parameter types of the function.

A declaration of a pointer to a function must have the pointer name in parentheses. The function call operator `()` has a higher precedence than the dereference operator `*`. Without them, the compiler interprets the statement as a function that returns a pointer to a specified return type. For example:

```
int *f(int a);          /* function f returning an int*          */
int (*g)(int a);       /* pointer g to a function returning an int          */
char (*h)(int, int) /* h is a function
                    that takes two integer parameters and returns char */
```

In the first declaration, `f` is interpreted as a function that takes an `int` as argument, and returns a pointer to an `int`. In the second declaration, `g` is interpreted as a pointer to a function that takes an `int` argument and that returns an `int`. Under z/OS XL 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.

The following example illustrates the correct and incorrect uses of function pointers under z/OS XL 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 z/OS, linkage compatibility affects all C library functions that accept a function pointer as a parameter.

Related information

- “Language linkage (C++ only)” on page 9
- “Pointers” on page 80
- “Pointer conversions” on page 107
- “The extern storage class specifier” on page 188

Chapter 9. Namespaces (C++ only)

A *namespace* is an optionally named scope. You declare names inside a namespace as you would for a class or an enumeration. You can access names declared inside a namespace the same way you access a nested class name by using the scope resolution (::) operator. However namespaces do not have the additional features of classes or enumerations. The primary purpose of the namespace is to add an additional identifier (the name of the namespace) to a name.

Related information

- “Scope resolution operator :: (C++ only)” on page 116

Defining namespaces (C++ only)

In order to uniquely identify a namespace, use the **namespace** keyword.

Namespace syntax

```
▶ namespace identifier { namespace_body } ▶
```

The *identifier* in an original namespace definition is the name of the namespace. The identifier may not be previously defined in the declarative region in which the original namespace definition appears, except in the case of extending namespace. If an identifier is not used, the namespace is an *unnamed namespace*.

Related information

- “Unnamed namespaces (C++ only)” on page 219

Declaring namespaces (C++ only)

The identifier used for a namespace name should be unique. It should not be used previously as a global identifier.

```
namespace Raymond {  
    // namespace body here...  
}
```

In this example, Raymond is the identifier of the namespace. If you intend to access a namespace’s elements, the namespace’s identifier must be known in all translation units.

Related information

- “File/global scope” on page 3

Creating a namespace alias (C++ only)

An alternate name can be used in order to refer to a specific namespace identifier.

```
namespace INTERNATIONAL_BUSINESS_MACHINES {  
    void f();  
}  
  
namespace IBM = INTERNATIONAL_BUSINESS_MACHINES;
```

In this example, the IBM identifier is an alias for INTERNATIONAL_BUSINESS_MACHINES. This is useful for referring to long namespace identifiers.

If a namespace name or alias is declared as the name of any other entity in the same declarative region, a compiler error will result. Also, if a namespace name defined at global scope is declared as the name of any other entity in any global scope of the program, a compiler error will result.

Related information

- “File/global scope” on page 3

Creating an alias for a nested namespace (C++ only)

Namespace definitions hold declarations. Since a namespace definition is a declaration itself, namespace definitions can be nested.

An alias can also be applied to a nested namespace.

```
namespace INTERNATIONAL_BUSINESS_MACHINES {
    int j;
    namespace NESTED_IBM_PRODUCT {
        void a() { j++; }
        int j;
        void b() { j++; }
    }
}
namespace NIBM = INTERNATIONAL_BUSINESS_MACHINES::NESTED_IBM_PRODUCT
```

In this example, the NIBM identifier is an alias for the namespace NESTED_IBM_PRODUCT. This namespace is nested within the INTERNATIONAL_BUSINESS_MACHINES namespace.

Related information

- “Creating a namespace alias (C++ only)” on page 217

Extending namespaces (C++ only)

Namespaces are extensible. You can add subsequent declarations to a previously defined namespace. Extensions may appear in files separate from or attached to the original namespace definition. For example:

```
namespace X { // namespace definition
    int a;
    int b;
}

namespace X { // namespace extension
    int c;
    int d;
}

namespace Y { // equivalent to namespace X
    int a;
    int b;
    int c;
    int d;
}
```

In this example, namespace X is defined with a and b and later extended with c and d. namespace X now contains all four members. You may also declare all of the

required members within one namespace. This method is represented by namespace Y. This namespace contains a, b, c, and d.

Namespaces and overloading (C++ only)

You can overload functions across namespaces. For example:

```
// Original X.h:
f(int);

// Original Y.h:
f(char);

// Original program.c:
#include "X.h"
#include "Y.h"

void z()
{
    f('a'); // calls f(char) from Y.h
}
```

Namespaces can be introduced to the previous example without drastically changing the source code.

```
// New X.h:
namespace X {
    f(int);
}

// New Y.h:
namespace Y {
    f(char);
}

// New program.c:
#include "X.h"
#include "Y.h"

using namespace X;
using namespace Y;

void z()
{
    f('a'); // calls f() from Y.h
}
```

In program.c, function void z() calls function f(), which is a member of namespace Y. If you place the using directives in the header files, the source code for program.c remains unchanged.

Related information

- Chapter 10, “Overloading (C++ only),” on page 225

Unnamed namespaces (C++ only)

A namespace with no identifier before an opening brace produces an *unnamed namespace*. Each translation unit may contain its own unique unnamed namespace. The following example demonstrates how unnamed namespaces are useful.

```
#include <iostream>

using namespace std;
```

```

namespace {
    const int i = 4;
    int variable;
}

int main()
{
    cout << i << endl;
    variable = 100;
    return 0;
}

```

In the previous example, the unnamed namespace permits access to `i` and `variable` without using a scope resolution operator.

The following example illustrates an improper use of unnamed namespaces.

```

#include <iostream>

using namespace std;

namespace {
    const int i = 4;
}

int i = 2;

int main()
{
    cout << i << endl; // error
    return 0;
}

```

Inside `main`, `i` causes an error because the compiler cannot distinguish between the global name and the unnamed namespace member with the same name. In order for the previous example to work, the namespace must be uniquely identified with an identifier and `i` must specify the namespace it is using.

You can extend an unnamed namespace within the same translation unit. For example:

```

#include <iostream>

using namespace std;

namespace {
    int variable;
    void funct (int);
}

namespace {
    void funct (int i) { cout << i << endl; }
}

int main()
{
    funct(variable);
    return 0;
}

```

both the prototype and definition for `funct` are members of the same unnamed namespace.

Note: Items defined in an unnamed namespace have internal linkage. Rather than using the keyword `static` to define items with internal linkage, define them in an unnamed namespace instead.

Related information

- “Program linkage” on page 7
- “Internal linkage” on page 7

Namespace member definitions (C++ only)

A namespace can define its own members within itself or externally using explicit qualification. The following is an example of a namespace defining a member internally:

```
namespace A {
    void b() { /* definition */ }
}
```

Within namespace A member `void b()` is defined internally.

A namespace can also define its members externally using explicit qualification on the name being defined. The entity being defined must already be declared in the namespace and the definition must appear after the point of declaration in a namespace that encloses the declaration’s namespace.

The following is an example of a namespace defining a member externally:

```
namespace A {
    namespace B {
        void f();
    }
    void B::f() { /* defined outside of B */ }
}
```

In this example, function `f()` is declared within namespace B and defined (outside B) in A.

Namespaces and friends (C++ only)

Every name first declared in a namespace is a member of that namespace. If a friend declaration in a non-local class first declares a class or function, the friend class or function is a member of the innermost enclosing namespace.

The following is an example of this structure:

```
// f has not yet been defined
void z(int);
namespace A {
    class X {
        friend void f(X); // A::f is a friend
    };
    // A::f is not visible here
    X x;
    void f(X) { /* definition */ } // f() is defined and known to be a friend
}

using A::x;

void z()
```

```

{
  A::f(x);    // OK
  A::X::f(x); // error: f is not a member of A::X
}

```

In this example, function `f()` can only be called through namespace `A` using the call `A::f(s)`. Attempting to call function `f()` through class `X` using the `A::X::f(x)`; call results in a compiler error. Since the friend declaration first occurs in a non-local class, the friend function is a member of the innermost enclosing namespace and may only be accessed through that namespace.

Related information

- “Friends (C++ only)” on page 267

The using directive (C++ only)

A `using` directive provides access to all namespace qualifiers and the scope operator. This is accomplished by applying the `using` keyword to a namespace identifier.

Using directive syntax

```

▶▶ using namespace name; ◀◀

```

The *name* must be a previously defined namespace. The `using` directive may be applied at the global and local scope but not the class scope. Local scope takes precedence over global scope by hiding similar declarations.

If a scope contains a `using` directive that nominates a second namespace and that second namespace contains another `using` directive, the `using` directive from the second namespace will act as if it resides within the first scope.

```

namespace A {
  int i;
}
namespace B {
  int i;
  using namespace A;
}
void f()
{
  using namespace B;
  i = 7; // error
}

```

In this example, attempting to initialize `i` within function `f()` causes a compiler error, because function `f()` cannot know which `i` to call; `i` from namespace `A`, or `i` from namespace `B`.

Related information

- “The using declaration and class members (C++ only)” on page 280

The using declaration and namespaces (C++ only)

A `using` declaration provides access to a specific namespace member. This is accomplished by applying the `using` keyword to a namespace name with its corresponding namespace member.

Using declaration syntax

► `using namespace :: member` ◄

In this syntax diagram, the qualifier name follows the using declaration and the *member* follows the qualifier name. For the declaration to work, the member must be declared inside the given namespace. For example:

```
namespace A {  
    int i;  
    int k;  
    void f;  
    void g;  
}
```

```
using A::k
```

In this example, the using declaration is followed by A, the name of namespace A, which is then followed by the scope operator (::), and k. This format allows k to be accessed outside of namespace A through a using declaration. After issuing a using declaration, any extension made to that specific namespace will not be known at the point at which the using declaration occurs.

Overloaded versions of a given function must be included in the namespace prior to that given function's declaration. A using declaration may appear at namespace, block and class scope.

Related information

- “The using declaration and class members (C++ only)” on page 280

Explicit access (C++ only)

To explicitly qualify a member of a namespace, use the namespace identifier with a :: scope resolution operator.

Explicit access qualification syntax

► `namespace_name :: member` ◄

For example:

```
namespace VENDITTI {  
    void j()  
};
```

```
VENDITTI::j();
```

In this example, the scope resolution operator provides access to the function j held within namespace VENDITTI. The scope resolution operator :: is used to access identifiers in both global and local namespaces. Any identifier in an application can be accessed with sufficient qualification. Explicit access cannot be applied to an unnamed namespace.

Related information

- “Scope resolution operator :: (C++ only)” on page 116

Chapter 10. Overloading (C++ only)

If you specify more than one definition for a function name or an operator in the same scope, you have *overloaded* that function name or operator. Overloaded functions and operators are described in “Overloading functions (C++ only)” and “Overloading operators (C++ only)” on page 227, respectively.

An *overloaded declaration* is a declaration that had been declared with the same name as a previously declared declaration in the same scope, except that both declarations have different types.

If you call an overloaded function name or operator, the compiler determines the most appropriate definition to use by comparing the argument types you used to call the function or operator with the parameter types specified in the definitions. The process of selecting the most appropriate overloaded function or operator is called *overload resolution*, as described in “Overload resolution (C++ only)” on page 236.

Overloading functions (C++ only)

You overload a function name *f* by declaring more than one function with the name *f* in the same scope. The declarations of *f* must differ from each other by the types and/or the number of arguments in the argument list. When you call an overloaded function named *f*, the correct function is selected by comparing the argument list of the function call with the parameter list of each of the overloaded candidate functions with the name *f*. A *candidate function* is a function that can be called based on the context of the call of the overloaded function name.

Consider a function `print`, which displays an `int`. As shown in the following example, you can overload the function `print` 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:

```
#include <iostream>
using namespace std;

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;
}

int main() {
    print(10);
    print(10.10);
    print("ten");
}
```

The following is the output of the above example:

```
Here is int 10
Here is float 10.1
Here is char* ten
```

Related information

- “Restrictions on overloaded functions (C++ only)”
- “Derivation (C++ only)” on page 275

Restrictions on overloaded functions (C++ only)

You cannot overload the following function declarations if they appear in the same scope. Note that this list applies only to explicitly declared functions and those that have been introduced through `using` declarations:

- Function declarations that differ only by return type. For example, you cannot declare the following declarations:

```
int f();
float f();
```

- Member function declarations that have the same name and the same parameter types, but one of these declarations is a static member function declaration. For example, you cannot declare the following two member function declarations of `f()`:

```
struct A {
    static int f();
    int f();
};
```

- Member function template declarations that have the same name, the same parameter types, and the same template parameter lists, but one of these declarations is a static template member function declaration.
- Function declarations that have equivalent parameter declarations. These declarations are not allowed because they would be declaring the same function.
- Function declarations with parameters that differ only by the use of typedef names that represent the same type. Note that a typedef is a synonym for another type, not a separate type. For example, the following two declarations of `f()` are declarations of the same function:

```
typedef int I;
void f(float, int);
void f(float, I);
```

- Function declarations with parameters that differ only because one is a pointer and the other is an array. For example, the following are declarations of the same function:

```
f(char*);
f(char[10]);
```

The first array dimension is insignificant when differentiating parameters; all other array dimensions are significant. For example, the following are declarations of the same function:

```
g(char(*)[20]);
g(char[5][20]);
```

The following two declarations are *not* equivalent:

```
g(char(*)[20]);
g(char(*)[40]);
```

- Function declarations with parameters that differ only because one is a function type and the other is a pointer to a function of the same type. For example, the following are declarations of the same function:

```
void f(int(float));
void f(int (*)(float));
```

- Function declarations with parameters that differ only because of cv-qualifiers `const`, `volatile`, and `restrict`. This restriction only applies if any of these

qualifiers appears at the outermost level of a parameter type specification. For example, the following are declarations of the same function:

```
int f(int);
int f(const int);
int f(volatile int);
```

Note that you can differentiate parameters with `const`, `volatile` and `restrict` qualifiers if you apply them *within* a parameter type specification. For example, the following declarations are *not* equivalent because `const` and `volatile` qualify `int`, rather than `*`, and thus are not at the outermost level of the parameter type specification.

```
void g(int*);
void g(const int*);
void g(volatile int*);
```

The following declarations are also not equivalent:

```
void g(float&);
void g(const float&);
void g(volatile float&);
```

- Function declarations with parameters that differ only because their default arguments differ. For example, the following are declarations of the same function:

```
void f(int);
void f(int i = 10);
```

- Multiple functions with extern "C" language-linkage and the same name, regardless of whether their parameter lists are different.

Related information

- "The using declaration and namespaces (C++ only)" on page 222
- "typedef definitions" on page 65
- "Type qualifiers" on page 67
- "Language linkage (C++ only)" on page 9

Overloading operators (C++ only)

You can redefine or overload the function of most built-in operators in C++. These operators can be overloaded globally or on a class-by-class basis. Overloaded operators are implemented as functions and can be member functions or global functions.

An overloaded operator is called an *operator function*. You declare an operator function with the keyword `operator` preceding the operator. Overloaded operators are distinct from overloaded functions, but like overloaded functions, they are distinguished by the number and types of operands used with the operator.

Consider the standard `+` (plus) operator. When this operator is used with operands of different standard types, the operators have slightly different meanings. For example, the addition of two integers is not implemented 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 they are applied to class types. In the following example, a class called `complex` is defined to model complex numbers, and the `+` (plus) operator is redefined in this class to add two complex numbers.

CCNX12B

```
// This example illustrates overloading the plus (+) operator.

#include <iostream>
using namespace std;

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;
}

int main()
{
    complx x(4,4);
    complx y(6,6);
    complx z = x + y; // calls complx::operator+()
}
```

You can overload any of the following operators:

+	-	*	/	%	^	&		~
!	=	<	>	+=	--	*=	/=	%=
^=	&=	=	<<	>>	<<=	>>=	==	!=
<=	>=	&&		++	--	,	->*	->
()	[]	new	delete	new[]	delete[]			

where () is the function call operator and [] is the subscript operator.

You can overload both the unary and binary forms of the following operators:

+	-	*	&
---	---	---	---

You cannot overload the following operators:

.	.*	::	?:
---	----	----	----

You cannot overload the preprocessor symbols # and ##.

An operator function can be either a nonstatic member function, or a nonmember function with at least one parameter that has class, reference to class, enumeration, or reference to enumeration type.

You cannot change the precedence, grouping, or the number of operands of an operator.

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`, `delete`, `new[]`, and `delete[]` do not follow the general rules described in this section.

All operators except the `=` operator are inherited.

Overloading unary operators (C++ only)

You overload a unary operator with either a nonstatic member function that has no parameters, or a nonmember function that has one parameter. Suppose a unary operator `@` is called with the statement `@t`, where `t` is an object of type `T`. A nonstatic member function that overloads this operator would have the following form:

```
return_type operator@()
```

A nonmember function that overloads the same operator would have the following form:

```
return_type operator@(T)
```

An overloaded unary operator may return any type.

The following example overloads the `!` operator:

```
#include <iostream>
using namespace std;

struct X { };

void operator!(X) {
    cout << "void operator!(X)" << endl;
}

struct Y {
    void operator!() {
        cout << "void Y::operator!()" << endl;
    }
};

struct Z { };

int main() {
    X ox; Y oy; Z oz;
    !ox;
    !oy;
    // !oz;
}
```

The following is the output of the above example:

```
void operator!(X)
void Y::operator!()
```

The operator function call `!ox` is interpreted as `operator!(X)`. The call `!oy` is interpreted as `Y::operator!()`. (The compiler would not allow `!oz` because the `!` operator has not been defined for class `Z`.)

Related information

- “Unary expressions” on page 118

Overloading increment and decrement operators (C++ only)

You overload the prefix increment operator `++` with either a nonmember function operator that has one argument of class type or a reference to class type, or with a member function operator that has no arguments.

In the following example, the increment operator is overloaded in both ways:

```
class X {
public:

    // member prefix ++x
    void operator++() { }
};

class Y { };

// non-member prefix ++y
void operator++(Y&) { }

int main() {
    X x;
    Y y;

    // calls x.operator++()
    ++x;

    // explicit call, like ++x
    x.operator++();

    // calls operator++(y)
    ++y;

    // explicit call, like ++y
    operator++(y);
}
```

The postfix increment operator `++` can be overloaded for a class type by declaring a nonmember function operator `operator++()` with two arguments, the first having class type and the second having 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:

```
class X {
public:

    // member postfix x++
    void operator++(int) { };
};

class Y { };

// nonmember postfix y++
void operator++(Y&, int) { };
```

```

int main() {
    X x;
    Y y;

    // calls x.operator++(0)
    // default argument of zero is supplied by compiler
    x++;
    // explicit call to member postfix x++
    x.operator++(0);

    // calls operator++(y, 0)
    y++;

    // explicit call to non-member postfix y++
    operator++(y, 0);
}

```

The prefix and postfix decrement operators follow the same rules as their increment counterparts.

Related information

- “Increment operator ++” on page 118
- “Decrement operator --” on page 119

Overloading binary operators (C++ only)

You overload a binary unary operator with either a nonstatic member function that has one parameter, or a nonmember function that has two parameters. Suppose a binary operator @ is called with the statement `t @ u`, where `t` is an object of type `T`, and `u` is an object of type `U`. A nonstatic member function that overloads this operator would have the following form:

```
return_type operator@(T)
```

A nonmember function that overloads the same operator would have the following form:

```
return_type operator@(T, U)
```

An overloaded binary operator may return any type.

The following example overloads the `*` operator:

```

struct X {

    // member binary operator
    void operator*(int) { }
};

// non-member binary operator
void operator*(X, float) { }

int main() {
    X x;
    int y = 10;
    float z = 10;

    x * y;
    x * z;
}

```

The call `x * y` is interpreted as `x.operator*(y)`. The call `x * z` is interpreted as `operator*(x, z)`.

Related information

- “Binary expressions” on page 127

Overloading assignments (C++ only)

You overload the assignment operator, `operator=`, with a nonstatic member function that has only one parameter. You cannot declare an overloaded assignment operator that is a nonmember function. The following example shows how you can overload the assignment operator for a particular class:

```
struct X {
    int data;
    X& operator=(X& a) { return a; }
    X& operator=(int a) {
        data = a;
        return *this;
    }
};

int main() {
    X x1, x2;
    x1 = x2;      // call x1.operator=(x2)
    x1 = 5;      // call x1.operator=(5)
}
```

The assignment `x1 = x2` calls the copy assignment operator `X& X::operator=(X&)`. The assignment `x1 = 5` calls the copy assignment operator `X& X::operator=(int)`. The compiler implicitly declares a copy assignment operator for a class if you do not define one yourself. Consequently, the copy assignment operator (`operator=`) of a derived class hides the copy assignment operator of its base class.

However, you can declare any copy assignment operator as virtual. The following example demonstrates this:

```
#include <iostream>
using namespace std;

struct A {
    A& operator=(char) {
        cout << "A& A::operator=(char)" << endl;
        return *this;
    }
    virtual A& operator=(const A&) {
        cout << "A& A::operator=(const A&)" << endl;
        return *this;
    }
};

struct B : A {
    B& operator=(char) {
        cout << "B& B::operator=(char)" << endl;
        return *this;
    }
    virtual B& operator=(const A&) {
        cout << "B& B::operator=(const A&)" << endl;
        return *this;
    }
};

struct C : B { };

int main() {
```

```

B b1;
B b2;
A* ap1 = &b1;
A* ap2 = &b1;
*ap1 = 'z';
*ap2 = b2;

C c1;
// c1 = 'z';
}

```

The following is the output of the above example:

```

A& A::operator=(char)
B& B::operator=(const A&)

```

The assignment `*ap1 = 'z'` calls `A& A::operator=(char)`. Because this operator has not been declared `virtual`, the compiler chooses the function based on the type of the pointer `ap1`. The assignment `*ap2 = b2` calls `B& B::operator=(const A&)`. Because this operator has been declared `virtual`, the compiler chooses the function based on the type of the object that the pointer `ap1` points to. The compiler would not allow the assignment `c1 = 'z'` because the implicitly declared copy assignment operator declared in class `C` hides `B& B::operator=(char)`.

Related information

- “Copy assignment operators (C++ only)” on page 317
- “Assignment operators” on page 128

Overloading function calls (C++ only)

The function call operator, when overloaded, does not modify how functions are called. Rather, it modifies how the operator is to be interpreted when applied to objects of a given type.

You overload the function call operator, `operator()`, with a nonstatic member function that has any number of parameters. If you overload a function call operator for a class its declaration will have the following form:

```
return_type operator()(parameter_list)
```

Unlike all other overloaded operators, you can provide default arguments and ellipses in the argument list for the function call operator.

The following example demonstrates how the compiler interprets function call operators:

```

struct A {
    void operator()(int a, char b, ...) { }
    void operator()(char c, int d = 20) { }
};

int main() {
    A a;
    a(5, 'z', 'a', 0);
    a('z');
    // a();
}

```

The function call `a(5, 'z', 'a', 0)` is interpreted as `a.operator()(5, 'z', 'a', 0)`. This calls `void A::operator()(int a, char b, ...)`. The function call `a('z')` is interpreted as `a.operator>('z')`. This calls `void A::operator()(char c, int d =`

20). The compiler would not allow the function call `a()` because its argument list does not match any function call parameter list defined in class `A`.

The following example demonstrates an overloaded function call operator:

```
class Point {
private:
    int x, y;
public:
    Point() : x(0), y(0) { }
    Point& operator()(int dx, int dy) {
        x += dx;
        y += dy;
        return *this;
    }
};

int main() {
    Point pt;

    // Offset this coordinate x with 3 points
    // and coordinate y with 2 points.
    pt(3, 2);
}
```

The above example reinterprets the function call operator for objects of class `Point`. If you treat an object of `Point` like a function and pass it two integer arguments, the function call operator will add the values of the arguments you passed to `Point::x` and `Point::y` respectively.

Related information

- “Function call expressions” on page 116

Overloading subscripting (C++ only)

You overload `operator[]` with a nonstatic member function that has only one parameter. The following example is a simple array class that has an overloaded subscripting operator. The overloaded subscripting operator throws an exception if you try to access the array outside of its specified bounds:

```
#include <iostream>
using namespace std;

template <class T> class MyArray {
private:
    T* storage;
    int size;
public:
    MyArray(int arg = 10) {
        storage = new T[arg];
        size = arg;
    }

    ~MyArray() {
        delete[] storage;
        storage = 0;
    }

    T& operator[](const int location) throw (const char *);
};

template <class T> T& MyArray<T>::operator[](const int location)
    throw (const char *) {
    if (location < 0 || location >= size) throw "Invalid array access";
    else return storage[location];
}
```

```

}

int main() {
    try {
        MyArray<int> x(13);
        x[0] = 45;
        x[1] = 2435;
        cout << x[0] << endl;
        cout << x[1] << endl;
        x[13] = 84;
    }
    catch (const char* e) {
        cout << e << endl;
    }
}

```

The following is the output of the above example:

```

45
2435
Invalid array access

```

The expression `x[1]` is interpreted as `x.operator[](1)` and calls `int& MyArray<int>::operator[](const int)`.

Related information

- “Array subscripting operator []” on page 138

Overloading class member access (C++ only)

You overload operator-`>` with a nonstatic member function that has no parameters. The following example demonstrates how the compiler interprets overloaded class member access operators:

```

struct Y {
    void f() { };
};

struct X {
    Y* ptr;
    Y* operator->() {
        return ptr;
    };
};

int main() {
    X x;
    x->f();
}

```

The statement `x->f()` is interpreted as `(x.operator->())->f()`.

The operator-`>` is used (often in conjunction with the pointer-dereference operator) to implement “smart pointers.” These pointers are objects that behave like normal pointers except they perform other tasks when you access an object through them, such as automatic object deletion (either when the pointer is destroyed, or the pointer is used to point to another object), or reference counting (counting the number of smart pointers that point to the same object, then automatically deleting the object when that count reaches zero).

One example of a smart pointer is included in the C++ Standard Library called `auto_ptr`. You can find it in the `<memory>` header. The `auto_ptr` class implements automatic object deletion.

Related information

- “Arrow operator `->`” on page 117

Overload resolution (C++ only)

The process of selecting the most appropriate overloaded function or operator is called *overload resolution*.

Suppose that `f` is an overloaded function name. When you call the overloaded function `f()`, the compiler creates a set of *candidate functions*. This set of functions includes all of the functions named `f` that can be accessed from the point where you called `f()`. The compiler may include as a candidate function an alternative representation of one of those accessible functions named `f` to facilitate overload resolution.

After creating a set of candidate functions, the compiler creates a set of *viable functions*. This set of functions is a subset of the candidate functions. The number of parameters of each viable function agrees with the number of arguments you used to call `f()`.

The compiler chooses the *best viable function*, the function declaration that the C++ run-time environment will use when you call `f()`, from the set of viable functions. The compiler does this by *implicit conversion sequences*. An implicit conversion sequence is the sequence of conversions required to convert an argument in a function call to the type of the corresponding parameter in a function declaration. The implicit conversion sequences are ranked; some implicit conversion sequences are better than others. The best viable function is the one whose parameters all have either better or equal-ranked implicit conversion sequences than all of the other viable functions. The compiler will not allow a program in which the compiler was able to find more than one best viable function. Implicit conversion sequences are described in more detail in “Implicit conversion sequences (C++ only)” on page 237.

When a variable length array is a function parameter, the leftmost array dimension does not distinguish functions among candidate functions. In the following, the second definition of `f` is not allowed because `void f(int [])` has already been defined.

```
void f(int a[*]) {}  
void f(int a[5]) {} // illegal
```

However, array dimensions other than the leftmost in a variable length array do differentiate candidate functions when the variable length array is a function parameter. For example, the overload set for function `f` might comprise the following:

```
void f(int a[] [5]) {}  
void f(int a[] [4]) {}  
void f(int a[] [g]) {} // assume g is a global int
```

but cannot include

```
void f(int a[] [g2]) {} // illegal, assuming g2 is a global int
```

because having candidate functions with second-level array dimensions `g` and `g2` creates ambiguity about which function `f` should be called: neither `g` nor `g2` is known at compile time.

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*) { };

int main() {
    f(0xaabb);           // matches f(int);
    f((void*) 0xaabb);  // matches f(void*)
}
```

Implicit conversion sequences (C++ only)

An *implicit conversion sequence* is the sequence of conversions required to convert an argument in a function call to the type of the corresponding parameter in a function declaration.

The compiler will try to determine an implicit conversion sequence for each argument. It will then categorize each implicit conversion sequence in one of three categories and rank them depending on the category. The compiler will not allow any program in which it cannot find an implicit conversion sequence for an argument.

The following are the three categories of conversion sequences in order from best to worst:

- Standard conversion sequences
- User-defined conversion sequences
- Ellipsis conversion sequences

Note: Two standard conversion sequences or two user-defined conversion sequences may have different ranks.

Standard conversion sequences

Standard conversion sequences are categorized in one of three ranks. The ranks are listed in order from best to worst:

- Exact match: This rank includes the following conversions:
 - Identity conversions
 - Lvalue-to-rvalue conversions
 - Array-to-pointer conversions
 - Qualification conversions
- Promotion: This rank includes integral and floating point promotions.
- Conversion: This rank includes the following conversions:
 - Integral and floating-point conversions
 - Floating-integral conversions
 - Pointer conversions
 - Pointer-to-member conversions
 - Boolean conversions

The compiler ranks a standard conversion sequence by its worst-ranked standard conversion. For example, if a standard conversion sequence has a floating-point conversion, then that sequence has conversion rank.

Related information

- “Lvalue-to-rvalue conversions” on page 107
- “Pointer conversions” on page 107

- “Qualification conversions (C++ only)” on page 109
- “Integral conversions” on page 104
- “Floating-point conversions” on page 104
- “Boolean conversions” on page 104

User-defined conversion sequences

A *user-defined conversion sequence* consists of the following:

- A standard conversion sequence
- A user-defined conversion
- A second standard conversion sequence

A user-defined conversion sequence A is better than a user-defined conversion sequence B if the both have the same user-defined conversion function or constructor, and the second standard conversion sequence of A is better than the second standard conversion sequence of B.

Ellipsis conversion sequences

An *ellipsis conversion sequence* occurs when the compiler matches an argument in a function call with a corresponding ellipsis parameter.

Resolving addresses of overloaded functions (C++ only)

If you use an overloaded function name *f* without any arguments, that name can refer to a function, a pointer to a function, a pointer to member function, or a specialization of a function template. Because you did not provide any arguments, the compiler cannot perform overload resolution the same way it would for a function call or for the use of an operator. Instead, the compiler will try to choose the best viable function that matches the type of one of the following expressions, depending on where you have used *f*:

- An object or reference you are initializing
- The left side of an assignment
- A parameter of a function or a user-defined operator
- The return value of a function, operator, or conversion
- An explicit type conversion

If the compiler chose a declaration of a nonmember function or a static member function when you used *f*, the compiler matched the declaration with an expression of type pointer-to-function or reference-to-function. If the compiler chose a declaration of a nonstatic member function, the compiler matched that declaration with an expression of type pointer-to-member function. The following example demonstrates this:

```
struct X {
    int f(int) { return 0; }
    static int f(char) { return 0; }
};

int main() {
    int (X::*a)(int) = &X::f;
    // int (*b)(int) = &X::f;
}
```

The compiler will not allow the initialization of the function pointer *b*. No nonmember function or static function of type `int(int)` has been declared.

If *f* is a template function, the compiler will perform template argument deduction to determine which template function to use. If successful, it will add that function to

the list of viable functions. If there is more than one function in this set, including a non-template function, the compiler will eliminate all template functions from the set and choose the non-template function. If there are only template functions in this set, the compiler will choose the most specialized template function. The following example demonstrates this:

```
template<class T> int f(T) { return 0; }
template<> int f(int) { return 0; }
int f(int) { return 0; }

int main() {
    int (*a)(int) = f;
    a(1);
}
```

The function call `a(1)` calls `int f(int)`.

Related information

- “Pointers to functions” on page 214
- “Pointers to members (C++ only)” on page 256
- “Function templates (C++ only)” on page 330
- “Explicit specialization (C++ only)” on page 341

Chapter 11. Classes (C++ only)

A *class* is a mechanism for creating user-defined data types. It is similar to the C language structure data type. In C, a structure is composed of a set of data members. In C++, a class type is like a C structure, except that a class is composed of a set of data members and a set of operations that can be performed on the class.

In C++, a class type can be declared with the keywords `union`, `struct`, or `class`. A union object can hold any one of a set of named members. Structure and class objects hold a complete set of members. Each class type represents a unique set of class members including data members, member functions, and other type names. The default access for members depends on the class key:

- The members of a class declared with the keyword `class` are private by default. A class is inherited privately by default.
- The members of a class declared with the keyword `struct` are public by default. A structure is inherited publicly by default.
- The members of a union (declared with the keyword `union`) are public by default. A union cannot be used as a base class in 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 */
};
int main()
{
    X xobject1;      // create an object of class type X
    X xobject2;     // create another object of class type X
}
```

You may have *polymorphic* classes in C++. Polymorphism is the ability to use a function name that appears in different classes (related by inheritance), without knowing exactly the class the function belongs to at compile time.

C++ allows you to redefine standard operators and functions through the concept of overloading. Operator overloading facilitates data abstraction by allowing you to use classes as easily as built-in types.

Related information

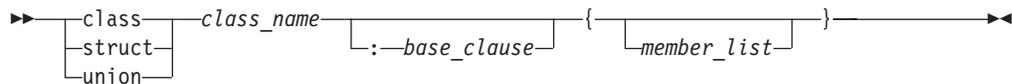
- “Structures and unions” on page 55
- Chapter 12, “Class members and friends (C++ only),” on page 251
- Chapter 13, “Inheritance (C++ only),” on page 273
- Chapter 10, “Overloading (C++ only),” on page 225
- “Virtual functions (C++ only)” on page 291

Declaring class types (C++ only)

A class declaration creates a unique type class name.

A *class specifier* is a type specifier used to declare a class. Once a class specifier has been seen and its members declared, a class is considered to be defined even if the member functions of that class are not yet defined.

Class specifier syntax



The *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.

The *member_list* 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 the size of the class is not required.

The *base_clause* specifies the base class or classes from which the class *class_name* inherits members. If the *base_clause* is not empty, the class *class_name* is called a *derived class*.

A *structure* is a class declared with the *class_key* struct. The members and base classes of a structure are public by default. A *union* is a class declared with the *class_key* union. The members of a union are public by default; a union holds only one data member at a time.

An *aggregate class* is a class that has no user-defined constructors, no private or protected non-static data members, no base classes, and no virtual functions.

Related information

- “Class member lists (C++ only)” on page 251
- “Derivation (C++ only)” on page 275

Using class objects (C++ only)

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 {  
    // members of class X  
};  
  
struct Y {  
    // members of struct Y  
};  
  
union Z {  
    // members of union Z  
};
```

You can then declare objects of each of these class types. Remember that classes, structures, and unions are all types of C++ classes.

```
int 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 { /* ... */ };
int main ()
{
    int X;        // hides the class name X
    Y yobj;       // valid
    X xobj;       // error, class name X is hidden
    class X xobj; // valid
}
```

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 of class `S` in a single declaration:

```
class S { /* ... */ };
int main()
{
    S S,T; // declare two objects of class type S
}
```

this declaration is equivalent to:

```
class S { /* ... */ };
int main()
{
    S S;
    class S T; // keyword class is required
               // since variable S hides class type S
}
```

but is not equivalent to:

```
class S { /* ... */ };
int main()
{
    S S;
    S T; // error, S class type is hidden
}
```

You can also declare references to classes, pointers to classes, and arrays of classes. For example:

```
class X { /* ... */ };
struct Y { /* ... */ };
union Z { /* ... */ };
int 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 initialize classes in external, static, and automatic definitions. The initializer contains an = (equal sign) followed by a brace-enclosed, comma-separated list of values. You do not need to initialize all members of a class.

Objects of class types that are not copy restricted can be assigned, passed as arguments to functions, and returned by functions.

Related information

- “Structures and unions” on page 55
- “References (C++ only)” on page 87
- “Scope of class names (C++ only)” on page 245

Classes and structures (C++ only)

The C++ class is an extension of the C language structure. Because the only difference between a structure and a class is that structure members have public access by default and class members have private access by default, 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`:

CCNX10C

```
class X {  
  
    // private by default  
    int a;  
  
public:  
  
    // public member function  
    int f() { return a = 5; };  
};  
  
struct Y {  
  
    // public by default  
    int f() { return a = 5; };  
  
private:  
  
    // private data member  
    int a;  
};
```

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 `obj_X` even though `obj_X` has been declared using an elaborated type specifier that uses the class key `class`:

CCNX10D

```
#include <iostream>  
using namespace std;  
  
struct X {  
    int a;  
    int b;  
};
```

```

class X obj_X;

int main() {
    obj_X.a = 0;
    obj_X.b = 1;
    cout << "Here are a and b: " << obj_X.a << " " << obj_X.b << endl;
}

```

The following is the output of the above example:

Here are a and b: 0 1

Related information

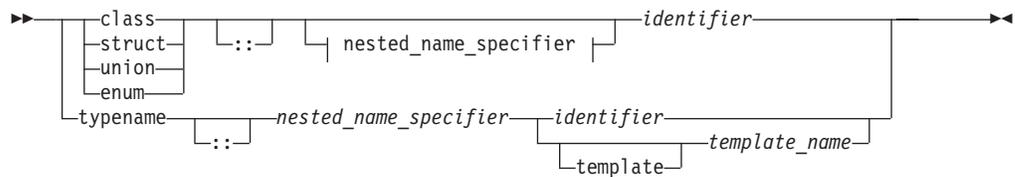
- “Structures and unions” on page 55

Scope of class names (C++ only)

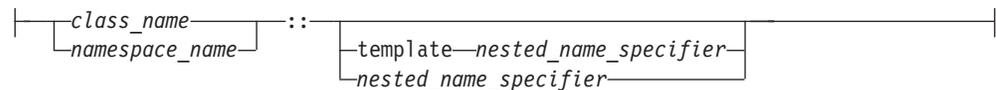
A class declaration introduces the class name into the scope where it is declared. Any class, object, function or other declaration of that name in an enclosing scope is hidden.

If a class name is declared in the same scope as a function, enumerator, or object with the same name, you must refer to that class using an *elaborated type specifier*.

Elaborated type specifier syntax



Nested name specifier:



The following example must use an elaborated type specifier to refer to class A because this class is hidden by the definition of the function A():

```

class A { };

void A (class A*) { };

int main()
{
    class A* x;
    A(x);
}

```

The declaration `class A* x` is an elaborated type specifier. Declaring a class with the same name of another function, enumerator, or object as demonstrated above is not recommended.

An elaborated type specifier can also be used in the incomplete declaration of a class type to reserve the name for a class type within the current scope.

Related information

- “Class scope (C++ only)” on page 4
- “Incomplete class declarations (C++ only)”

Incomplete class declarations (C++ only)

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 the size of the class is not required.

For example, you can define a pointer to the structure `first` in the definition of the structure `second`. Structure `first` is declared in an incomplete class declaration prior to the definition of `second`, and 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;
};
```

However, 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
};
```

Related information

- “Class member lists (C++ only)” on page 251

Nested classes (C++ only)

A *nested class* is declared 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, including 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. The following example demonstrates this:

```
class A {
    int x;

    class B { };

    class C {

        // The compiler cannot allow the following
        // declaration because A::B is private:
        // B b;

        int y;
        void f(A* p, int i) {

            // The compiler cannot allow the following
            // statement because A::x is private:
            // p->x = i;

        }
    };

    void g(C* p) {

        // The compiler cannot allow the following
        // statement because C::y is private:
        // int z = p->y;
    }
};

int main() { }
```

The compiler would not allow the declaration of object `b` because class `A::B` is private. The compiler would not allow the statement `p->x = i` because `A::x` is private. The compiler would not allow the statement `int z = p->y` because `C::y` is private.

You can define member functions and static data members of a nested class in namespace scope. For example, in the following code fragment, you can access the static members `x` and `y` and member functions `f()` and `g()` of the nested class nested by using a qualified type name. Qualified type names allow you to define a typedef to represent a qualified class name. You can then use the typedef with the `::` (scope resolution) operator to refer to a nested class or class member, as shown in the following example:

```
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; };

```

However, using a typedef to represent a nested class name hides information and may make the code harder to understand.

You cannot use a typedef name in an elaborated type specifier. To illustrate, you cannot use the following declaration in the above example:

```
class outnest obj;
```

A nested class may inherit from private members of its enclosing class. The following example demonstrates this:

```

class A {
private:
    class B { };
    B *z;

    class C : private B {
private:
        B y;
//      A::B y2;
        C *x;
//      A::C *x2;
    };
};

```

The nested class `A::C` inherits from `A::B`. The compiler does not allow the declarations `A::B y2` and `A::C *x2` because both `A::B` and `A::C` are private.

Related information

- “Class scope (C++ only)” on page 4
- “Scope of class names (C++ only)” on page 245
- “Member access (C++ only)” on page 265
- “Static members (C++ only)” on page 260

Local classes (C++ only)

A *local class* is declared within a function definition. 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
    };
}

int main()
{
    local* z;                    // error: the class local is not visible
    // ...}

```

Member functions of a local class have to be defined within their class definition, if they are defined at all. As a result, member functions of a local class are inline functions. Like all member functions, those defined within the scope of a local class do not need the keyword `inline`.

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 noninline
                                // 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.

Related information

- “Member functions (C++ only)” on page 253
- “The inline function specifier” on page 190

Local type names (C++ only)

Local type names follow the same scope rules as other names. Type names defined within a class declaration have class scope and cannot be used outside their class without qualification.

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 it is used in the class declaration.

For example:

```

int 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);
};
```

Here, function `f()` takes an argument of type `s::T`. However, the following declarations, where the order of the members of `s` has been reversed, 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.

Related information

- “Scope” on page 1
- “typedef definitions” on page 65

Chapter 12. Class members and friends (C++ only)

This section discusses the declaration of class members with respect to the information hiding mechanism and how a class can grant functions and classes access to its nonpublic members by the use of the friend mechanism. C++ expands the concept of information hiding to include the notion of having a public class interface but a private implementation. It is the mechanism for limiting direct access to the internal representation of a class type by functions in a program.

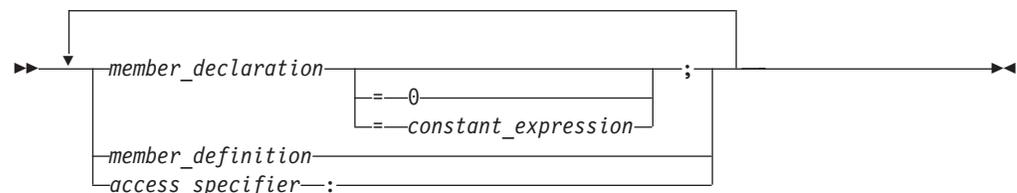
Related information

- “Member access (C++ only)” on page 265
- “Inherited member access (C++ only)” on page 278

Class member lists (C++ only)

An optional *member list* declares subobjects called *class members*. Class members can be data, functions, nested types, and enumerators.

Class member list syntax



The member list follows the class name and is placed between braces. The following applies to member lists, and members of member lists:

- A *member_declaration* or a *member_definition* may be a declaration or definition of a data member, member function, nested type, or enumeration. (The enumerators of a enumeration defined in a class member list are also members of the class.)
- A member list is the only place where you can declare class members.
- Friend declarations are not class members but must appear in member lists.
- The member list in a class definition declares all the members of a class; you cannot add members elsewhere.
- You cannot declare a member twice in a member list.
- You may declare a data member or member function as `static` but not `auto`, `extern`, or `register`.
- You may declare a nested class, a member class template, or a member function, and define it outside the class.
- You must define static data members outside the class.
- Nonstatic members that are class objects must be objects of previously defined classes; a class A cannot contain an object of class A, but it can contain a pointer or reference to an object of class A.
- You must specify all dimensions of a nonstatic array member.

A *constant initializer* (`= constant_expression`) may only appear in a class member of integral or enumeration type that has been declared `static`.

A *pure specifier* (= 0) indicates that a function has no definition. It is only used with member functions declared as `virtual` and replaces the function definition of a member function in the member list.

An *access specifier* is one of `public`, `private`, or `protected`.

A *member declaration* declares a class member for the class containing the declaration.

Suppose A is a name of a class. The following class members of A must have a name different from A:

- All data members
- All type members
- All enumerators of enumerated type members
- All members of all anonymous union members

Related information

- “Declaring class types (C++ only)” on page 242
- “Member access (C++ only)” on page 265
- “Inherited member access (C++ only)” on page 278
- “Static members (C++ only)” on page 260

Data members (C++ only)

Data members include members that are declared with any of the fundamental types, as well as other types, including pointer, reference, array types, bit fields, and user-defined types. You can declare a data member the same way as a variable, except that explicit initializers are not allowed inside the class definition. However, a `const static` data member of integral or enumeration type may have an explicit initializer.

If an array is declared as a nonstatic class member, you must specify all of the dimensions of the array.

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 has been previously declared. An incomplete class type can be used in a member declaration as long as the size of the class is not needed. For example, a member can be declared that is a pointer to an incomplete class type.

A class X cannot have a member that is of type X, but it can 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);
};
```

Related information

- “Member access (C++ only)” on page 265

- “Inherited member access (C++ only)” on page 278
- “Static members (C++ only)” on page 260

Member functions (C++ only)

Member functions are operators and functions that are declared as members of a class. Member functions do not include operators and functions declared with the friend specifier. These are called *friends* of a class. You can declare a member function as `static`; this is called a *static member function*. A member function that is not declared as `static` is called a *nonstatic member function*.

The definition of a member function is within the scope of its enclosing class. The body of a member function is analyzed after the class declaration so that members of that class can be used in the member function body, even if the member function definition appears before the declaration of that member in the class member list. 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;
};
```

Inline member functions (C++ only)

You may either define a member function inside its class definition, or you may define it outside if you have already declared (but not defined) the member function in the class definition.

A member function that is defined inside its class member list is called an *inline member function*. Member functions containing a few lines of code are usually declared inline. In the above example, `add()` is an inline member function. If you define a member function outside of its class definition, it must appear in a namespace scope enclosing the class definition. You must also qualify the member function name using the scope resolution (`::`) operator.

An equivalent way to declare an inline member function is to either declare it in the class with the `inline` keyword (and define the function outside of its class) or to define it outside of the class declaration using the `inline` keyword.

In the following example, member function `Y::f()` is an inline member function:

```
struct Y {
private:
    char* a;
public:
    char* f() { return a; }
};
```

The following example is equivalent to the previous example; `Y::f()` is an inline member function:

```
struct Y {
private:
    char* a;
public:
```

```

    char* f();
};

inline char* Y::f() { return a; }

```

The `inline` specifier does not affect the linkage of a member or nonmember function: linkage is external by default.

Member functions of a local class must be defined within their class definition. As a result, member functions of a local class are implicitly inline functions. These inline member functions have no linkage.

Related information

- “Friends (C++ only)” on page 267
- “Static member functions (C++ only)” on page 263
- “The inline function specifier” on page 190
- “Local classes (C++ only)” on page 248

Constant and volatile member functions (C++ only)

A member function declared with the `const` qualifier can be called for constant and nonconstant objects. A nonconstant member function can only be called for a nonconstant object. Similarly, a member function declared with the `volatile` qualifier can be called for volatile and nonvolatile objects. A nonvolatile member function can only be called for a nonvolatile object.

Related information

- “Type qualifiers” on page 67
- “The `this` pointer (C++ only)” on page 257

Virtual member functions (C++ only)

Virtual member functions are declared with the keyword `virtual`. They allow dynamic binding of member functions. Because all virtual functions must be member functions, virtual member functions are simply called *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. A class that has at least one pure virtual function is called an *abstract class*.

Related information

- “Virtual functions (C++ only)” on page 291
- “Abstract classes (C++ only)” on page 296

Special member functions (C++ only)

Special member functions are used to create, destroy, initialize, convert, and copy class objects. These include the following:

- Constructors
- Destructors
- Conversion constructors
- Conversion functions
- Copy constructors

For full descriptions of these functions, see Chapter 14, “Special member functions (C++ only),” on page 299.

Member scope (C++ only)

Member functions and static members can be defined outside their class declaration if they have already been declared, but not defined, in the class member list. Nonstatic data members are defined when an object of their class is created. The declaration of a static data member is not a definition. The declaration of a member function is a definition if the body of the function is also given.

Whenever the definition of a class member appears outside of the class declaration, the member name must be qualified by the class name using the `::` (scope resolution) operator.

The following example defines a member function outside of its class declaration.

CCNX11A

```
#include <iostream>
using namespace std;

struct X {
    int a, b ;

    // member function declaration only
    int add();
};

// global variable
int a = 10;

// define member function outside its class declaration
int X::add() { return a + b; }

int main() {
    int answer;
    X xobject;
    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 they are defined 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. Unless you use one of the class access operators, `.` (dot), or `->` (arrow), or `::` (scope resolution) operator, 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()
{
    char j;
    return 0;
}
```

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 that when the containing classes are being searched, only the definitions of the containing classes and their base classes are searched. The scope containing the base class definitions (global scope, in this example) is not searched.

Related information

- “Class scope (C++ only)” on page 4

Pointers to members (C++ only)

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 is not associated 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.

Pointers to members can be declared and used as shown in the following example:

```
#include <iostream>
using namespace std;

class X {
public:
    int a;
    void f(int b) {
        cout << "The value of b is "<< b << endl;
    }
};

int main() {

    // declare pointer to data member
    int X::*ptiptr = &X::a;
```

```

// declare a pointer to member function
void (X::* ptfptr) (int) = &X::f;

// create an object of class type X
X xobject;

// initialize data member
xobject.*ptiptr = 10;

cout << "The value of a is " << xobject.*ptiptr << endl;

// call member function
(xobject.*ptfptr) (20);
}

```

The output for this example is:

```

The value of a is 10
The value of b is 20

```

To reduce complex syntax, you can declare a typedef to be a pointer to a member. A pointer to a member can be declared and used as shown in the following code fragment:

```

typedef int X::*my_pointer_to_member;
typedef void (X::*my_pointer_to_function) (int);

int main() {
    my_pointer_to_member ptiptr = &X::a;
    my_pointer_to_function ptfptr = &X::f;
    X xobject;
    xobject.*ptiptr = 10;
    cout << "The value of a is " << xobject.*ptiptr << endl;
    (xobject.*ptfptr) (20);
}

```

The pointer to member operators `.*` and `->*` are used 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`.

Pointer-to-member conversion can occur when pointers to members are initialized, assigned, or compared. Note that pointer to a member is not the same as a pointer to an object or a pointer to a function.

Related information

- “Pointer to member operators `.*` `->*` (C++ only)” on page 141

The this pointer (C++ only)

The keyword `this` identifies a special type of pointer. Suppose that you create an object named `x` of class `A`, and class `A` has a nonstatic member function `f()`. If you call the function `x.f()`, the keyword `this` in the body of `f()` stores the address of `x`. You cannot declare the `this` pointer or make assignments to it.

A static member function does not have a `this` pointer.

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 **const** qualifier, the type of the `this` pointer for that member function for class `X`, is `const X* const`.

A `const` this pointer can be used only with `const` member functions. Data members of the class will be constant within that function. The function is still able to change the value, but requires a `const_cast` to do so:

```
void foo::p() const{
member = 1;                // illegal
const_cast <int&> (member) = 1; // a bad practice but legal
}
```

A better technique would be to declare member mutable.

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`. For example, the compiler will not allow the following:

```
struct A {
    int a;
    int f() const { return a++; }
};
```

The compiler will not allow the statement `a++` in the body of function `f()`. In the function `f()`, the `this` pointer is of type `A* const`. The function `f()` is trying to modify part of the object to which `this` points.

The `this` pointer is passed as a hidden argument to all nonstatic member function calls and 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 for by using the `this` pointer in the body of the member function. The following code example produces the output `a = 5`:

CCNX11C

```
#include <iostream>
using namespace std;

struct X {
private:
    int a;
public:
    void Set_a(int a) {

        // The 'this' pointer is used to retrieve 'xobj.a'
        // hidden by the automatic variable 'a'
        this->a = a;
    }
    void Print_a() { cout << "a = " << a << endl; }
};

int main() {
    X xobj;
    int a = 5;
    xobj.Set_a(a);
    xobj.Print_a();
}
```

In the member function `Set_a()`, the statement `this->a = a` uses the `this` pointer to retrieve `xobj.a` hidden by the automatic variable `a`.

Unless a class member name is hidden, using the class member name is equivalent to using the class member name with the `this` pointer and the class member access operator (`->`).

The example in the first column of the following table shows code that uses class members without the this pointer. The code in the second column uses the variable THIS to simulate the first column's hidden use of the this pointer:

Code without using this pointer	Equivalent code, the THIS variable simulating the hidden use of the this pointer
<pre>#include <string> #include <iostream> using namespace std; struct X { private: int len; char *ptr; public: int GetLen() { return len; } char * GetPtr() { return ptr; } X& Set(char *); X& Cat(char *); X& Copy(X&); void Print(); }; X& X::Set(char *pc) { len = strlen(pc); ptr = new char[len]; strcpy(ptr, pc); return *this; } X& X::Cat(char *pc) { len += strlen(pc); strcat(ptr,pc); return *this; } X& X::Copy(X& x) { Set(x.GetPtr()); return *this; } void X::Print() { cout << ptr << endl; } int main() { X xobj1; xobj1.Set("abcd") .Cat("efgh"); xobj1.Print(); X xobj2; xobj2.Copy(xobj1) .Cat("ijkl"); xobj2.Print(); }</pre>	<pre>#include <string> #include <iostream> using namespace std; struct X { private: int len; char *ptr; public: int GetLen (X* const THIS) { return THIS->len; } char * GetPtr (X* const THIS) { return THIS->ptr; } X& Set(X* const, char *); X& Cat(X* const, char *); X& Copy(X* const, X&); void Print(X* const); }; X& X::Set(X* const THIS, char *pc) { THIS->len = strlen(pc); THIS->ptr = new char[THIS->len]; strcpy(THIS->ptr, pc); return *THIS; } X& X::Cat(X* const THIS, char *pc) { THIS->len += strlen(pc); strcat(THIS->ptr, pc); return *THIS; } X& X::Copy(X* const THIS, X& x) { THIS->Set(THIS, x.GetPtr(&x)); return *THIS; } void X::Print(X* const THIS) { cout << THIS->ptr << endl; } int main() { X xobj1; xobj1.Set(&xobj1 , "abcd") .Cat(&xobj1 , "efgh"); xobj1.Print(&xobj1); X xobj2; xobj2.Copy(&xobj2 , xobj1) .Cat(&xobj2 , "ijkl"); xobj2.Print(&xobj2); }</pre>

Both examples produce the following output:

```
abcdefgh
abcdefghijkl
```

Related information

- “Overloading assignments (C++ only)” on page 232
- “Copy constructors (C++ only)” on page 315

Static members (C++ only)

Class members can be declared using the storage class specifier `static` in the class member list. Only one copy of the static member is shared by all objects of a class in a program. When you declare an object of a class having 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 keep track of the total number of objects.

You access a static member by qualifying the class name using the `::` (scope resolution) operator. In the following example, you can refer to the static member `f()` of class type `X` as `X::f()` even if no object of type `X` is ever declared:

```
struct X {
    static int f();
};

int main() {
    X::f();
}
```

Related information

- “Constant and volatile member functions (C++ only)” on page 254
- “The static storage class specifier” on page 44
- “Class member lists (C++ only)” on page 251

Using the class access operators with static members (C++ only)

You do not have to use the class member access syntax to refer to a static member; to access a static member `s` of class `X`, you could use the expression `X::s`. The following example demonstrates accessing a static member:

```
#include <iostream>
using namespace std;

struct A {
    static void f() { cout << "In static function A::f()" << endl; }
};

int main() {

    // no object required for static member
    A::f();

    A a;
    A* ap = &a;
    a.f();
    ap->f();
}
```

The three statements `A::f()`, `a.f()`, and `ap->f()` all call the same static member function `A::f()`.

You can directly refer to a static member in the same scope of its class, or in the scope of a class derived from the static member's class. The following example demonstrates the latter case (directly referring to a static member in the scope of a class derived from the static member's class):

```
#include <iostream>
using namespace std;

int g() {
    cout << "In function g()" << endl;
    return 0;
}

class X {
public:
    static int g() {
        cout << "In static member function X::g()" << endl;
        return 1;
    }
};

class Y: public X {
public:
    static int i;
};

int Y::i = g();

int main() { }
```

The following is the output of the above code:

```
In static member function X::g()
```

The initialization `int Y::i = g()` calls `X::g()`, not the function `g()` declared in the global namespace.

Related information

- “The static storage class specifier” on page 44
- “Scope resolution operator `::` (C++ only)” on page 116
- “Dot operator `.`” on page 117
- “Arrow operator `->`” on page 117

Static data members (C++ only)

The declaration of a static data member in the member list of a class is not a definition. You must define the static member outside of the class declaration, in namespace scope. For example:

```
class X
{
public:
    static int i;
};
int X::i = 0; // definition outside class declaration
```

Once you define a static data member, it exists even though no objects of the static data member's class exist. In the above example, no objects of class `X` exist even though the static data member `X::i` has been defined.

Static data members of a class in namespace scope have external linkage. The initializer for a static data member is in the scope of the class declaring the member.

A static data member can be of any type except for `void` or `void` qualified with `const` or `volatile`. You cannot declare a static data member as `mutable`.

You can only have one definition of a static member in a program. Unnamed classes, classes contained within unnamed classes, and local classes cannot have static data members.

Static data members and their initializers can access other static private and protected members of their class. 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::q` cause errors because `y` is an object of a class that is derived privately from `C`, and its members are not accessible to members of `C`.

If a static data member is of `const` integral or `const` enumeration type, you may specify a *constant initializer* in the static data member's declaration. This constant initializer must be an integral constant expression. Note that the constant initializer is not a definition. You still need to define the static member in an enclosing namespace. The following example demonstrates this:

```
#include <iostream>
using namespace std;

struct X {
    static const int a = 76;
};

const int X::a;
```

```
int main() {
    cout << X::a << endl;
}
```

The tokens = 76 at the end of the declaration of static data member a is a constant initializer.

Related information

- “External linkage” on page 8
- “Member access (C++ only)” on page 265
- “Local classes (C++ only)” on page 248

Static member functions (C++ only)

You cannot have static and nonstatic member functions with the same names and the same number and type of arguments.

Like static data members, you may access a static member function `f()` of a class A without using an object of class A.

A static member function does not have a `this` pointer. The following example demonstrates this:

```
#include <iostream>
using namespace std;

struct X {
private:
    int i;
    static int si;
public:
    void set_i(int arg) { i = arg; }
    static void set_si(int arg) { si = arg; }

    void print_i() {
        cout << "Value of i = " << i << endl;
        cout << "Again, value of i = " << this->i << endl;
    }

    static void print_si() {
        cout << "Value of si = " << si << endl;
        // cout << "Again, value of si = " << this->si << endl;
    }
};

int X::si = 77;      // Initialize static data member

int main() {
    X xobj;
    xobj.set_i(11);
    xobj.print_i();

    // static data members and functions belong to the class and
    // can be accessed without using an object of class X
    X::print_si();
    X::set_si(22);
    X::print_si();
}
```

The following is the output of the above example:

```
Value of i = 11
Again, value of i = 11
Value of si = 77
Value of si = 22
```

The compiler does not allow the member access operation `this->si` in function `A::print_si()` because this member function has been declared as static, and therefore 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:

CCNX11H

```
#include <iostream>
using namespace std;

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;

int main() {
    C obj_C(0);
    obj_C.printall();
}
```

The following is the output of the above example:

```
Here is j: 0
Here is i: 3
```

A static member function cannot be declared with the keywords `virtual`, `const`, `volatile`, or `const volatile`.

A static member function can access only the names of static members, enumerators, and nested types of the class in which it is declared. Suppose a static member function `f()` is a member of class `X`. The static member function `f()` cannot access the nonstatic members `X` or the nonstatic members of a base class of `X`.

Related information

- “The `this` pointer (C++ only)” on page 257

Member access (C++ only)

Member access determines if a class member is accessible in an expression or declaration. Suppose *x* is a member of class *A*. Class member *x* can be declared to have one of the following levels of accessibility:

- *public*: *x* can be used anywhere without the access restrictions defined by *private* or *protected*.
- *private*: *x* can be used only by the members and friends of class *A*.
- *protected*: *x* can be used only by the members and friends of class *A*, and the members and friends of classes derived from class *A*.

Members of classes declared with the keyword `class` are *private* by default. Members of classes declared with the keyword `struct` or `union` are *public* by default.

To control the access of a class member, you use one of the *access specifiers* `public`, `private`, or `protected` as a label in a class member list. The following example demonstrates these access specifiers:

```
struct A {
    friend class C;
private:
    int a;
public:
    int b;
protected:
    int c;
};
```

```
struct B : A {
    void f() {
        // a = 1;
        b = 2;
        c = 3;
    }
};
```

```
struct C {
    void f(A x) {
        x.a = 4;
        x.b = 5;
        x.c = 6;
    }
};
```

```
int main() {
    A y;
    // y.a = 7;
    y.b = 8;
    // y.c = 9;

    B z;
    // z.a = 10;
    z.b = 11;
    // z.c = 12;
}
```

The following table lists the access of data members `A::a`, `A::b`, and `A::c` in various scopes of the above example.

Scope	A::a	A::b	A::c
function B::f()	No access. Member A::a is private.	Access. Member A::b is public.	Access. Class B inherits from A.
function C::f()	Access. Class C is a friend of A.	Access. Member A::b is public.	Access. Class C is a friend of A.
object y in main()	No access. Member y.a is private.	Access. Member y.a is public.	No access. Member y.c is protected.
object z in main()	No access. Member z.a is private.	Access. Member z.a is public.	No access. Member z.c is protected.

An access specifier specifies the accessibility of members that follow it until the next access specifier or until the end of the class definition. You can use any number of access specifiers in any order. If you later define a class member within its class definition, its access specification must be the same as its declaration. The following example demonstrates this:

```
class A {
    class B;
    public:
        class B { };
};
```

The compiler will not allow the definition of class B because this class has already been declared as private.

A class member has the same access control regardless whether it has been defined within its class or outside its class.

Access control applies to names. In particular, if you add access control to a typedef name, it affects only the typedef name. The following example demonstrates this:

```
class A {
    class B { };
    public:
        typedef B C;
};

int main() {
    A::C x;
    // A::B y;
}
```

The compiler will allow the declaration A::C x because the typedef name A::C is public. The compiler would not allow the declaration A::B y because A::B is private.

Note that accessibility and visibility are independent. Visibility is based on the scoping rules of C++. A class member can be visible and inaccessible at the same time.

Related information

- “Scope” on page 1
- “Class member lists (C++ only)” on page 251
- “Inherited member access (C++ only)” on page 278

Friends (C++ only)

A friend of a class *X* is a function or class that is not a member of *X*, but is granted the same access to *X* as the members of *X*. Functions declared with the friend specifier in a class member list are called *friend functions* of that class. Classes declared with the friend specifier in the member list of another class are called *friend classes* of that class.

A class *Y* must be defined before any member of *Y* can be declared a friend of another class.

In the following example, the friend function `print` is a member of class *Y* and accesses the private data members `a` and `b` of class *X*.

CCNX11I

```
#include <iostream>
using namespace std;

class X;

class Y {
public:
    void print(X& x);
};

class X {
    int a, b;
    friend void Y::print(X& x);
public:
    X() : a(1), b(2) { }
};

void Y::print(X& x) {
    cout << "a is " << x.a << endl;
    cout << "b is " << x.b << endl;
}

int main() {
    X xobj;
    Y yobj;
    yobj.print(xobj);
}
```

The following is the output of the above example:

```
a is 1
b is 2
```

You can declare an entire class as a friend. Suppose class *F* is a friend of class *A*. This means that every member function and static data member definition of class *F* has access to class *A*.

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* and performs the same task as the friend function `print` in the above example. Any other members declared in class *F* also have access to all members of class *X*.

CCNX11J

```
#include <iostream>
using namespace std;
```

```

class X {
    int a, b;
    friend class F;
public:
    X() : a(1), b(2) { }
};

class F {
public:
    void print(X& x) {
        cout << "a is " << x.a << endl;
        cout << "b is " << x.b << endl;
    }
};

int main() {
    X xobj;
    F fobj;
    fobj.print(xobj);
}

```

The following is the output of the above example:

```

a is 1
b is 2

```

You must use an elaborated type specifier when you declare a class as a friend. The following example demonstrates this:

```

class F;
class G;
class X {
    friend class F;
    friend G;
};

```

You cannot define a class in a friend declaration. For example, the compiler will not allow the following:

```

class F;
class X {
    friend class F { };
};

```

However, you can define a function in a friend declaration. The class must be a non-local class, function, the function name must be unqualified, and the function has namespace scope. The following example demonstrates this:

```

class A {
    void g();
};

void z() {
    class B {
    // friend void f() { };
    };
}

class C {
// friend void A::g() { }
    friend void h() { }
};

```

The compiler would not allow the function definition of `f()` or `g()`. The compiler will allow the definition of `h()`.

You cannot declare a friend with a storage class specifier.

Related information

- “Member access (C++ only)” on page 265
- “Inherited member access (C++ only)” on page 278

Friend scope (C++ only)

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 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, unqualified names in the body of the function definition are searched for starting from the class containing the function definition.

If the name of a friend class has been introduced before the friend declaration, the compiler searches for a class name that matches the name of the friend class beginning at the scope of the friend declaration. If 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. For example:

```
class A {
    class B { // arbitrary nested class definitions
        friend class C;
    };
};
```

is equivalent to:

```
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 scope resolution operator (::). For example:

```
class A {
public:
    int f() { }
};

class B {
    friend int A::f();
};
```

Friends of a base class are not inherited by any classes derived from that base class. The following example demonstrates this:

```
class A {
    friend class B;
    int a;
};
```

```

class B { };

class C : public B {
    void f(A* p) {
        // p->a = 2;
    }
};

```

The compiler would not allow the statement `p->a = 2` because class `C` is not a friend of class `A`, although `C` inherits from a friend of `A`.

Friendship is not transitive. The following example demonstrates this:

```

class A {
    friend class B;
    int a;
};

class B {
    friend class C;
};

class C {
    void f(A* p) {
        // p->a = 2;
    }
};

```

The compiler would not allow the statement `p->a = 2` because class `C` is not a friend of class `A`, although `C` is a friend of a friend of `A`.

If you declare a friend in a local class, and the friend's name is unqualified, the compiler will look for the name only within the innermost enclosing nonclass scope. You must declare a function before declaring it as a friend of a local scope. You do not have to do so with classes. However, a declaration of a friend class will hide a class in an enclosing scope with the same name. The following example demonstrates this:

```

class X { };
void a();

void f() {
    class Y { };
    void b();
    class A {
        friend class X;
        friend class Y;
        friend class Z;
        // friend void a();
        friend void b();
        // friend void c();
    };
    ::X moocow;
    // X moocow2;
}

```

In the above example, the compiler will allow the following statements:

- `friend class X`: This statement does not declare `::X` as a friend of `A`, but the local class `X` as a friend, even though this class is not otherwise declared.
- `friend class Y`: Local class `Y` has been declared in the scope of `f()`.
- `friend class Z`: This statement declares the local class `Z` as a friend of `A` even though `Z` is not otherwise declared.
- `friend void b()`: Function `b()` has been declared in the scope of `f()`.

- `::X moocow`: This declaration creates an object of the nonlocal class `::X`.

The compiler would not allow the following statements:

- `friend void a()`: This statement does not consider function `a()` declared in namespace scope. Since function `a()` has not been declared in the scope of `f()`, the compiler would not allow this statement.
- `friend void c()`: Since function `c()` has not been declared in the scope of `f()`, the compiler would not allow this statement.
- `X moocow2`: This declaration tries to create an object of the local class `X`, not the nonlocal class `::X`. Since local class `X` has not been defined, the compiler would not allow this statement.

Related information

- “Scope of class names (C++ only)” on page 245
- “Nested classes (C++ only)” on page 246
- “Local classes (C++ only)” on page 248

Friend access (C++ only)

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, and you can only access the protected members of a class through member functions of a class or classes derived from that class.

Friend declarations are not affected by access specifiers.

Related information

- “Member access (C++ only)” on page 265

Chapter 13. Inheritance (C++ only)

Inheritance is a mechanism of reusing and extending existing classes without modifying them, thus producing hierarchical relationships between them.

Inheritance is almost like embedding an object into a class. Suppose that you declare an object *x* of class *A* in the class definition of *B*. As a result, class *B* will have access to all the public data members and member functions of class *A*. However, in class *B*, you have to access the data members and member functions of class *A* through object *x*. The following example demonstrates this:

```
#include <iostream>
using namespace std;

class A {
    int data;
public:
    void f(int arg) { data = arg; }
    int g() { return data; }
};

class B {
public:
    A x;
};

int main() {
    B obj;
    obj.x.f(20);
    cout << obj.x.g() << endl;
    // cout << obj.g() << endl;
}
```

In the main function, object *obj* accesses function *A::f()* through its data member *B::x* with the statement *obj.x.f(20)*. Object *obj* accesses *A::g()* in a similar manner with the statement *obj.x.g()*. The compiler would not allow the statement *obj.g()* because *g()* is a member function of class *A*, not class *B*.

The inheritance mechanism lets you use a statement like *obj.g()* in the above example. In order for that statement to be legal, *g()* must be a member function of class *B*.

Inheritance lets you include the names and definitions of another class's members as part of a new class. The class whose members you want to include in your new class is called a *base class*. Your new class is *derived* from the base class. The new class contains a *subobject* of the type of the base class. The following example is the same as the previous example except it uses the inheritance mechanism to give class *B* access to the members of class *A*:

```
#include <iostream>
using namespace std;

class A {
    int data;
public:
    void f(int arg) { data = arg; }
    int g() { return data; }
};

class B : public A { };

int main() {
```

```

    B obj;
    obj.f(20);
    cout << obj.g() << endl;
}

```

Class A is a base class of class B. The names and definitions of the members of class A are included in the definition of class B; class B inherits the members of class A. Class B is derived from class A. Class B contains a subobject of type A.

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.

You may derive classes from other derived classes, thereby creating another level of inheritance. The following example demonstrates this:

```

struct A { };
struct B : A { };
struct C : B { };

```

Class B is a derived class of A, but is also a base class of C. The number of levels of inheritance is only limited by resources.

Multiple inheritance allows you to create a derived class that inherits properties from more than one base class. 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. See “Multiple inheritance (C++ only)” on page 284 for more detailed information.

A *direct base class* is a base class that appears directly as a base specifier in the declaration of its derived class.

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. For a given class, all base classes that are not direct base classes are indirect base classes. The following example demonstrates direct and indirect base classes:

```

class A {
public:
    int x;
};
class B : public A {
public:
    int y;
};
class C : public B { };

```

Class B is a direct base class of C. Class A is a direct base class of B. Class A is an indirect base class of C. (Class C has x and y as its data members.)

Polymorphic functions are functions that can be applied to objects of more than one type. In C++, polymorphic functions are implemented in two ways:

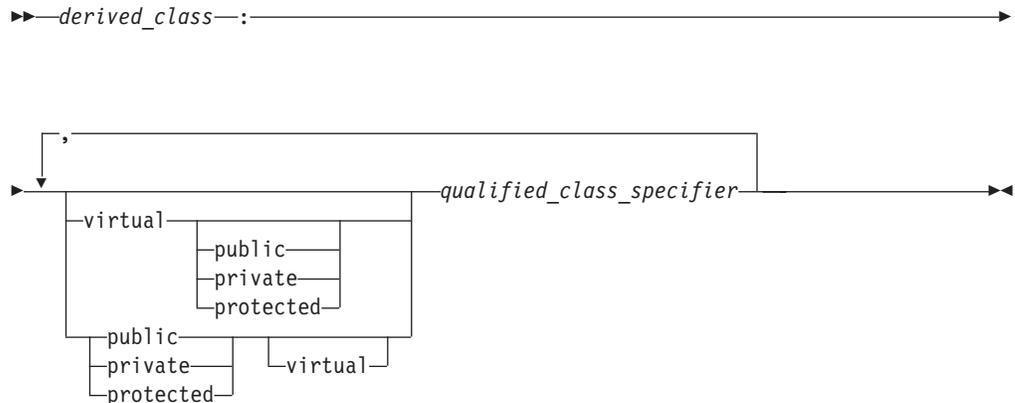
- Overloaded functions are statically bound at compile time.
- C++ provides virtual functions. A *virtual function* is a function that can be called for a number of different user-defined types that are related through derivation.

Virtual functions are bound dynamically at run time. They are described in more detail in “Virtual functions (C++ only)” on page 291.

Derivation (C++ only)

Inheritance is implemented in C++ through the mechanism of derivation. Derivation allows you to derive a class, called a *derived class*, from another class, called a *base class*.

Derived class syntax



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.

The *qualified_class_specifier* must be a class that has been previously declared in a class declaration.

An *access specifier* is one of `public`, `private`, or `protected`.

The `virtual` keyword can be used to declare virtual base classes.

The following example shows the declaration of the derived class `D` and the base classes `V`, `B1`, and `B2`. The class `B1` is both a base class and a derived class because it is derived from class `V` and is a base class for `D`:

```
class V { /* ... */ };
class B1 : virtual public V { /* ... */ };
class B2 { /* ... */ };
class D : public B1, private B2 { /* ... */ };
```

Classes that are declared but not defined are not allowed in base lists.

For example:

```
class X;

// error
class Y: public X { };
```

The compiler will not allow the declaration of class `Y` because `X` has not been defined.

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. For example:

CCNX14A

```
class Base {
public:
    int a,b;
};

class Derived : public Base {
public:
    int c;
};

int 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. For example:

CCNX14B

```
#include <iostream>
using namespace std;

class Base {
public:
    char* name;
    void display() {
        cout << name << endl;
    }
};

class Derived: public Base {
public:
    char* name;
    void display() {
        cout << name << ", " << Base::name << endl;
    }
};

int main() {
    Derived d;
    d.name = "Derived Class";
    d.Base::name = "Base Class";

    // call Derived::display()
    d.display();

    // call Base::display()
    d.Base::display();
}
```

The following is the output of the above example:

```
Derived Class, Base Class
Base Class
```

You can manipulate a derived class object as if it were 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 expecting a pointer or reference to the base class of D. You do not need to use an explicit cast to achieve this; a standard conversion is performed. You can implicitly convert a pointer to a derived class to point to an accessible unambiguous base class. You can also implicitly convert a reference to a derived class to a reference to a base class.

The following example demonstrates a standard conversion from a pointer to a derived class to a pointer to a base class:

CCNX14C

```
#include <iostream>
using namespace std;

class Base {
public:
    char* name;
    void display() {
        cout << name << endl;
    }
};

class Derived: public Base {
public:
    char* name;
    void display() {
        cout << name << ", " << Base::name << endl;
    }
};

int main() {
    Derived d;
    d.name = "Derived Class";
    d.Base::name = "Base Class";

    Derived* dptr = &d;

    // standard conversion from Derived* to Base*
    Base* bptr = dptr;

    // call Base::display()
    bptr->display();
}
```

The following is the output of the above example:

```
Base Class
```

The statement `Base* bptr = dptr` converts a pointer of type `Derived` to a pointer of type `Base`.

The reverse case is not allowed. You cannot implicitly convert a pointer or a reference to a base class object to a pointer or reference to a derived class. For example, the compiler will not allow the following code if the classes `Base` and `Class` are defined as in the above example:

```
int main() {
    Base b;
    b.name = "Base class";

    Derived* dptr = &b;
}
```

The compiler will not allow the statement `Derived* dptr = &b` because the statement is trying to implicitly convert a pointer of type `Base` to a pointer of type `Derived`.

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 base class name is hidden in the derived class.

Related information

- “Virtual base classes (C++ only)” on page 285
- “Inherited member access (C++ only)”
- “Incomplete class declarations (C++ only)” on page 246
- “Scope resolution operator `::` (C++ only)” on page 116

Inherited member access (C++ only)

The following sections discuss the access rules affecting a protected nonstatic base class member and how to declare a derived class using an access specifier:

- “Protected members (C++ only)”
- “Access control of base class members (C++ only)” on page 279

Related information

- “Member access (C++ only)” on page 265

Protected members (C++ only)

A protected nonstatic base class member can be accessed by members and friends of any classes derived from that base class by using one of the following:

- 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 a class is derived privately from a base class, all protected base class members become private members of the derived class.

If you reference a protected nonstatic member `x` of a base class `A` in a friend or a member function of a derived class `B`, you must access `x` through a pointer to, reference to, or object of a class derived from `A`. However, if you are accessing `x` to create a pointer to member, you must qualify `x` with a nested name specifier that names the derived class `B`. The following example demonstrates this:

```
class A {
public:
protected:
    int i;
};

class B : public A {
    friend void f(A*, B*);
    void g(A*);
};

void f(A* pa, B* pb) {
    // pa->i = 1;
    pb->i = 2;
}
```

```

// int A::* point_i = &A::i;
int A::* point_i2 = &B::i;
}

void B::g(A* pa) {
// pa->i = 1;
i = 2;

// int A::* point_i = &A::i;
int A::* point_i2 = &B::i;
}

void h(A* pa, B* pb) {
// pa->i = 1;
// pb->i = 2;
}

int main() { }

```

Class A contains one protected data member, an integer `i`. Because B derives from A, the members of B have access to the protected member of A. Function `f()` is a friend of class B:

- The compiler would not allow `pa->i = 1` because `pa` is not a pointer to the derived class B.
- The compiler would not allow `int A::* point_i = &A::i` because `i` has not been qualified with the name of the derived class B.

Function `g()` is a member function of class B. The previous list of remarks about which statements the compiler would and would not allow apply for `g()` except for the following:

- The compiler allows `i = 2` because it is equivalent to `this->i = 2`.

Function `h()` cannot access any of the protected members of A because `h()` is neither a friend or a member of a derived class of A.

Access control of base class members (C++ only)

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, but allows the derived class to restrict the access control of the members of a base class.

You can derive classes using any of the 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 `protected` base class, `public` and `protected` members of the base class are `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 all cases, `private` members of the base class remain `private`. `Private` members of the base class cannot be used by the derived class unless friend declarations within the base class explicitly grant access to them.

In the following example, class `d` is derived publicly from class `b`. Class `b` is declared a `public` base class by this declaration.

```

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 the derived class is declared with the keyword `class`, the default access specifier in its base list specifiers is `private`.
- If the derived class is declared with the keyword `struct`, the default access specifier in its base list specifiers is `public`.

In the following example, private derivation is used by default because no access specifier is used in the base list and the derived class is declared with the keyword `class`:

```
struct B
{ };
class D : B // 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)

Related information

- “Member access (C++ only)” on page 265
- “Member scope (C++ only)” on page 255

The using declaration and class members (C++ only)

A `using` declaration in a definition of a class `A` allows you to introduce a *name* of a data member or member function from a base class of `A` into the scope of `A`.

You would need a `using` declaration in a class definition if you want to create a set of overload a member functions from base and derived classes, or you want to change the access of a class member.

using declaration syntax

```
▶▶ using [typename] [::] nested_name_specifier unqualified_id; ▶▶
      [::] unqualified_id;
```

A `using` declaration in a class `A` may name one of the following:

- A member of a base class of `A`
- A member of an anonymous union that is a member of a base class of `A`
- An enumerator for an enumeration type that is a member of a base class of `A`

The following example demonstrates this:

```
struct Z {
    int g();
};

struct A {
    void f();
    enum E { e };
    union { int u; };
};

struct B : A {
```

```

    using A::f;
    using A::e;
    using A::u;
    // using Z::g;
};

```

The compiler would not allow the using declaration `using Z::g` because `Z` is not a base class of `A`.

A using declaration cannot name a template. For example, the compiler will not allow the following:

```

struct A {
    template<class T> void f(T);
};

struct B : A {
    using A::f<int>;
};

```

Every instance of the name mentioned in a using declaration must be accessible. The following example demonstrates this:

```

struct A {
private:
    void f(int);
public:
    int f();
protected:
    void g();
};

struct B : A {
    // using A::f;
    using A::g;
};

```

The compiler would not allow the using declaration `using A::f` because `void A::f(int)` is not accessible from `B` even though `int A::f()` is accessible.

Related information

- “Scope of class names (C++ only)” on page 245
- “The using declaration and namespaces (C++ only)” on page 222

Overloading member functions from base and derived classes (C++ only)

A member function named `f` in a class `A` will hide all other members named `f` in the base classes of `A`, regardless of return types or arguments. The following example demonstrates this:

```

struct A {
    void f() { }
};

struct B : A {
    void f(int) { }
};

int main() {
    B obj_B;
    obj_B.f(3);
    // obj_B.f();
}

```

The compiler would not allow the function call `obj_B.f()` because the declaration of `void B::f(int)` has hidden `A::f()`.

To overload, rather than hide, a function of a base class A in a derived class B, you introduce the name of the function into the scope of B with a using declaration. The following example is the same as the previous example except for the using declaration `using A::f`:

```
struct A {
    void f() { }
};

struct B : A {
    using A::f;
    void f(int) { }
};

int main() {
    B obj_B;
    obj_B.f(3);
    obj_B.f();
}
```

Because of the using declaration in class B, the name `f` is overloaded with two functions. The compiler will now allow the function call `obj_B.f()`.

You can overload virtual functions in the same way. The following example demonstrates this:

```
#include <iostream>
using namespace std;

struct A {
    virtual void f() { cout << "void A::f()" << endl; }
    virtual void f(int) { cout << "void A::f(int)" << endl; }
};

struct B : A {
    using A::f;
    void f(int) { cout << "void B::f(int)" << endl; }
};

int main() {
    B obj_B;
    B* pb = &obj_B;
    pb->f(3);
    pb->f();
}
```

The following is the output of the above example:

```
void B::f(int)
void A::f()
```

Suppose that you introduce a function `f` from a base class A a derived class B with a using declaration, and there exists a function named `B::f` that has the same parameter types as `A::f`. Function `B::f` will hide, rather than conflict with, function `A::f`. The following example demonstrates this:

```
#include <iostream>
using namespace std;

struct A {
    void f() { }
    void f(int) { cout << "void A::f(int)" << endl; }
};
```

```

struct B : A {
    using A::f;
    void f(int) { cout << "void B::f(int)" << endl; }
};

int main() {
    B obj_B;
    obj_B.f(3);
}

```

The following is the output of the above example:

```
void B::f(int)
```

Related information

- Chapter 10, “Overloading (C++ only),” on page 225
- “Name hiding (C++ only)” on page 6
- “The using declaration and class members (C++ only)” on page 280

Changing the access of a class member (C++ only)

Suppose class B is a direct base class of class A. To restrict access of class B to the members of class A, derive B from A using either the access specifiers `protected` or `private`.

To increase the access of a member `x` of class A inherited from class B, use a `using` declaration. You cannot restrict the access to `x` with a `using` declaration. You may increase the access of the following members:

- A member inherited as `private`. (You cannot increase the access of a member declared as `private` because a `using` declaration must have access to the member’s name.)
- A member either inherited or declared as `protected`

The following example demonstrates this:

```

struct A {
protected:
    int y;
public:
    int z;
};

struct B : private A { };

struct C : private A {
public:
    using A::y;
    using A::z;
};

struct D : private A {
protected:
    using A::y;
    using A::z;
};

struct E : D {
    void f() {
        y = 1;
        z = 2;
    }
};

```

```

struct F : A {
public:
    using A::y;
private:
    using A::z;
};

int main() {
    B obj_B;
    // obj_B.y = 3;
    // obj_B.z = 4;

    C obj_C;
    obj_C.y = 5;
    obj_C.z = 6;

    D obj_D;
    // obj_D.y = 7;
    // obj_D.z = 8;

    F obj_F;
    obj_F.y = 9;
    obj_F.z = 10;
}

```

The compiler would not allow the following assignments from the above example:

- `obj_B.y = 3` and `obj_B.z = 4`: Members `y` and `z` have been inherited as `private`.
- `obj_D.y = 7` and `obj_D.z = 8`: Members `y` and `z` have been inherited as `private`, but their access have been changed to `protected`.

The compiler allows the following statements from the above example:

- `y = 1` and `z = 2` in `D::f()`: Members `y` and `z` have been inherited as `private`, but their access have been changed to `protected`.
- `obj_C.y = 5` and `obj_C.z = 6`: Members `y` and `z` have been inherited as `private`, but their access have been changed to `public`.
- `obj_F.y = 9`: The access of member `y` has been changed from `protected` to `public`.
- `obj_F.z = 10`: The access of member `z` is still `public`. The `private using` declaration using `A::z` has no effect on the access of `z`.

Related information

- “Member access (C++ only)” on page 265
- “Inherited member access (C++ only)” on page 278

Multiple inheritance (C++ only)

You can derive a class from any number of base classes. Deriving a class from more than one direct base class is called *multiple inheritance*.

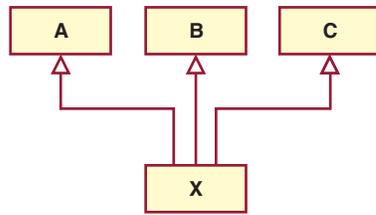
In the following example, classes `A`, `B`, and `C` are direct base classes for the derived class `X`:

```

class A { /* ... */ };
class B { /* ... */ };
class C { /* ... */ };
class X : public A, private B, public C { /* ... */ };

```

The following *inheritance graph* describes the inheritance relationships of the above example. An arrow points to the direct base class of the class at the tail of the arrow:



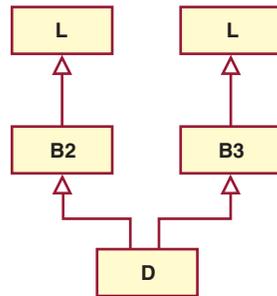
The order of derivation is relevant only to determine the order of default initialization by constructors and cleanup by destructors.

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 subobjects of class L exist, and both are accessible through class D. You can avoid this ambiguity by referring to class L using a qualified class name. For example:

B2::L

or

B3::L.

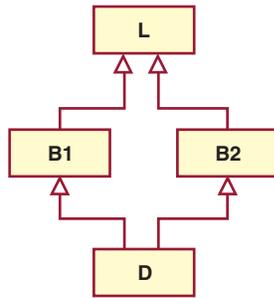
You can also avoid this ambiguity by using the base specifier `virtual` to declare a base class, as described in “Derivation (C++ only)” on page 275.

Virtual base classes (C++ only)

Suppose you have two derived classes B and C that have a common base class A, and you also have another class D that inherits from B and C. You can declare the base class A as *virtual* to ensure that B and C share the same subobject of A.

In the following example, an object of class D has two distinct subobjects 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 to indicate that only one subobject of type L, shared by class B1 and class B2, exists.

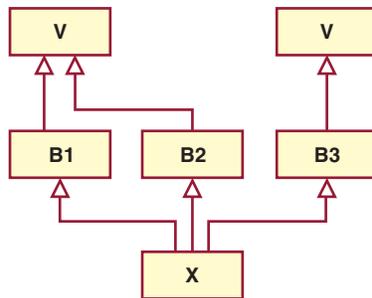
For example:



```
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 subobject 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 X : public B1, public B2, public B3 { /* ... */ };
```

In the above example, class `X` has two subobjects of class `V`, one that is shared by classes `B1` and `B2` and one through class `B3`.

Related information

- “Derivation (C++ only)” on page 275

Multiple access (C++ only)

In an inheritance graph containing virtual base classes, a name that can be reached through more than one path is accessed through the path that gives the most access.

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() is accessed through B2
        // and is public
        L::f();
    }
};

```

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.

Related information

- “Member access (C++ only)” on page 265
- “Protected members (C++ only)” on page 278
- “Access control of base class members (C++ only)” on page 279

Ambiguous base classes (C++ only)

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 or object. The declaration of a member with an ambiguous name in a derived class is not an error. The ambiguity is only flagged as an error if you use the ambiguous member name.

For example, suppose that two classes named `A` and `B` both have a member named `x`, and a class named `C` inherits from both `A` and `B`. An attempt to access `x` from class `C` would be ambiguous. You can resolve ambiguity by qualifying a member with its class name using the scope resolution (`::`) operator.

CCNX14G

```

class B1 {
public:
    int i;
    int j;
    void g(int) { }
};

class B2 {
public:
    int j;
    void g() { }
};

class D : public B1, public B2 {
public:
    int i;
};

int main() {
    D dobj;
    D *dptr = &dobj;
    dptr->i = 5;
    // dptr->j = 10;
}

```

```

    dptr->B1::j = 10;
    // dobj.g();
    dobj.B2::g();
}

```

The statement `dptr->j = 10` is ambiguous because the name `j` appears both in `B1` and `B2`. The statement `dobj.g()` is ambiguous because the name `g` appears both in `B1` and `B2`, even though `B1::g(int)` and `B2::g()` have different parameters.

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.

Name hiding

Suppose two subobjects named `A` and `B` both have a member name `x`. The member name `x` of subobject `B` *hides* the member name `x` of subobject `A` if `A` is a base class of `B`. The following example demonstrates this:

```

struct A {
    int x;
};

struct B: A {
    int x;
};

struct C: A, B {
    void f() { x = 0; }
};

int main() {
    C i;
    i.f();
}

```

The assignment `x = 0` in function `C::f()` is not ambiguous because the declaration `B::x` has hidden `A::x`. However, the compiler will warn you that deriving `C` from `A` is redundant because you already have access to the subobject `A` through `B`.

A base class declaration can be hidden along one path in the inheritance graph and not hidden along another path. The following example demonstrates this:

```

struct A { int x; };
struct B { int y; };
struct C: A, virtual B { };
struct D: A, virtual B {
    int x;
    int y;
};
struct E: C, D { };

int main() {
    E e;
    // e.x = 1;
    e.y = 2;
}

```

The assignment `e.x = 1` is ambiguous. The declaration `D::x` hides `A::x` along the path `D::A::x`, but it does not hide `A::x` along the path `C::A::x`. Therefore the variable `x` could refer to either `D::x` or `A::x`. The assignment `e.y = 2` is not ambiguous. The declaration `D::y` hides `B::y` along both paths `D::B::y` and `C::B::y` because `B` is a virtual base class.

Ambiguity and using declarations

Suppose you have a class named C that inherits from a class named A, and x is a member name of A. If you use a using declaration to declare A::x in C, then x is also a member of C; C::x does not hide A::x. Therefore using declarations cannot resolve ambiguities due to inherited members. The following example demonstrates this:

```
struct A {
    int x;
};

struct B: A { };

struct C: A {
    using A::x;
};

struct D: B, C {
    void f() { x = 0; }
};

int main() {
    D i;
    i.f();
}
```

The compiler will not allow the assignment `x = 0` in function `D::f()` because it is ambiguous. The compiler can find x in two ways: as `B::x` or as `C::x`.

Unambiguous class members

The compiler can unambiguously find static members, nested types, and enumerators defined in a base class A regardless of the number of subobjects of type A an object has. The following example demonstrates this:

```
struct A {
    int x;
    static int s;
    typedef A* Pointer_A;
    enum { e };
};

int A::s;

struct B: A { };

struct C: A { };

struct D: B, C {
    void f() {
        s = 1;
        Pointer_A pa;
        int i = e;
        // x = 1;
    }
};

int main() {
    D i;
    i.f();
}
```

The compiler allows the assignment `s = 1`, the declaration `Pointer_A pa`, and the statement `int i = e`. There is only one static variable `s`, only one typedef `Pointer_A`, and only one enumerator `e`. The compiler would not allow the assignment `x = 1` because `x` can be reached either from class B or class C.

Pointer conversions

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. (An *accessible base class* is a publicly derived base class that is neither hidden nor ambiguous in the inheritance hierarchy.) For example:

```
class W { /* ... */ };
class X : public W { /* ... */ };
class Y : public W { /* ... */ };
class Z : public X, public Y { /* ... */ };
int 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 { /* ... */ };
int main ()
{
    Z z;
    X* xptr = &z;           // valid
    Y* yptr = &z;           // valid
    W* wptr = &z;           // valid, W is virtual therefore only one
                           // W subobject exists
}
```

A pointer to a member of a base class can be converted 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.
- A pointer to the derived class can be converted to a pointer to the base class. If this is the case, the base class is said to be *accessible*.
- Member types must match. For example suppose class A is a base class of class B. You cannot convert a pointer to member of A of type int to a pointer to member of type B of type float.
- The base class cannot be virtual.

Overload resolution

Overload resolution takes place *after* the compiler unambiguously finds a given function name. The following example demonstrates this:

```
struct A {
    int f() { return 1; }
};

struct B {
    int f(int arg) { return arg; }
};

struct C: A, B {
    int g() { return f(); }
};
```

The compiler will not allow the function call to `f()` in `C::g()` because the name `f` has been declared both in `A` and `B`. The compiler detects the ambiguity error before overload resolution can select the base match `A::f()`.

Related information

- “Scope resolution operator `::` (C++ only)” on page 116
- “Virtual base classes (C++ only)” on page 285

Virtual functions (C++ only)

By default, C++ matches a function call with the correct function definition at compile time. This is called *static binding*. You can specify that the compiler match a function call with the correct function definition at run time; this is called *dynamic binding*. You declare a function with the keyword `virtual` if you want the compiler to use dynamic binding for that specific function.

The following examples demonstrate the differences between static and dynamic binding. The first example demonstrates static binding:

```
#include <iostream>
using namespace std;

struct A {
    void f() { cout << "Class A" << endl; }
};

struct B: A {
    void f() { cout << "Class B" << endl; }
};

void g(A& arg) {
    arg.f();
}

int main() {
    B x;
    g(x);
}
```

The following is the output of the above example:

```
Class A
```

When function `g()` is called, function `A::f()` is called, although the argument refers to an object of type `B`. At compile time, the compiler knows only that the argument of function `g()` will be a reference to an object derived from `A`; it cannot determine whether the argument will be a reference to an object of type `A` or type `B`. However, this can be determined at run time. The following example is the same as the previous example, except that `A::f()` is declared with the `virtual` keyword:

```
#include <iostream>
using namespace std;

struct A {
    virtual void f() { cout << "Class A" << endl; }
};

struct B: A {
    void f() { cout << "Class B" << endl; }
};

void g(A& arg) {
    arg.f();
}
```

```

}

int main() {
    B x;
    g(x);
}

```

The following is the output of the above example:

```
Class B
```

The `virtual` keyword indicates to the compiler that it should choose the appropriate definition of `f()` not by the type of reference, but by the type of object that the reference refers to.

Therefore, a *virtual function* is a member function you may redefine for other derived classes, and can ensure that the compiler will call the redefined virtual function for an object of the corresponding derived class, even if you call that function with a pointer or reference to a base class of the object.

A class that declares or inherits a virtual function is called a *polymorphic class*.

You redefine a virtual member function, like any member function, in any derived class. Suppose you declare a virtual function named `f` in a class `A`, and you derive directly or indirectly from `A` a class named `B`. If you declare a function named `f` in class `B` with the same name and same parameter list as `A::f`, then `B::f` is also virtual (regardless whether or not you declare `B::f` with the `virtual` keyword) and it *overrides* `A::f`. However, if the parameter lists of `A::f` and `B::f` are different, `A::f` and `B::f` are considered different, `B::f` does not override `A::f`, and `B::f` is not virtual (unless you have declared it with the `virtual` keyword). Instead `B::f` *hides* `A::f`. The following example demonstrates this:

```

#include <iostream>
using namespace std;

struct A {
    virtual void f() { cout << "Class A" << endl; }
};

struct B: A {
    void f(int) { cout << "Class B" << endl; }
};

struct C: B {
    void f() { cout << "Class C" << endl; }
};

int main() {
    B b; C c;
    A* pa1 = &b;
    A* pa2 = &c;
    // b.f();
    pa1->f();
    pa2->f();
}

```

The following is the output of the above example:

```
Class A
Class C
```

The function `B::f` is not virtual. It hides `A::f`. Thus the compiler will not allow the function call `b.f()`. The function `C::f` is virtual; it overrides `A::f` even though `A::f` is not visible in `C`.

If you declare a base class destructor as virtual, a derived class destructor will override that base class destructor, even though destructors are not inherited.

The return type of an overriding virtual function may differ from the return type of the overridden virtual function. This overriding function would then be called a *covariant virtual function*. Suppose that `B::f` overrides the virtual function `A::f`. The return types of `A::f` and `B::f` may differ if all the following conditions are met:

- The function `B::f` returns a reference or pointer to a class of type `T`, and `A::f` returns a pointer or a reference to an unambiguous direct or indirect base class of `T`.
- The `const` or `volatile` qualification of the pointer or reference returned by `B::f` has the same or less `const` or `volatile` qualification of the pointer or reference returned by `A::f`.
- The return type of `B::f` must be complete at the point of declaration of `B::f`, or it can be of type `B`.

The following example demonstrates this:

```
#include <iostream>
using namespace std;

struct A { };

class B : private A {
    friend class D;
    friend class F;
};

A global_A;
B global_B;

struct C {
    virtual A* f() {
        cout << "A* C::f()" << endl;
        return &global_A;
    }
};

struct D : C {
    B* f() {
        cout << "B* D::f()" << endl;
        return &global_B;
    }
};

struct E;

struct F : C {

    // Error:
    // E is incomplete
    // E* f();
};

struct G : C {

    // Error:
    // A is an inaccessible base class of B
    // B* f();
};
```

```

};

int main() {
    D d;
    C* cp = &d;
    D* dp = &d;

    A* ap = cp->f();
    B* bp = dp->f();
};

```

The following is the output of the above example:

```

B* D::f()
B* D::f()

```

The statement `A* ap = cp->f()` calls `D::f()` and converts the pointer returned to type `A*`. The statement `B* bp = dp->f()` calls `D::f()` as well but does not convert the pointer returned; the type returned is `B*`. The compiler would not allow the declaration of the virtual function `F::f()` because `E` is not a complete class. The compiler would not allow the declaration of the virtual function `G::f()` because class `A` is not an accessible base class of `B` (unlike friend classes `D` and `F`, the definition of `B` does not give access to its members for class `G`).

A virtual function cannot be global or static because, 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.

If a function is declared virtual in its base class, you can still access it directly using the scope resolution (`::`) operator. In this case, the virtual function call mechanism is suppressed and the function implementation defined in the base class is used. In addition, if you do not override a virtual member function in a derived class, a call to that function uses the function implementation defined in the base class.

A virtual function must be one of the following:

- Defined
- Declared pure
- Defined and declared pure

A base class containing one or more pure virtual member functions is called an *abstract class*.

Related information

- “Abstract classes (C++ only)” on page 296

Ambiguous virtual function calls (C++ only)

You cannot 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() { }
};

// Error:
// Both A::f() and B::f() try to override V::f()
class D : public A, public B { };

int main() {
    D d;
    V* vptr = &d;

    // which f(), A::f() or B::f()?
    vptr->f();
}

```

The compiler will not allow the definition of class D. 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() will try to override V::f(). This attempt is not allowed because it is not possible to decide which function to call if a D object is referenced with a pointer to class V, as shown in the above example. 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, class D has two separate subobjects of class A:

```

#include <iostream>
using namespace std;

struct A {
    virtual void f() { cout << "A::f()" << endl; };
};

struct B : A {
    void f() { cout << "B::f()" << endl; };
};

struct C : A {
    void f() { cout << "C::f()" << endl; };
};

struct D : B, C { };

int main() {
    D d;

    B* bp = &d;
    A* ap = bp;
    D* dp = &d;

    ap->f();
    // dp->f();
}

```

Class D has two occurrences of class A, one inherited from B, and another inherited from C. Therefore there are also two occurrences of the virtual function A::f. The statement ap->f() calls D::B::f. However the compiler would not allow the statement dp->f() because it could either call D::B::f or D::C::f.

Virtual function access (C++ only)

The access for a virtual function is specified when it is declared. The access rules for a virtual function are not affected by the access rules for the function that later overrides the virtual function. In general, the access of the overriding member function is not known.

If a virtual function is called with a pointer or reference to a class object, the type of the class object is not used to determine the access of the virtual function. Instead, the type of the pointer or reference to the class object is used.

In the following example, when the function `f()` is called using a pointer having type `B*`, `bptr` is used to determine the access to the function `f()`. Although the definition of `f()` 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();
};

int main() {
    D dobj;
    B* bptr = &dobj;
    D* dptr = &dobj;

    // valid, virtual B::f() is public,
    // D::f() is called
    bptr->f();

    // error, D::f() is private
    dptr->f();
}
```

Abstract classes (C++ only)

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*. You declare a pure virtual function by using a *pure specifier* (`= 0`) in the declaration of a virtual member function in the class declaration.

The following is an example of an abstract class:

```
class AB {
public:
    virtual void f() = 0;
};
```

Function `AB::f` is a pure virtual function. A function declaration cannot have both a pure specifier and a definition. For example, the compiler will not allow the following:

```
struct A {
    virtual void g() { } = 0;
};
```

You cannot use an abstract class as a parameter type, a function return type, or the type of an explicit conversion, nor can you declare an object of an abstract class. You can, however, declare pointers and references to an abstract class. The following example demonstrates this:

```
struct A {
    virtual void f() = 0;
};

struct B : A {
    virtual void f() { }
};

// Error:
// Class A is an abstract class
// A g();

// Error:
// Class A is an abstract class
// void h(A);
A& i(A&);

int main() {

// Error:
// Class A is an abstract class
// A a;

    A* pa;
    B b;

// Error:
// Class A is an abstract class
// static_cast<A>(b);
}
```

Class A is an abstract class. The compiler would not allow the function declarations `A g()` or `void h(A)`, declaration of object `a`, nor the static cast of `b` to type `A`.

Virtual member functions are inherited. A class derived from an abstract base class will also be abstract unless you override each pure virtual function in the derived class.

For example:

```
class AB {
public:
    virtual void f() = 0;
};

class D2 : public AB {
    void g();
};

int main() {
    D2 d;
}
```

The compiler will not allow the declaration of object `d` because `D2` is an abstract class; it inherited the pure virtual function `f()` from `AB`. The compiler will allow the declaration of object `d` if you define function `D2::g()`.

Note that you can derive an abstract class from a nonabstract class, and you can override a non-pure virtual function with a pure virtual function.

You can call member functions from a constructor or destructor of an abstract class. However, the results of calling (directly or indirectly) a pure virtual function from its constructor are undefined. The following example demonstrates this:

```
struct A {
    A() {
        direct();
        indirect();
    }
    virtual void direct() = 0;
    virtual void indirect() { direct(); }
};
```

The default constructor of A calls the pure virtual function `direct()` both directly and indirectly (through `indirect()`).

Related information

- “Virtual functions (C++ only)” on page 291
- “Virtual function access (C++ only)” on page 296

Chapter 14. Special member functions (C++ only)

The default constructor, destructor, copy constructor, and copy assignment operator are *special member functions*. These functions create, destroy, convert, initialize, and copy class objects, and are discussed in the following sections:

- “Overview of constructors and destructors (C++ only)”
- “Constructors (C++ only)” on page 301
- “Destructors (C++ only)” on page 308
- “Conversion constructors (C++ only)” on page 312
- “Conversion functions (C++ only)” on page 314
- “Copy constructors (C++ only)” on page 315

Overview of constructors and destructors (C++ only)

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. Unlike other member functions, constructors can have member initialization lists. The following restrictions apply to constructors and destructors:

- Constructors and destructors do not have return types nor can they return values.
- References and pointers cannot be used on constructors and destructors because their addresses cannot be taken.
- Constructors cannot be declared with the keyword `virtual`.
- Constructors and destructors cannot be declared `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 protected access, only derived classes and friends can use it to create class objects.

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 its `this` pointer refers to, but may allocate storage for more objects than its class object refers to. If memory allocation is required for objects, constructors can explicitly call the `new` operator. During cleanup, a destructor may release objects allocated by the corresponding constructor. To release objects, use the `delete` operator.

Derived classes do not inherit or overload constructors or destructors from their base classes, but they do call the constructor and destructor of base classes. Destructors can be declared with the keyword `virtual`.

Constructors are also called when local or temporary class objects are created, and 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 of a class A. In this case, the function called is the one defined in A or a base class of A, but not a function overridden in any class derived from A. This avoids the possibility of accessing an unconstructed object from a constructor or destructor. The following example demonstrates this:

```
#include <iostream>
using namespace std;

struct A {
    virtual void f() { cout << "void A::f()" << endl; }
    virtual void g() { cout << "void A::g()" << endl; }
    virtual void h() { cout << "void A::h()" << endl; }
};

struct B : A {
    virtual void f() { cout << "void B::f()" << endl; }
    B() {
        f();
        g();
        h();
    }
};

struct C : B {
    virtual void f() { cout << "void C::f()" << endl; }
    virtual void g() { cout << "void C::g()" << endl; }
    virtual void h() { cout << "void C::h()" << endl; }
};

int main() {
    C obj;
}
```

The following is the output of the above example:

```
void B::f()
void A::g()
void A::h()
```

The constructor of B does not call any of the functions overridden in C because C has been derived from B, although the example creates an object of type C named obj.

You can use the typeid or the dynamic_cast operator in constructors or destructors, as well as member initializers of constructors.

Related information

- “new expressions (C++ only)” on page 151
- “delete expressions (C++ only)” on page 155

Constructors (C++ only)

A *constructor* is a member function with the same name as its class. For example:

```
class X {
public:
    X();      // constructor for class X
};
```

Constructors are used to create, and can initialize, objects of their class type.

You cannot declare a constructor as `virtual` or `static`, nor can you declare a constructor as `const`, `volatile`, or `const volatile`.

You do not specify a return type for a constructor. A return statement in the body of a constructor cannot have a return value.

Default constructors (C++ only)

A *default constructor* is a constructor that either has no parameters, or if it has parameters, *all* the parameters have default values.

If no user-defined constructor exists for a class *A* and one is needed, the compiler implicitly *declares* a default parameterless constructor `A::A()`. This constructor is an inline public member of its class. The compiler will implicitly *define* `A::A()` when the compiler uses this constructor to create an object of type *A*. The constructor will have no constructor initializer and a null body.

The compiler first implicitly defines the implicitly declared constructors of the base classes and nonstatic data members of a class *A* before defining the implicitly declared constructor of *A*. No default constructor is created for a class that has any constant or reference type members.

A constructor of a class *A* is *trivial* if all the following are true:

- It is implicitly defined
- *A* has no virtual functions and no virtual base classes
- All the direct base classes of *A* have trivial constructors
- The classes of all the nonstatic data members of *A* have trivial constructors

If any of the above are false, then the constructor is *nontrivial*.

A union member cannot be of a class type that has a nontrivial constructor.

Like all functions, a constructor can have default arguments. They are used to initialize member objects. If default values are supplied, the trailing arguments can be omitted in the expression list of the constructor. Note that if a constructor has any arguments that do not have default values, it is not a default constructor.

A *copy constructor* for a class *A* is a constructor whose first parameter is of type `A&`, `const A&`, `volatile A&`, or `const volatile A&`. Copy constructors are used to make a copy of one class object from another class object of the same 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 parameters as long as they all have default arguments. If a user-defined copy constructor does not exist for a class and one is needed, the compiler implicitly creates a copy constructor, with public access, for that class. A copy constructor is not created 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:

    // default constructor, no arguments
    X();

    // constructor
    X(int, int , int = 0);

    // copy constructor
    X(const X&);

    // error, incorrect argument type
    X(X);
};

class Y {
public:

    // default constructor with one
    // default argument
    Y( int = 0);

    // default argument
    // copy constructor
    Y(const Y&, int = 0);
};
```

Related information

- “Copy constructors (C++ only)” on page 315

Explicit initialization with constructors (C++ only)

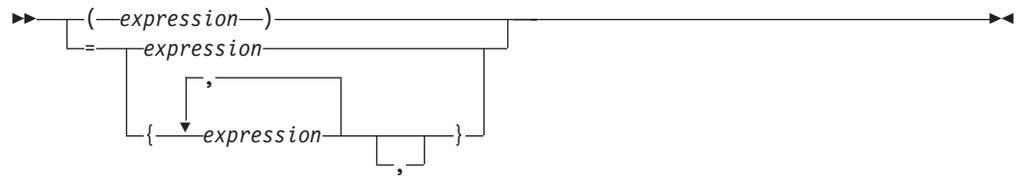
A class object with a constructor must be explicitly initialized or have a default constructor. Except for aggregate initialization, explicit initialization using a constructor is the only way to initialize non-static constant and reference class members.

| A class object that has no user-declared constructors, no virtual functions, no
| private or protected non-static data members, and no base classes is called an
| *aggregate*. Examples of aggregates are C-style structures and unions.

You explicitly initialize a class object when you create that object. There are two ways to initialize a class object:

- Using a parenthesized expression list. The compiler calls the constructor of the class using this list as the constructor’s argument list.
- Using a single initialization value and the = operator. Because this type of expression is an initialization, not an assignment, the assignment operator function, if one exists, is not called. 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.

Initializer syntax



The following example shows the declaration and use of several constructors that explicitly initialize class objects:

CCNX13A

```
// This example illustrates explicit initialization
// by constructor.
#include <iostream>
using namespace std;

class complx {
    double re, im;
public:

    // default constructor
    complx() : re(0), im(0) { }

    // copy constructor
    complx(const complx& c) { re = c.re; im = c.im; }

    // constructor with default trailing argument
    complx( double r, double i = 0.0) { re = r; im = i; }

    void display() {
        cout << "re = " << re << " im = " << im << endl;
    }
};

int main() {

    // initialize with complx(double, double)
    complx one(1);

    // initialize with a copy of one
    // using complx::complx(const complx&)
    complx two = one;

    // construct complx(3,4)
    // directly into three
    complx three = complx(3,4);

    // initialize with default constructor
    complx four;

    // complx(double, double) and construct
    // directly into five
    complx five = 5;

    one.display();
    two.display();
    three.display();
    four.display();
    five.display();
}

```

The above 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

```

Related information

- “Initializers” on page 88

Initialization of base classes and members (C++ only)

Constructors can initialize their members in two different ways. A constructor can use the arguments passed to it to initialize member variables in the constructor definition:

```

complex(double r, double i = 0.0) { re = r; im = i; }

```

Or a constructor can have an *initializer list* within the definition but prior to the constructor body:

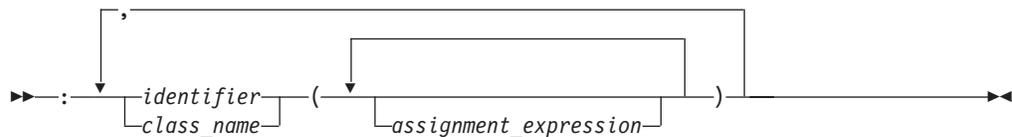
```

complex(double r, double i = 0) : re(r), im(i) { /* ... */ }

```

Both methods assign the argument values to the appropriate data members of the class.

Initializer list syntax



Include the initialization list as part of the constructor definition, not as part of the constructor declaration. For example:

```

#include <iostream>
using namespace std;

class B1 {
    int b;
public:
    B1() { cout << "B1::B1()" << endl; };

    // inline constructor
    B1(int i) : b(i) { cout << "B1::B1(int)" << endl; }
};

class B2 {
    int b;
protected:
    B2() { cout << "B2::B2()" << endl; }

    // noninline constructor
    B2(int i);
};

// B2 constructor definition including initialization list
B2::B2(int i) : b(i) { cout << "B2::B2(int)" << endl; }

class D : public B1, public B2 {
    int d1, d2;
public:
    D(int i, int j) : B1(i+1), B2(), d1(i) {

```

```

        cout << "D1::D1(int, int)" << endl;
        d2 = j;}
};

int main() {
    D obj(1, 2);
}

```

The following is the output of the above example:

```

B1::B1(int)
B1::B1()
D1::D1(int, int)

```

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. In the above example, if you leave out the call `B2()` in the constructor of class `D` (as shown below), a constructor initializer with an empty expression list is automatically created to initialize `B2`. The constructors for class `D`, shown above and below, result in the same construction of an object of class `D`:

```

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) {
        cout << "D1::D1(int, int)" << endl;
        d2 = j;}
};

```

In the above example, the compiler will automatically call the default constructor for `B2()`.

Note that you must declare constructors as `public` or `protected` to enable a derived class to call them. For example:

```

class B {
    B() { }
};

class D : public B {

    // error: implicit call to private B() not allowed
    D() { }
};

```

The compiler does not allow the definition of `D::D()` because this constructor cannot access the private constructor `B::B()`.

You must initialize the following with an initializer list: base classes with no default constructors, reference data members, non-static `const` data members, or a class type which contains a constant data member. The following example demonstrates this:

```

class A {
public:
    A(int) { }
};

class B : public A {
    static const int i;
    const int j;
    int &k;
public:

```

```

    B(int& arg) : A(0), j(1), k(arg) { }
};

int main() {
    int x = 0;
    B obj(x);
};

```

The data members `j` and `k`, as well as the base class `A` must be initialized in the initializer list of the constructor of `B`.

You can use data members when initializing members of a class. The following example demonstrate this:

```

struct A {
    int k;
    A(int i) : k(i) { }
};
struct B: A {
    int x;
    int i;
    int j;
    int& r;
    B(int i): r(x), A(i), j(this->i), i(i) { }
};

```

The constructor `B(int i)` initializes the following:

- `B::r` to refer to `B::x`
- Class `A` with the value of the argument to `B(int i)`
- `B::j` with the value of `B::i`
- `B::i` with the value of the argument to `B(int i)`

You can also call member functions (including virtual member functions) or use the operators `typeid` or `dynamic_cast` when initializing members of a class. However if you perform any of these operations in a member initialization list before all base classes have been initialized, the behavior is undefined. The following example demonstrates this:

```

#include <iostream>
using namespace std;

struct A {
    int i;
    A(int arg) : i(arg) {
        cout << "Value of i: " << i << endl;
    }
};

struct B : A {
    int j;
    int f() { return i; }
    B();
};

B::B() : A(f()), j(1234) {
    cout << "Value of j: " << j << endl;
}

int main() {
    B obj;
}

```

The output of the above example would be similar to the following:

```
Value of i: 8
Value of j: 1234
```

The behavior of the initializer `A(f())` in the constructor of `B` is undefined. The run time will call `B::f()` and try to access `A::i` even though the base `A` has not been initialized.

The following example is the same as the previous example except that the initializers of `B::B()` have different arguments:

```
#include <iostream>
using namespace std;

struct A {
    int i;
    A(int arg) : i(arg) {
        cout << "Value of i: " << i << endl;
    }
};

struct B : A {
    int j;
    int f() { return i; }
    B();
};

B::B() : A(5678), j(f()) {
    cout << "Value of j: " << j << endl;
}

int main() {
    B obj;
}
```

The following is the output of the above example:

```
Value of i: 5678
Value of j: 5678
```

The behavior of the initializer `j(f())` in the constructor of `B` is well-defined. The base class `A` is already initialized when `B::j` is initialized.

Related information

- “The typeid operator (C++ only)” on page 122
- “The dynamic_cast operator (C++ only)” on page 149

Construction order of derived class objects (C++ only)

When a derived class object is created using constructors, it is created in the following order:

1. Virtual base classes are initialized, in the order they appear in the base list.
2. Nonvirtual base classes are initialized, in declaration order.
3. Class members are initialized in declaration order (regardless of their order in the initialization list).
4. The body of the constructor is executed.

The following example demonstrates this:

```
#include <iostream>
using namespace std;
struct V {
    V() { cout << "V()" << endl; }
```

```

};
struct V2 {
    V2() { cout << "V2()" << endl; }
};
struct A {
    A() { cout << "A()" << endl; }
};
struct B : virtual V {
    B() { cout << "B()" << endl; }
};
struct C : B, virtual V2 {
    C() { cout << "C()" << endl; }
};
struct D : C, virtual V {
    A obj_A;
    D() { cout << "D()" << endl; }
};
int main() {
    D c;
}

```

The following is the output of the above example:

```

V()
V2()
B()
C()
A()
D()

```

The above output lists the order in which the C++ run time calls the constructors to create an object of type D.

Related information

- “Virtual base classes (C++ only)” on page 285

Destructors (C++ only)

Destructors are usually used to deallocate memory and do other cleanup for a class object and its class members when the object is destroyed. A destructor is called for a class object when that object passes out of scope or is explicitly deleted.

A destructor is a member function with the same name as its class prefixed by a ~ (tilde). For example:

```

class X {
public:
    // Constructor for class X
    X();
    // Destructor for class X
    ~X();
};

```

A destructor takes no arguments and has no return type. Its address cannot be taken. Destructors cannot be declared `const`, `volatile`, `const volatile` or `static`. A destructor can be declared `virtual` or `pure virtual`.

If no user-defined destructor exists for a class and one is needed, the compiler implicitly declares a destructor. This implicitly declared destructor is an inline public member of its class.

The compiler will implicitly define an implicitly declared destructor when the compiler uses the destructor to destroy an object of the destructor's class type. Suppose a class A has an implicitly declared destructor. The following is equivalent to the function the compiler would implicitly define for A:

```
A::~~A() { }
```

The compiler first implicitly defines the implicitly declared destructors of the base classes and nonstatic data members of a class A before defining the implicitly declared destructor of A

A destructor of a class A is *trivial* if all the following are true:

- It is implicitly defined
- All the direct base classes of A have trivial destructors
- The classes of all the nonstatic data members of A have trivial destructors

If any of the above are false, then the destructor is *nontrivial*.

A union member cannot be of a class type that has a nontrivial destructor.

Class members that are class types can have their own destructors. Both base and derived classes can have destructors, although destructors are not inherited. If a base class A or a member of A has a destructor, and a class derived from A does not declare a destructor, a default destructor is generated.

The default destructor calls the destructors of the base class and members of the derived class.

The destructors of base classes and members are called in the reverse order of the completion of their constructor:

1. The destructor for a class object is called before destructors for members and bases are called.
2. Destructors for nonstatic members are called before destructors for base classes are called.
3. Destructors for nonvirtual base classes are called before destructors for virtual base classes are called.

When an exception is thrown for a class object with a destructor, the destructor for the temporary object thrown is not called until control passes out of the catch block.

Destructors are implicitly called when an automatic object (a local object that has been declared `auto` or `register`, or not declared as `static` or `extern`) or temporary object passes out of scope. They are implicitly called at program termination for constructed external and static objects. Destructors are invoked when you use the `delete` operator for objects created with the `new` operator.

For example:

```
#include <string>

class Y {
private:
    char * string;
    int number;
public:
    // Constructor
    Y(const char*, int);
    // Destructor
```

```

    ~Y() { delete[] string; }
};

// Define class Y constructor
Y::Y(const char* n, int a) {
    string = strcpy(new char[strlen(n) + 1 ], n);
    number = a;
}

int main () {
    // Create and initialize
    // object of class Y
    Y yobj = Y("somestring", 10);

    // ...

    // Destructor ~Y is called before
    // control returns from main()
}

```

You can use a destructor explicitly to destroy objects, although this practice is not recommended. However to destroy an object created with the placement `new` operator, you can explicitly call the object's destructor. The following example demonstrates this:

```

#include <new>
#include <iostream>
using namespace std;
class A {
public:
    A() { cout << "A::A()" << endl; }
    ~A() { cout << "A::~A()" << endl; }
};
int main () {
    char* p = new char[sizeof(A)];
    A* ap = new (p) A;
    ap->A::~A();
    delete [] p;
}

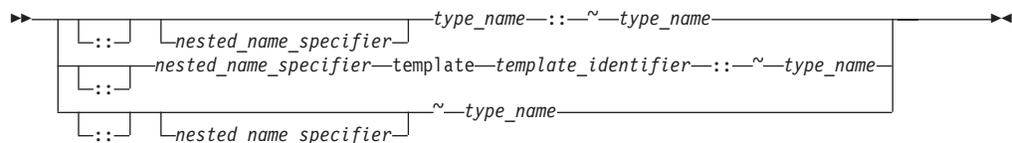
```

The statement `A* ap = new (p) A` dynamically creates a new object of type `A` not in the free store but in the memory allocated by `p`. The statement `delete [] p` will delete the storage allocated by `p`, but the run time will still believe that the object pointed to by `ap` still exists until you explicitly call the destructor of `A` (with the statement `ap->A::~A()`).

Pseudo-destructors (C++ only)

A *pseudo-destructor* is a destructor of a nonclass type.

Pseudo-destructor syntax



The following example calls the pseudo destructor for an integer type:

```

typedef int I;
int main() {
    I x = 10;
    x.I::~~I();
    x = 20;
}

```

The call to the pseudo destructor, `x.I::~~I()`, has no effect at all. Object `x` has not been destroyed; the assignment `x = 20` is still valid. Because pseudo destructors require the syntax for explicitly calling a destructor for a nonclass type to be valid, you can write code without having to know whether or not a destructor exists for a given type.

Related information

- Chapter 12, “Class members and friends (C++ only),” on page 251
- “Scope of class names (C++ only)” on page 245

User-defined conversions (C++ only)

User-defined conversions allow you to specify object conversions with constructors or with conversion functions. User-defined conversions are implicitly used 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 constructors
- Conversion functions

The compiler can use only one user-defined conversion (either a conversion constructor or a conversion function) when implicitly converting a single value. The following example demonstrates this:

```

class A {
    int x;
public:
    operator int() { return x; };
};

class B {
    A y;
public:
    operator A() { return y; };
};

int main () {
    B b_obj;
    // int i = b_obj;
    int j = A(b_obj);
}

```

The compiler would not allow the statement `int i = b_obj`. The compiler would have to implicitly convert `b_obj` into an object of type `A` (with `B::operator A()`), then implicitly convert that object to an integer (with `A::operator int()`). The statement `int j = A(b_obj)` explicitly converts `b_obj` into an object of type `A`, then implicitly converts that object to an integer.

User-defined conversions must be unambiguous, or they are not called. A conversion function in a derived class does not hide another conversion function in

a base class unless both conversion functions convert to the same type. Function overload resolution selects the most appropriate conversion function. The following example demonstrates this:

```
class A {
    int a_int;
    char* a_carp;
public:
    operator int() { return a_int; }
    operator char*() { return a_carp; }
};

class B : public A {
    float b_float;
    char* b_carp;
public:
    operator float() { return b_float; }
    operator char*() { return b_carp; }
};

int main () {
    B b_obj;
    // long a = b_obj;
    char* c_p = b_obj;
}
```

The compiler would not allow the statement `long a = b_obj`. The compiler could either use `A::operator int()` or `B::operator float()` to convert `b_obj` into a `long`. The statement `char* c_p = b_obj` uses `B::operator char*()` to convert `b_obj` into a `char*` because `B::operator char*()` hides `A::operator char*()`.

When you call a constructor with an argument and you have not defined a constructor accepting that argument type, only standard conversions are used to convert the argument to another argument type acceptable to a constructor for that class. No other constructors or conversions functions are called to convert the argument to a type acceptable to a constructor defined for that class. The following example demonstrates this:

```
class A {
public:
    A() { }
    A(int) { }
};

int main() {
    A a1 = 1.234;
    // A moocow = "text string";
}
```

The compiler allows the statement `A a1 = 1.234`. The compiler uses the standard conversion of converting 1.234 into an `int`, then implicitly calls the converting constructor `A(int)`. The compiler would not allow the statement `A moocow = "text string"`; converting a text string to an integer is not a standard conversion.

Related information

- Chapter 5, “Type conversions,” on page 103

Conversion constructors (C++ only)

A *conversion constructor* is a single-parameter constructor that is declared without the function specifier `explicit`. The compiler uses conversion constructors to convert objects from the type of the first parameter to the type of the conversion constructor’s class. The following example demonstrates this:

```

class Y {
    int a, b;
    char* name;
public:
    Y(int i) { };
    Y(const char* n, int j = 0) { };
};

void add(Y) { };

int main() {

    // equivalent to
    // obj1 = Y(2)
    Y obj1 = 2;

    // equivalent to
    // obj2 = Y("somestring",0)
    Y obj2 = "somestring";

    // equivalent to
    // obj1 = Y(10)
    obj1 = 10;

    // equivalent to
    // add(Y(5))
    add(5);
}

```

The above example has the following two conversion constructors:

- `Y(int i)` which is used to convert integers to objects of class `Y`.
- `Y(const char* n, int j = 0)` which is used to convert pointers to strings to objects of class `Y`.

The compiler will not implicitly convert types as demonstrated above with constructors declared with the `explicit` keyword. The compiler will only use explicitly declared constructors in new expressions, the `static_cast` expressions and explicit casts, and the initialization of bases and members. The following example demonstrates this:

```

class A {
public:
    explicit A() { };
    explicit A(int) { };
};

int main() {
    A z;
    // A y = 1;
    A x = A(1);
    A w(1);
    A* v = new A(1);
    A u = (A)1;
    A t = static_cast<A>(1);
}

```

The compiler would not allow the statement `A y = 1` because this is an implicit conversion; class `A` has no conversion constructors.

A copy constructor is a conversion constructor.

Related information

- “new expressions (C++ only)” on page 151

- “The `static_cast` operator (C++ only)” on page 145

The explicit specifier (C++ only)

The `explicit` function specifier controls unwanted implicit type conversions. It can only be used in declarations of constructors within a class declaration. For example, except for the default constructor, the constructors in the following class are converting constructors.

```
class A
{ public:
  A();
  A(int);
  A(const char*, int = 0);
};
```

The following declarations are legal.

```
A c = 1;
A d = "Venditti";
```

The first declaration is equivalent to `A c = A(1)`.

If you declare the constructor of the class with the `explicit` keyword, the previous declarations would be illegal.

For example, if you declare the class as:

```
class A
{ public:
  explicit A();
  explicit A(int);
  explicit A(const char*, int = 0);
};
```

You can only assign values that match the values of the class type.

For example, the following statements will be legal:

```
A a1;
A a2 = A(1);
A a3(1);
A a4 = A("Venditti");
A* p = new A(1);
A a5 = (A)1;
A a6 = static_cast<A>(1);
```

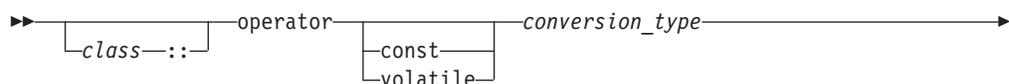
Related information

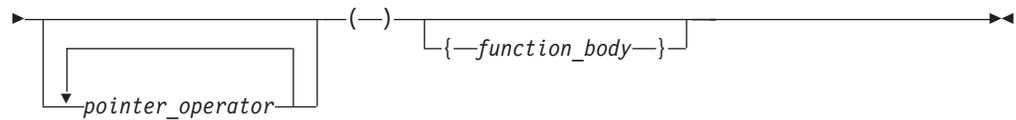
- “Conversion constructors (C++ only)” on page 312

Conversion functions (C++ only)

You can define a member function of a class, called a *conversion function*, that converts from the type of its class to another specified type.

Conversion function syntax





A conversion function that belongs to a class *X* specifies a conversion from the class type *X* to the type specified by the *conversion_type*. The following code fragment shows a conversion function called `operator int()`:

```
class Y {
    int b;
public:
    operator int();
};
Y::operator int() {
    return b;
}
void f(Y obj) {
    int i = int(obj);
    int j = (int)obj;
    int k = i + obj;
}
```

All three statements in function `f(Y)` use the conversion function `Y::operator int()`.

Classes, enumerations, typedef names, function types, or array types cannot be declared or defined in the *conversion_type*. You cannot use a conversion function to convert an object of type *A* to type *A*, to a base class of *A*, or to `void`.

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.

Related information

- Chapter 5, “Type conversions,” on page 103
- “User-defined conversions (C++ only)” on page 311
- “Conversion constructors (C++ only)” on page 312
- Chapter 5, “Type conversions,” on page 103

Copy constructors (C++ only)

The *copy constructor* lets you create a new object from an existing one by initialization. A copy constructor of a class *A* is a non-template constructor in which the first parameter is of type `A&`, `const A&`, `volatile A&`, or `const volatile A&`, and the rest of its parameters (if there are any) have default values.

If you do not declare a copy constructor for a class *A*, the compiler will implicitly declare one for you, which will be an inline public member.

The following example demonstrates implicitly defined and user-defined copy constructors:

```
#include <iostream>
using namespace std;

struct A {
    int i;
    A() : i(10) { }
```

```

};

struct B {
    int j;
    B() : j(20) {
        cout << "Constructor B(), j = " << j << endl;
    }

    B(B& arg) : j(arg.j) {
        cout << "Copy constructor B(B&), j = " << j << endl;
    }

    B(const B&, int val = 30) : j(val) {
        cout << "Copy constructor B(const B&, int), j = " << j << endl;
    }
};

struct C {
    C() { }
    C(C&) { }
};

int main() {
    A a;
    A a1(a);
    B b;
    const B b_const;
    B b1(b);
    B b2(b_const);
    const C c_const;
    // C c1(c_const);
}

```

The following is the output of the above example:

```

Constructor B(), j = 20
Constructor B(), j = 20
Copy constructor B(B&), j = 20
Copy constructor B(const B&, int), j = 30

```

The statement `A a1(a)` creates a new object from `a` with an implicitly defined copy constructor. The statement `B b1(b)` creates a new object from `b` with the user-defined copy constructor `B::B(B&)`. The statement `B b2(b_const)` creates a new object with the copy constructor `B::B(const B&, int)`. The compiler would not allow the statement `C c1(c_const)` because a copy constructor that takes as its first parameter an object of type `const C&` has not been defined.

The implicitly declared copy constructor of a class `A` will have the form `A::A(const A&)` if the following are true:

- The direct and virtual bases of `A` have copy constructors whose first parameters have been qualified with `const` or `const volatile`
- The nonstatic class type or array of class type data members of `A` have copy constructors whose first parameters have been qualified with `const` or `const volatile`

If the above are not true for a class `A`, the compiler will implicitly declare a copy constructor with the form `A::A(A&)`.

The compiler cannot allow a program in which the compiler must implicitly define a copy constructor for a class `A` and one or more of the following are true:

- Class `A` has a nonstatic data member of a type which has an inaccessible or ambiguous copy constructor.

- Class A is derived from a class which has an inaccessible or ambiguous copy constructor.

The compiler will implicitly define an implicitly declared constructor of a class A if you initialize an object of type A or an object derived from class A.

An implicitly defined copy constructor will copy the bases and members of an object in the same order that a constructor would initialize the bases and members of the object.

Related information

- “Overview of constructors and destructors (C++ only)” on page 299

Copy assignment operators (C++ only)

The *copy assignment operator* lets you create a new object from an existing one by initialization. A copy assignment operator of a class A is a nonstatic non-template member function that has one of the following forms:

- `A::operator=(A)`
- `A::operator=(A&)`
- `A::operator=(const A&)`
- `A::operator=(volatile A&)`
- `A::operator=(const volatile A&)`

If you do not declare a copy assignment operator for a class A, the compiler will implicitly declare one for you which will be inline public.

The following example demonstrates implicitly defined and user-defined copy assignment operators:

```
#include <iostream>
using namespace std;

struct A {
    A& operator=(const A&) {
        cout << "A::operator=(const A&)" << endl;
        return *this;
    }

    A& operator=(A&) {
        cout << "A::operator=(A&)" << endl;
        return *this;
    }
};

class B {
    A a;
};

struct C {
    C& operator=(C&) {
        cout << "C::operator=(C&)" << endl;
        return *this;
    }
    C() { }
};

int main() {
    B x, y;
    x = y;
```

```

A w, z;
w = z;

C i;
const C j();
// i = j;
}

```

The following is the output of the above example:

```

A::operator=(const A&)
A::operator=(A&)

```

The assignment `x = y` calls the implicitly defined copy assignment operator of `B`, which calls the user-defined copy assignment operator `A::operator=(const A&)`. The assignment `w = z` calls the user-defined operator `A::operator=(A&)`. The compiler will not allow the assignment `i = j` because an operator `C::operator=(const C&)` has not been defined.

The implicitly declared copy assignment operator of a class `A` will have the form `A& A::operator=(const A&)` if the following are true:

- A direct or virtual base `B` of class `A` has a copy assignment operator whose parameter is of type `const B&`, `const volatile B&`, or `B`.
- A non-static class type data member of type `X` that belongs to class `A` has a copy constructor whose parameter is of type `const X&`, `const volatile X&`, or `X`.

If the above are not true for a class `A`, the compiler will implicitly declare a copy assignment operator with the form `A& A::operator=(A&)`.

The implicitly declared copy assignment operator returns a reference to the operator's argument.

The copy assignment operator of a derived class hides the copy assignment operator of its base class.

The compiler cannot allow a program in which the compiler must implicitly define a copy assignment operator for a class `A` and one or more of the following are true:

- Class `A` has a nonstatic data member of a `const` type or a reference type
- Class `A` has a nonstatic data member of a type which has an inaccessible copy assignment operator
- Class `A` is derived from a base class with an inaccessible copy assignment operator.

An implicitly defined copy assignment operator of a class `A` will first assign the direct base classes of `A` in the order that they appear in the definition of `A`. Next, the implicitly defined copy assignment operator will assign the nonstatic data members of `A` in the order of their declaration in the definition of `A`.

Related information

- "Assignment operators" on page 128

Chapter 15. Templates (C++ only)

A *template* describes a set of related classes or set of related functions in which a list of parameters in the declaration describe how the members of the set vary. The compiler generates new classes or functions when you supply arguments for these parameters; this process is called *template instantiation*, and is described in detail in “Template instantiation (C++ only)” on page 338. This class or function definition generated from a template and a set of template parameters is called a *specialization*, as described in “Template specialization (C++ only)” on page 341.

Template declaration syntax

→ `template` ← `template_parameter_list` → `declaration` →
└── export ─┘

The compiler accepts and silently ignores the `export` keyword on a template.

The *template_parameter_list* is a comma-separated list of template parameters, which are described in “Template parameters (C++ only)” on page 320.

The *declaration* is one of the following::

- a declaration or definition of a function or a class
- a definition of a member function or a member class of a class template
- a definition of a static data member of a class template
- a definition of a static data member of a class nested within a class template
- a definition of a member template of a class or class template

The *identifier* of a *type* is defined to be a *type_name* in the scope of the template declaration. A template declaration can appear as a namespace scope or class scope declaration.

The following example demonstrates the use of a class template:

```
template<class T> class Key
{
    T k;
    T* kptr;
    int length;
public:
    Key(T);
    // ...
};
```

Suppose the following declarations appear later:

```
Key<int> i;
Key<char*> c;
Key<mytype> m;
```

The compiler would create three instances of `class Key`. The following table shows the definitions of these three class instances if they were written out in source form as regular classes, not as templates:

<code>class Key<int> i;</code>	<code>class Key<char*> c;</code>	<code>class Key<mytype> m;</code>
<pre>class Key { int k; int * kptr; int length; public: Key(int); // ... };</pre>	<pre>class Key { char* k; char** kptr; int length; public: Key(char*); // ... };</pre>	<pre>class Key { mytype k; mytype* kptr; int length; public: Key(mytype); // ... };</pre>

Note that these three classes have different names. The arguments contained within the angle braces are not just the arguments to the class names, but part of the class names themselves. `Key<int>` and `Key<char*>` are class names.

Template parameters (C++ only)

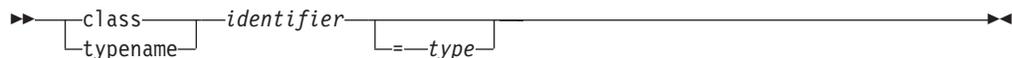
There are three kinds of template parameters:

- Type template parameters (C++ only)
- Non-type template parameters (C++ only)
- Template template parameters (C++ only)

You can interchange the keywords `class` and `typename` in a template parameter declaration. You cannot use storage class specifiers (`static` and `auto`) in a template parameter declaration.

Type template parameters (C++ only)

Type template parameter declaration syntax



The *identifier* is the name of a type.

Related information

- “The `typename` keyword (C++ only)” on page 349

Non-type template parameters (C++ only)

The syntax of a non-type template parameter is the same as a declaration of one of the following types:

- integral or enumeration
- pointer to object or pointer to function
- reference to object or reference to function
- pointer to member

Non-type template parameters that are declared as arrays or functions are converted to pointers or pointers to functions, respectively. The following example demonstrates this:

```
template<int a[4]> struct A { };
template<int f(int)> struct B { };

int i;
```

```
int g(int) { return 0;}

A<&i> x;
B<&g> y;
```

The type deduced from `&i` is `int *`, and the type deduced from `&g` is `int (*)(int)`.

You may qualify a non-type template parameter with `const` or `volatile`.

You cannot declare a non-type template parameter as a floating point, class, or void type.

Non-type template parameters are not lvalues.

Related information

- “Type qualifiers” on page 67
- “Lvalues and rvalues” on page 111

Template template parameters (C++ only)

Template template parameter declaration syntax

```
template<template-parameter-list> class [identifier] [==id-expression]
```

The following example demonstrates a declaration and use of a template template parameter:

```
template<template <class T> class X> class A { };
template<class T> class B { };

A<B> a;
```

Related information

- “Template parameters (C++ only)” on page 320

Default arguments for template parameters (C++ only)

Template parameters may have default arguments. The set of default template arguments accumulates over all declarations of a given template. The following example demonstrates this:

```
template<class T, class U = int> class A;
template<class T = float, class U> class A;

template<class T, class U> class A {
public:
    T x;
    U y;
};

A<> a;
```

The type of member `a.x` is `float`, and the type of `a.y` is `int`.

You cannot give default arguments to the same template parameters in different declarations in the same scope. For example, the compiler will not allow the following:

```
template<class T = char> class X;  
template<class T = char> class X { };
```

If one template parameter has a default argument, then all template parameters following it must also have default arguments. For example, the compiler will not allow the following:

```
template<class T = char, class U, class V = int> class X { };
```

Template parameter U needs a default argument or the default for T must be removed.

The scope of a template parameter starts from the point of its declaration to the end of its template definition. This implies that you may use the name of a template parameter in other template parameter declarations and their default arguments. The following example demonstrates this:

```
template<class T = int> class A;  
template<class T = float> class B;  
template<class V, V obj> class C;  
// a template parameter (T) used as the default argument  
// to another template parameter (U)  
template<class T, class U = T> class D { };
```

Related information

- “Template parameters (C++ only)” on page 320

Template arguments (C++ only)

There are three kinds of template arguments corresponding to the three types of template parameters:

- Template type arguments (C++ only)
- Template non-type arguments (C++ only)
- Template template arguments (C++ only)

A template argument must match the type and form specified by the corresponding parameter declared in the template.

To use the default value of a template parameter, you omit the corresponding template argument. However, even if all template parameters have defaults, you still must use the <> brackets. For example, the following will yield a syntax error:

```
template<class T = int> class X { };  
X<> a;  
X b;
```

The last declaration, X b, will yield an error.

Related information

Template type arguments (C++ only)

You cannot use one of the following as a template argument for a type template parameter:

- a local type
- a type with no linkage
- an unnamed type
- a type compounded from any of the above types

If it is ambiguous whether a template argument is a type or an expression, the template argument is considered to be a type. The following example demonstrates this:

```
template<class T> void f() { };
template<int i> void f() { };

int main() {
    f<int>();
}
```

The function call `f<int>()` calls the function with `T` as a template argument – the compiler considers `int()` as a type – and therefore implicitly instantiates and calls the first `f()`.

Related information

- “Block/local scope” on page 2
- “No linkage” on page 9
- “Bit field members” on page 57
- “typedef definitions” on page 65

Template non-type arguments (C++ only)

A non-type template argument 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 or objects with external linkage, or addresses of static class members. Non-type template arguments are normally used to initialize a class or to specify the sizes of class members.

For non-type integral arguments, the instance argument matches the corresponding template parameter as long as the instance argument has a value and sign appropriate to the parameter type.

For non-type address arguments, the type of the instance argument must be of the form *identifier* or *&identifier*, and the type of the instance argument must match the template parameter exactly, except that a function name is changed to a pointer to function type before matching.

The resulting values of non-type template arguments within a template argument list form part of the template class 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 non-type 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
};
```

In this example, the template argument `size` becomes a part of the template class name. An object of such a template class is created with both the type argument `T` of the class and the value of the non-type template argument `size`.

An object `x`, and its corresponding template class with arguments `double` and `size=200`, can be created from this template with a value as its second template argument:

```
Myfilebuf<double,200> x;
```

`x` can also be created using an arithmetic expression:

```
Myfilebuf<double,10*20> x;
```

The objects 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 to prevent either symbol from being parsed as a template argument list delimiter when it is in fact being used as a relational operator. For example, the arguments in the following definition are valid:

```
Myfilebuf<double, (75>25)> x;          // valid
```

The following definition, however, is not valid because the greater than operator (`>`) is interpreted as the closing delimiter of the template argument list:

```
Myfilebuf<double, 75>25> x;          // error
```

If the template arguments do not evaluate identically, the objects 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`.

The following two objects:

```
Myfilebuf<double, 128> x
Myfilebuf<double, 512> y
```

are objects of separate template specializations. Referring 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 non-type arguments. For example, the following template is a valid class template:

```
template<int i> class C
{
    public:
        int k;
        C() { k = i; }
};
```

This class template can be instantiated by declarations such as:

```
class C<100>;
class C<200>;
```

Again, these two declarations refer to distinct classes because the values of their non-type arguments differ.

Related information

- “Integer constant expressions” on page 113
- “References (C++ only)” on page 87
- “External linkage” on page 8
- “Static members (C++ only)” on page 260

Template template arguments (C++ only)

A template argument for a template template parameter is the name of a class template.

When the compiler tries to find a template to match the template template argument, it only considers primary class templates. (A *primary template* is the template that is being specialized.) The compiler will not consider any partial specialization even if their parameter lists match that of the template template parameter. For example, the compiler will not allow the following code:

```
template<class T, int i> class A {
    int x;
};

template<class T> class A<T, 5> {
    short x;
};

template<template<class T> class U> class B1 { };

B1<A> c;
```

The compiler will not allow the declaration `B1<A> c`. Although the partial specialization of `A` seems to match the template template parameter `U` of `B1`, the compiler considers only the primary template of `A`, which has different template parameters than `U`.

The compiler considers the partial specializations based on a template template argument once you have instantiated a specialization based on the corresponding template template parameter. The following example demonstrates this:

```
#include <iostream>
using namespace std;

template<class T, class U> class A {
    int x;
};

template<class U> class A<int, U> {
    short x;
};

template<template<class T, class U> class V> class B {
    V<int, char> i;
    V<char, char> j;
};

B<A> c;

int main() {
    cout << typeid(c.i.x).name() << endl;
    cout << typeid(c.j.x).name() << endl;
}
```

The following is the output of the above example:

```
short  
int
```

The declaration `V<int, char> i` uses the partial specialization while the declaration `V<char, char> j` uses the primary template.

Related information

- “Partial specialization (C++ only)” on page 346
- “Template instantiation (C++ only)” on page 338

Class templates (C++ only)

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. You cannot declare an object of a class template.

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 class template definition is preceded by

```
template< template-parameter-list >
```

where *template-parameter-list* is a comma-separated list of one or more of the following kinds of template parameters:

- type
 - non-type
 - template
- Types, variables, constants and objects within the class template can be declared using the template parameters as well as explicit types (for example, `int` or `char`).

A class template can be declared without being defined by using an elaborated type specifier. For example:

```
template<class L, class T> class Key;
```

This 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. Only one template declaration containing the class definition is allowed.

Note: When you have nested template argument lists, you must have a separating space between the `>` at the end of the inner list and the `>` at the end of the outer list. Otherwise, there is an ambiguity between the extraction operator `>>` and two template list delimiters `>`.

```
template<class L, class T> class Key { /* ... */};  
template<class L> class Vector { /* ... */};
```

```

int main ()
{
    class Key <int, Vector<int> > my_key_vector;
    // implicitly instantiates template
}

```

Objects and function members of individual template classes can be accessed by any of the techniques used to access ordinary class member objects and functions. Given a class template:

```

template<class T> class Vehicle
{
public:
    Vehicle() { /* ... */ } // constructor
    ~Vehicle() {}; // destructor
    T kind[16];
    T* drive();
    static void roadmap();
    // ...
};

```

and the declaration:

```
Vehicle<char> bicycle; // instantiates the template
```

the constructor, the constructed object, and the member function drive() can be accessed with any of the following (assuming 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();

Related information

- “Declaring class types (C++ only)” on page 242
- “Scope of class names (C++ only)” on page 245

Class template declarations and definitions (C++ only)

A class template must be declared before any instantiation of a corresponding template class. A class template definition can only appear once in any single translation unit. A class template must be defined before any use of a template class that requires the size of the class or refers to members of the class.

In the following example, the class template Key is declared before it is defined. The declaration of the pointer keyiptr is valid because the size of the class is not needed. 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 a template class is used before the corresponding class template is defined, the compiler issues an error. A class name with the appearance of a template class name is considered 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 previous example uses the elaborated type specifier `class` to declare the class template `key` and the pointer `keyiptr`. The declaration of `keyiptr` can also be made without the elaborated type specifier.

```

template <class L> class Key; // class template declared,
// not defined yet
//
Key<int> *keyiptr; // declaration of pointer
//
Key<int> keyi; // error, cannot declare keyi
// without knowing size
//
template <class L> class Key // now class template defined
{ /* ... */ };

```

z/OS only

In the z/OS implementation, the compiler checks the syntax of the entire template class definition when the template include files are being compiled if the `TEMPINC` compiler option is used, or during the original compiler pass if the `NOTEMPINC` compiler option is used. Any errors in the class definition are flagged. The compiler does not generate code or data until it requires a specialization. At that point it generates appropriate code and data for the specialization by using the argument list supplied.

End of z/OS only

Related information

- “Class templates (C++ only)” on page 326

Static data members and templates (C++ only)

Each class template instantiation has its own copy of any static data members. The static declaration can be of template argument type or of any defined type.

You must separately define static members. The following example demonstrates this:

```

template <class T> class K
{
public:
    static T x;
};
template <class T> T K<T> ::x;

int main()
{
    K<int>::x = 0;
}

```

The statement `template T K::x` defines the static member of class `K`, while the statement in the `main()` function assigns a value to the data member for `K <int>`.

Related information

- “Static members (C++ only)” on page 260

Member functions of class templates (C++ only)

You may define a template member function outside of its class template definition.

When you call a member function of a class template specialization, the compiler will use the template arguments that you used to generate the class template. The following example demonstrates this:

```
template<class T> class X {
    public:
        T operator+(T);
};

template<class T> T X<T>::operator+(T arg1) {
    return arg1;
};

int main() {
    X<char> a;
    X<int> b;
    a + 'z';
    b + 4;
}
```

The overloaded addition operator has been defined outside of class X. The statement `a + 'z'` is equivalent to `a.operator+('z')`. The statement `b + 4` is equivalent to `b.operator+(4)`.

Related information

- “Member functions (C++ only)” on page 253

Friends and templates (C++ only)

There are four kinds of relationships between classes and their friends when templates are involved:

- *One-to-many*: A non-template function may be a friend to all template class instantiations.
- *Many-to-one*: All instantiations of a template function may be friends to a regular non-template class.
- *One-to-one*: A template function instantiated with one set of template arguments may be a friend to one template class instantiated with the same set of template arguments. This is also the relationship between a regular non-template class and a regular non-template friend function.
- *Many-to-many*: All instantiations of a template function may be a friend to all instantiations of the template class.

The following example demonstrates these relationships:

```
class B{
    template<class V> friend int j();
}

template<class S> g();

template<class T> class A {
    friend int e();
}
```

```

    friend int f(T);
    friend int g<T>();
    template<class U> friend int h();
};

```

- Function `e()` has a one-to-many relationship with class A. Function `e()` is a friend to all instantiations of class A.
- Function `f()` has a one-to-one relationship with class A. The compiler will give you a warning for this kind of declaration similar to the following:
 The friend function declaration "f" will cause an error when the enclosing template class is instantiated with arguments that declare a friend function that does not match an existing definition. The function declares only one function because it is not a template but the function type depends on one or more template parameters.
- Function `g()` has a one-to-one relationship with class A. Function `g()` is a function template. It must be declared before here or else the compiler will not recognize `g<T>` as a template name. For each instantiation of A there is one matching instantiation of `g()`. For example, `g<int>` is a friend of `A<int>`.
- Function `h()` has a many-to-many relationship with class A. Function `h()` is a function template. For all instantiations of A all instantiations of `h()` are friends.
- Function `j()` has a many-to-one relationship with class B.

These relationships also apply to friend classes.

Related information

- "Friends (C++ only)" on page 267

Function templates (C++ only)

A *function template* defines how a group of functions can be generated.

A non-template function is not related to a function template, even though the non-template function may have the same name and parameter profile as those of a specialization generated from a template. A non-template function is never considered to be a specialization of a function template.

The following example implements the quicksort algorithm with a function template named `quicksort`:

```

#include <iostream>
#include <cstdlib>
using namespace std;

template<class T> void quicksort(T a[], const int& leftarg, const int& rightarg)
{
    if (leftarg < rightarg) {

        T pivotvalue = a[leftarg];
        int left = leftarg - 1;
        int right = rightarg + 1;

        for(;;) {

            while (a[--right] > pivotvalue);
            while (a[++left] < pivotvalue);

            if (left >= right) break;

            T temp = a[right];
            a[right] = a[left];
            a[left] = temp;
        }
    }
}

```

```

        int pivot = right;
        quicksort(a, leftarg, pivot);
        quicksort(a, pivot + 1, rightarg);
    }
}

int main(void) {
    int sortme[10];

    for (int i = 0; i < 10; i++) {
        sortme[i] = rand();
        cout << sortme[i] << " ";
    };
    cout << endl;

    quicksort<int>(sortme, 0, 10 - 1);

    for (int i = 0; i < 10; i++) cout << sortme[i] << "
";
    cout << endl;
    return 0;
}

```

The above example will have output similar to the following:

```

16838 5758 10113 17515 31051 5627 23010 7419 16212 4086
4086 5627 5758 7419 10113 16212 16838 17515 23010 31051

```

This quicksort algorithm will sort an array of type T (whose relational and assignment operators have been defined). The template function takes one template argument and three function arguments:

- the type of the array to be sorted, T
- the name of the array to be sorted, a
- the lower bound of the array, leftarg
- the upper bound of the array, rightarg

In the above example, you can also call the `quicksort()` template function with the following statement:

```
quicksort(sortme, 0, 10 - 1);
```

You may omit any template argument if the compiler can deduce it by the usage and context of the template function call. In this case, the compiler deduces that `sortme` is an array of type `int`.

Template argument deduction (C++ only)

When you call a template function, you may omit any template argument that the compiler can determine or *deduce* by the usage and context of that template function call.

The compiler tries to deduce a template argument by comparing the type of the corresponding template parameter with the type of the argument used in the function call. The two types that the compiler compares (the template parameter and the argument used in the function call) must be of a certain structure in order for template argument deduction to work. The following lists these type structures:

```

T
const T
volatile T
T&
T*

```

```

T[10]
A<T>
C(*) (T)
T(*) ()
T(*) (U)
T C::*
C T::*
T U::*
T (C::* )()
C (T::* )()
D (C::* ) (T)
C (T::* ) (U)
T (C::* ) (U)
T (U::* ) ()
T (U::* ) (V)
E[10][i]
B<i>
TT<T>
TT<i>
TT<C>

```

- T, U, and V represent a template type argument
- 10 represents any integer constant
- i represents a template non-type argument
- [i] represents an array bound of a reference or pointer type, or a non-major array bound of a normal array.
- TT represents a template template argument
- (T), (U), and (V) represents an argument list that has at least one template type argument
- () represents an argument list that has no template arguments
- <T> represents a template argument list that has at least one template type argument
- <i> represents a template argument list that has at least one template non-type argument
- <C> represents a template argument list that has no template arguments dependent on a template parameter

The following example demonstrates the use of each of these type structures. The example declares a template function using each of the above structures as an argument. These functions are then called (without template arguments) in order of declaration. The example outputs the same list of type structures:

```

#include <iostream>
using namespace std;

template<class T> class A { };
template<int i> class B { };

class C {
public:
    int x;
};

class D {
public:
    C y;
    int z;
};

template<class T> void f (T)          { cout << "T" << endl; };
template<class T> void f1(const T)   { cout << "const T" << endl; };
template<class T> void f2(volatile T) { cout << "volatile T" << endl; };

```

```

template<class T> void g (T*)      { cout << "T*" << endl; };
template<class T> void g (T&)    { cout << "T&" << endl; };
template<class T> void g1(T[10]) { cout << "T[10]" << endl; };
template<class T> void h1(A<T>)  { cout << "A<T>" << endl; };

void test_1() {
    A<char> a;
    C c;

    f(c);  f1(c);  f2(c);
    g(c);  g(&c);  g1(&c);
    h1(a);
}

template<class T>      void j(C(*) (T)) { cout << "C(*) (T)" << endl; };
template<class T>      void j(T(*) ()) { cout << "T(*) ()" << endl; };
template<class T, class U> void j(T(*) (U)) { cout << "T(*) (U)" << endl; };

void test_2() {
    C (*c_pfunct1)(int);
    C (*c_pfunct2)(void);
    int (*c_pfunct3)(int);
    j(c_pfunct1);
    j(c_pfunct2);
    j(c_pfunct3);
}

template<class T>      void k(T C::*) { cout << "T C::*" << endl; };
template<class T>      void k(C T::*) { cout << "C T::*" << endl; };
template<class T, class U> void k(T U::*) { cout << "T U::*" << endl; };

void test_3() {
    k(&C::x);
    k(&D::y);
    k(&D::z);
}

template<class T>      void m(T (C::*) () )
    { cout << "T (C::*) ()" << endl; };
template<class T>      void m(C (T::*) () )
    { cout << "C (T::*) ()" << endl; };
template<class T>      void m(D (C::*) (T))
    { cout << "D (C::*) (T)" << endl; };
template<class T, class U> void m(C (T::*) (U))
    { cout << "C (T::*) (U)" << endl; };
template<class T, class U> void m(T (C::*) (U))
    { cout << "T (C::*) (U)" << endl; };
template<class T, class U> void m(T (U::*) () )
    { cout << "T (U::*) ()" << endl; };
template<class T, class U, class V> void m(T (U::*) (V))
    { cout << "T (U::*) (V)" << endl; };

void test_4() {
    int (C::*f_membp1)(void);
    C (D::*f_membp2)(void);
    D (C::*f_membp3)(int);
    m(f_membp1);
    m(f_membp2);
    m(f_membp3);

    C (D::*f_membp4)(int);
    int (C::*f_membp5)(int);
    int (D::*f_membp6)(void);
    m(f_membp4);
    m(f_membp5);
    m(f_membp6);
}

```

```

    int (D::*f_membp7)(int);
    m(f_membp7);
}

template<int i> void n(C[10][i]) { cout << "E[10][i]" << endl; };
template<int i> void n(B<i>)    { cout << "B<i>" << endl; };

void test_5() {
    C array[10][20];
    n(array);
    B<20> b;
    n(b);
}

template<template<class> class TT, class T> void p1(TT<T>)
    { cout << "TT<T>" << endl; };
template<template<int> class TT, int i>    void p2(TT<i>)
    { cout << "TT<i>" << endl; };
template<template<class> class TT>        void p3(TT<C>)
    { cout << "TT<C>" << endl; };

void test_6() {
    A<char> a;
    B<20> b;
    A<C> c;
    p1(a);
    p2(b);
    p3(c);
}

int main() { test_1(); test_2(); test_3(); test_4(); test_5(); test_6(); }

```

Deducing type template arguments

The compiler can deduce template arguments from a type composed of several of the listed type structures. The following example demonstrates template argument deduction for a type composed of several type structures:

```

template<class T> class Y { };

template<class T, int i> class X {
public:
    Y<T> f(char[20][i]) { return x; };
    Y<T> x;
};

template<template<class> class T, class U, class V, class W, int i>
    void g( T<U> (V::*)(W[20][i]) ) { };

int main()
{
    Y<int> (X<int, 20>::*p)(char[20][20]) = &X<int, 20>::f;
    g(p);
}

```

The type `Y<int> (X<int, 20>::*p)(char[20][20])T<U> (V::*)(W[20][i])` is based on the type structure `T (U::*)(V)`:

- T is `Y<int>`
- U is `X<int, 20>`
- V is `char[20][20]`

If you qualify a type with the class to which that type belongs, and that class (a *nested name specifier*) depends on a template parameter, the compiler will not deduce a template argument for that parameter. If a type contains a template

argument that cannot be deduced for this reason, all template arguments in that type will not be deduced. The following example demonstrates this:

```
template<class T, class U, class V>
    void h(typename Y<T>::template Z<U>, Y<T>, Y<V>) { };

int main() {
    Y<int>::Z<char> a;
    Y<int> b;
    Y<float> c;

    h<int, char, float>(a, b, c);
    h<int, char>(a, b, c);
    // h<int>(a, b, c);
}
```

The compiler will not deduce the template arguments T and U in typename Y<T>::template Z<U> (but it will deduce the T in Y<T>). The compiler would not allow the template function call h<int>(a, b, c) because U is not deduced by the compiler.

The compiler can deduce a function template argument from a pointer to function or pointer to member function argument given several overloaded function names. However, none of the overloaded functions may be function templates, nor can more than one overloaded function match the required type. The following example demonstrates this:

```
template<class T> void f(void(*) (T,int)) { };

template<class T> void g1(T, int) { };

void g2(int, int) { };
void g2(char, int) { };

void g3(int, int, int) { };
void g3(float, int) { };

int main() {
    // f(&g1);
    // f(&g2);
    f(&g3);
}
```

The compiler would not allow the call f(&g1) because g1() is a function template. The compiler would not allow the call f(&g2) because both functions named g2() match the type required by f().

The compiler cannot deduce a template argument from the type of a default argument. The following example demonstrates this:

```
template<class T> void f(T = 2, T = 3) { };

int main() {
    f(6);
    // f();
    f<int>();
}
```

The compiler allows the call f(6) because the compiler deduces the template argument (int) by the value of the function call's argument. The compiler would not allow the call f() because the compiler cannot deduce template argument from the default arguments of f().

The compiler cannot deduce a template type argument from the type of a non-type template argument. For example, the compiler will not allow the following:

```
template<class T, T i> void f(int[20][i]) { };

int main() {
    int a[20][30];
    f(a);
}
```

The compiler cannot deduce the type of template parameter T.

Deducing non-type template arguments

The compiler cannot deduce the value of a major array bound unless the bound refers to a reference or pointer type. Major array bounds are not part of function parameter types. The following code demonstrates this:

```
template<int i> void f(int a[10][i]) { };
template<int i> void g(int a[i]) { };
template<int i> void h(int (&a)[i]) { };

int main () {
    int b[10][20];
    int c[10];
    f(b);
    // g(c);
    h(c);
}
```

The compiler would not allow the call `g(c)`; the compiler cannot deduce template argument `i`.

The compiler cannot deduce the value of a non-type template argument used in an expression in the template function's parameter list. The following example demonstrates this:

```
template<int i> class X { };

template<int i> void f(X<i - 1>) { };

int main () {
    X<0> a;
    f<1>(a);
    // f(a);
}
```

In order to call function `f()` with object `a`, the function must accept an argument of type `X<0>`. However, the compiler cannot deduce that the template argument `i` must be equal to 1 in order for the function template argument type `X<i - 1>` to be equivalent to `X<0>`. Therefore the compiler would not allow the function call `f(a)`.

If you want the compiler to deduce a non-type template argument, the type of the parameter must match exactly the type of value used in the function call. For example, the compiler will not allow the following:

```
template<int i> class A { };
template<short d> void f(A<d>) { };

int main() {
    A<1> a;
    f(a);
}
```

The compiler will not convert `int` to `short` when the example calls `f()`.

However, deduced array bounds may be of any integral type.

Overloading function templates (C++ only)

You may overload a function template either by a non-template function or by another function template.

If you call the name of an overloaded function template, the compiler will try to deduce its template arguments and check its explicitly declared template arguments. If successful, it will instantiate a function template specialization, then add this specialization to the set of *candidate functions* used in overload resolution. The compiler proceeds with overload resolution, choosing the most appropriate function from the set of candidate functions. Non-template functions take precedence over template functions. The following example describes this:

```
#include <iostream>
using namespace std;

template<class T> void f(T x, T y) { cout << "Template" << endl; }

void f(int w, int z) { cout << "Non-template" << endl; }

int main() {
    f( 1, 2 );
    f('a', 'b');
    f( 1, 'b');
}
```

The following is the output of the above example:

```
Non-template
Template
Non-template
```

The function call `f(1, 2)` could match the argument types of both the template function and the non-template function. The non-template function is called because a non-template function takes precedence in overload resolution.

The function call `f('a', 'b')` can only match the argument types of the template function. The template function is called.

Argument deduction fails for the function call `f(1, 'b')`; the compiler does not generate any template function specialization and overload resolution does not take place. The non-template function resolves this function call after using the standard conversion from `char` to `int` for the function argument `'b'`.

Related information

- “Overload resolution (C++ only)” on page 236

Partial ordering of function templates (C++ only)

A function template specialization might be ambiguous because template argument deduction might associate the specialization with more than one of the overloaded definitions. The compiler will then choose the definition that is the most specialized. This process of selecting a function template definition is called *partial ordering*.

A template `X` is more specialized than a template `Y` if every argument list that matches the one specified by `X` also matches the one specified by `Y`, but not the other way around. The following example demonstrates partial ordering:

```

template<class T> void f(T) { }
template<class T> void f(T*) { }
template<class T> void f(const T*) { }

template<class T> void g(T) { }
template<class T> void g(T&) { }

template<class T> void h(T) { }
template<class T> void h(T, ...) { }

int main() {
    const int *p;
    f(p);

    int q;
    // g(q);
    // h(q);
}

```

The declaration `template<class T> void f(const T*)` is more specialized than `template<class T> void f(T*)`. Therefore, the function call `f(p)` calls `template<class T> void f(const T*)`. However, neither `void g(T)` nor `void g(T&)` is more specialized than the other. Therefore, the function call `g(q)` would be ambiguous.

Ellipses do not affect partial ordering. Therefore, the function call `h(q)` would also be ambiguous.

The compiler uses partial ordering in the following cases:

- Calling a function template specialization that requires overload resolution.
- Taking the address of a function template specialization.
- When a friend function declaration, an explicit instantiation, or explicit specialization refers to a function template specialization.
- Determining the appropriate deallocation function that is also a function template for a given placement operator `new`.

Related information

- “Template specialization (C++ only)” on page 341
- “new expressions (C++ only)” on page 151

Template instantiation (C++ only)

The act of creating a new definition of a function, class, or member of a class from a template declaration and one or more template arguments is called *template instantiation*. The definition created from a template instantiation to handle a specific set of template arguments is called a *specialization*.

Template instantiation declaration syntax

►►—extern—template—*template_declaration*—◄◄

The language feature is an orthogonal extension to Standard C++ for compatibility with GNU C++, and is described further in “Explicit instantiation (C++ only)” on page 340.

Related information

- “Template specialization (C++ only)” on page 341

Implicit instantiation (C++ only)

Unless a template specialization has been explicitly instantiated or explicitly specialized, the compiler will generate a specialization for the template only when it needs the definition. This is called *implicit instantiation*.

If the compiler must instantiate a class template specialization and the template is declared, you must also define the template.

For example, if you declare a pointer to a class, the definition of that class is not needed and the class will not be implicitly instantiated. The following example demonstrates when the compiler instantiates a template class:

```
template<class T> class X {
    public:
        X* p;
        void f();
        void g();
};

X<int>* q;
X<int> r;
X<float>* s;
r.f();
s->g();
```

The compiler requires the instantiation of the following classes and functions:

- `X<int>` when the object `r` is declared
- `X<int>::f()` at the member function call `r.f()`
- `X<float>` and `X<float>::g()` at the class member access function call `s->g()`

Therefore, the functions `X<T>::f()` and `X<T>::g()` must be defined in order for the above example to compile. (The compiler will use the default constructor of class `X` when it creates object `r`.) The compiler does not require the instantiation of the following definitions:

- `class X` when the pointer `p` is declared
- `X<int>` when the pointer `q` is declared
- `X<float>` when the pointer `s` is declared

The compiler will implicitly instantiate a class template specialization if it is involved in pointer conversion or pointer to member conversion. The following example demonstrates this:

```
template<class T> class B { };
template<class T> class D : public B<T> { };

void g(D<double>* p, D<int>* q)
{
    B<double>* r = p;
    delete q;
}
```

The assignment `B<double>* r = p` converts `p` of type `D<double>*` to a type of `B<double>*`; the compiler must instantiate `D<double>`. The compiler must instantiate `D<int>` when it tries to delete `q`.

If the compiler implicitly instantiates a class template that contains static members, those static members are not implicitly instantiated. The compiler will instantiate a static member only when the compiler needs the static member's definition. Every instantiated class template specialization has its own copy of static members. The following example demonstrates this:

```

template<class T> class X {
public:
    static T v;
};

template<class T> T X<T>::v = 0;

X<char*> a;
X<float> b;
X<float> c;

```

Object a has a static member variable v of type char*. Object b has a static variable v of type float. Objects b and c share the single static data member v.

An implicitly instantiated template is in the same namespace where you defined the template.

If a function template or a member function template specialization is involved with overload resolution, the compiler implicitly instantiates a declaration of the specialization.

Related information

- “Template instantiation (C++ only)” on page 338

Explicit instantiation (C++ only)

You can explicitly tell the compiler when it should generate a definition from a template. This is called *explicit instantiation*.

Explicit instantiation declaration syntax

►—template—*template_declaration*—►

The following are examples of explicit instantiations:

```

template<class T> class Array { void mf(); };
template class Array<char>; // explicit instantiation
template void Array<int>::mf(); // explicit instantiation

template<class T> void sort(Array<T>& v) { }
template void sort(Array<char>&); // explicit instantiation

namespace N {
    template<class T> void f(T&) { }
}

template void N::f<int>(int&);
// The explicit instantiation is in namespace N.

int* p = 0;
template<class T> T g(T = &p);
template char g(char); // explicit instantiation

template <class T> class X {
    private:
        T v(T arg) { return arg; };
};

template int X<int>::v(int); // explicit instantiation

template<class T> T g(T val) { return val;}
template<class T> void Array<T>::mf() { }

```

A template declaration must be in scope at the point of instantiation of the template's explicit instantiation. An explicit instantiation of a template specialization is in the same namespace where you defined the template.

Access checking rules do not apply to names in explicit instantiations. Template arguments and names in a declaration of an explicit instantiation may be private types or objects. In the above example, the compiler allows the explicit instantiation `template int X<int>::v(int)` even though the member function is declared private.

The compiler does not use default arguments when you explicitly instantiate a template. In the above example the compiler allows the explicit instantiation `template char g(char)` even though the default argument is an address of type `int`.

Template specialization (C++ only)

The act of creating a new definition of a function, class, or member of a class from a template declaration and one or more template arguments is called *template instantiation*. The definition created from a template instantiation is called a *specialization*. A *primary template* is the template that is being specialized.

Related information

- “Template instantiation (C++ only)” on page 338

Explicit specialization (C++ only)

When you instantiate a template with a given set of template arguments the compiler generates a new definition based on those template arguments. You can override this behavior of definition generation. You can instead specify the definition the compiler uses for a given set of template arguments. This is called *explicit specialization*. You can explicitly specialize any of the following:

- Function or class template
- Member function of a class template
- Static data member of a class template
- Member class of a class template
- Member function template of a class template
- Member class template of a class template

Explicit specialization declaration syntax

►►—template—<—>—*declaration_name*—<—*template_argument_list*—>—*declaration_body*—◄◄

The `template<>` prefix indicates that the following template declaration takes no template parameters. The *declaration_name* is the name of a previously declared template. Note that you can forward-declare an explicit specialization so the *declaration_body* is optional, at least until the specialization is referenced.

The following example demonstrates explicit specialization:

```
using namespace std;

template<class T = float, int i = 5> class A
{
    public:
        A();
};
```

```

        int value;
    };

    template<> class A<> { public: A(); };
    template<> class A<double, 10> { public: A(); };

    template<class T, int i> A<T, i>::A() : value(i) {
        cout << "Primary template, "
             << "non-type argument is " << value << endl;
    }

    A<>::A() {
        cout << "Explicit specialization "
             << "default arguments" << endl;
    }

    A<double, 10>::A() {
        cout << "Explicit specialization "
             << "<double, 10>" << endl;
    }

    int main() {
        A<int,6> x;
        A<> y;
        A<double, 10> z;
    }

```

The following is the output of the above example:

```

Primary template non-type argument is: 6
Explicit specialization default arguments
Explicit specialization <double, 10>

```

This example declared two explicit specializations for the *primary template* (the template which is being specialized) class A. Object x uses the constructor of the primary template. Object y uses the explicit specialization A<>::A(). Object z uses the explicit specialization A<double, 10>::A().

Related information

- “Function templates (C++ only)” on page 330
- “Class templates (C++ only)” on page 326
- “Member functions of class templates (C++ only)” on page 329
- “Static data members and templates (C++ only)” on page 328

Definition and declaration of explicit specializations

The definition of an explicitly specialized class is unrelated to the definition of the primary template. You do not have to define the primary template in order to define the specialization (nor do you have to define the specialization in order to define the primary template). For example, the compiler will allow the following:

```

template<class T> class A;
template<> class A<int>;

template<> class A<int> { /* ... */ };

```

The primary template is not defined, but the explicit specialization is.

You can use the name of an explicit specialization that has been declared but not defined the same way as an incompletely defined class. The following example demonstrates this:

```

template<class T> class X { };
template<> class X<char>;
X<char>* p;
X<int> i;
// X<char> j;

```

The compiler does not allow the declaration `X<char> j` because the explicit specialization of `X<char>` is not defined.

Explicit specialization and scope

A declaration of a primary template must be in scope at the *point of declaration* of the explicit specialization. In other words, an explicit specialization declaration must appear after the declaration of the primary template. For example, the compiler will not allow the following:

```

template<> class A<int>;
template<class T> class A;

```

An explicit specialization is in the same namespace as the definition of the primary template.

Class members of explicit specializations

A member of an explicitly specialized class is not implicitly instantiated from the member declaration of the primary template. You have to explicitly define members of a class template specialization. You define members of an explicitly specialized template class as you would normal classes, without the `template<>` prefix. In addition, you can define the members of an explicit specialization inline; no special template syntax is used in this case. The following example demonstrates a class template specialization:

```

template<class T> class A {
public:
    void f(T);
};

template<> class A<int> {
public:
    int g(int);
};

int A<int>::g(int arg) { return 0; }

int main() {
    A<int> a;
    a.g(1234);
}

```

The explicit specialization `A<int>` contains the member function `g()`, which the primary template does not.

If you explicitly specialize a template, a member template, or the member of a class template, then you must declare this specialization before that specialization is implicitly instantiated. For example, the compiler will not allow the following code:

```

template<class T> class A { };

void f() { A<int> x; }
template<> class A<int> { };

int main() { f(); }

```

The compiler will not allow the explicit specialization `template<> class A<int> { }`; because function `f()` uses this specialization (in the construction of `x`) before the specialization.

Explicit specialization of function templates

In a function template specialization, a template argument is optional if the compiler can deduce it from the type of the function arguments. The following example demonstrates this:

```
template<class T> class X { };
template<class T> void f(X<T>);
template<> void f(X<int>);
```

The explicit specialization `template<> void f(X<int>)` is equivalent to `template<> void f<int>(X<int>)`.

You cannot specify default function arguments in a declaration or a definition for any of the following:

- Explicit specialization of a function template
- Explicit specialization of a member function template

For example, the compiler will not allow the following code:

```
template<class T> void f(T a) { };
template<> void f<int>(int a = 5) { };

template<class T> class X {
    void f(T a) { }
};
template<> void X<int>::f(int a = 10) { };
```

Related information

- “Function templates (C++ only)” on page 330

Explicit specialization of members of class templates

Each instantiated class template specialization has its own copy of any static members. You may explicitly specialize static members. The following example demonstrates this:

```
template<class T> class X {
public:
    static T v;
    static void f(T);
};

template<class T> T X<T>::v = 0;
template<class T> void X<T>::f(T arg) { v = arg; }

template<> char* X<char*>::v = "Hello";
template<> void X<float>::f(float arg) { v = arg * 2; }

int main() {
    X<char*> a, b;
    X<float> c;
    c.f(10);
}
```

This code explicitly specializes the initialization of static data member `X::v` to point to the string "Hello" for the template argument `char*`. The function `X::f()` is explicitly specialized for the template argument `float`. The static data member `v` in objects `a` and `b` point to the same string, "Hello". The value of `c.v` is equal to 20 after the call function call `c.f(10)`.

You can nest member templates within many enclosing class templates. If you explicitly specialize a template nested within several enclosing class templates, you must prefix the declaration with `template<>` for every enclosing class template you specialize. You may leave some enclosing class templates unspecialized, however you cannot explicitly specialize a class template unless its enclosing class templates are also explicitly specialized. The following example demonstrates explicit specialization of nested member templates:

```
#include <iostream>
using namespace std;

template<class T> class X {
public:
    template<class U> class Y {
    public:
        template<class V> void f(U,V);
        void g(U);
    };
};

template<class T> template<class U> template<class V>
void X<T>::Y<U>::f(U, V) { cout << "Template 1" << endl; }

template<class T> template<class U>
void X<T>::Y<U>::g(U) { cout << "Template 2" << endl; }

template<> template<>
void X<int>::Y<int>::g(int) { cout << "Template 3" << endl; }

template<> template<> template<class V>
void X<int>::Y<int>::f(int, V) { cout << "Template 4" << endl; }

template<> template<> template<>
void X<int>::Y<int>::f<int>(int, int) { cout << "Template 5" << endl; }

// template<> template<class U> template<class V>
// void X<char>::Y<U>::f(U, V) { cout << "Template 6" << endl; }

// template<class T> template<>
// void X<T>::Y<float>::g(float) { cout << "Template 7" << endl; }

int main() {
    X<int>::Y<int> a;
    X<char>::Y<char> b;
    a.f(1, 2);
    a.f(3, 'x');
    a.g(3);
    b.f('x', 'y');
    b.g('z');
}
```

The following is the output of the above program:

```
Template 5
Template 4
Template 3
Template 1
Template 2
```

- The compiler would not allow the template specialization definition that would output "Template 6" because it is attempting to specialize a member (function `f()`) without specialization of its containing class (`Y`).
- The compiler would not allow the template specialization definition that would output "Template 7" because the enclosing class of class `Y` (which is class `X`) is not explicitly specialized.

A friend declaration cannot declare an explicit specialization.

Related information

- “Static data members and templates (C++ only)” on page 328

Partial specialization (C++ only)

When you instantiate a class template, the compiler creates a definition based on the template arguments you have passed. Alternatively, if all those template arguments match those of an explicit specialization, the compiler uses the definition defined by the explicit specialization.

A *partial specialization* is a generalization of explicit specialization. An explicit specialization only has a template argument list. A partial specialization has both a template argument list and a template parameter list. The compiler uses the partial specialization if its template argument list matches a subset of the template arguments of a template instantiation. The compiler will then generate a new definition from the partial specialization with the rest of the unmatched template arguments of the template instantiation.

You cannot partially specialize function templates.

Partial specialization syntax

```
▶▶—template—<template_parameter_list>—declaration_name—————▶
▶<template_argument_list>—declaration_body—————▶▶
```

The *declaration_name* is a name of a previously declared template. Note that you can forward-declare a partial specialization so that the *declaration_body* is optional.

The following demonstrates the use of partial specializations:

```
#include <iostream>
using namespace std;

template<class T, class U, int I> struct X
{ void f() { cout << "Primary template" << endl; } };

template<class T, int I> struct X<T, T*, I>
{ void f() { cout << "Partial specialization 1" << endl;
} };

template<class T, class U, int I> struct X<T*, U, I>
{ void f() { cout << "Partial specialization 2" << endl;
} };

template<class T> struct X<int, T*, 10>
{ void f() { cout << "Partial specialization 3" << endl;
} };

template<class T, class U, int I> struct X<T, U*, I>
{ void f() { cout << "Partial specialization 4" << endl;
} };

int main() {
    X<int, int, 10> a;
    X<int, int*, 5> b;
    X<int*, float, 10> c;
    X<int, char*, 10> d;
```

```

    X<float, int*, 10> e;
//   X<int, int*, 10> f;
    a.f(); b.f(); c.f(); d.f(); e.f();
}

```

The following is the output of the above example:

```

Primary template
Partial specialization 1
Partial specialization 2
Partial specialization 3
Partial specialization 4

```

The compiler would not allow the declaration `X<int, int*, 10> f` because it can match template struct `X<T, T*, I>`, template struct `X<int, T*, 10>`, or template struct `X<T, U*, I>`, and none of these declarations are a better match than the others.

Each class template partial specialization is a separate template. You must provide definitions for each member of a class template partial specialization.

Template parameter and argument lists of partial specializations

Primary templates do not have template argument lists; this list is implied in the template parameter list.

Template parameters specified in a primary template but not used in a partial specialization are omitted from the template parameter list of the partial specialization. The order of a partial specialization's argument list is the same as the order of the primary template's implied argument list.

In a template argument list of a partial template parameter, you cannot have an expression that involves non-type arguments unless that expression is only an identifier. In the following example, the compiler will not allow the first partial specialization, but will allow the second one:

```

template<int I, int J> class X { };

// Invalid partial specialization
template<int I> class X <I * 4, I + 3> { };

// Valid partial specialization
template <int I> class X <I, I> { };

```

The type of a non-type template argument cannot depend on a template parameter of a partial specialization. The compiler will not allow the following partial specialization:

```

template<class T, T i> class X { };

// Invalid partial specialization
template<class T> class X<T, 25> { };

```

A partial specialization's template argument list cannot be the same as the list implied by the primary template.

You cannot have default values in the template parameter list of a partial specialization.

Related information

- “Template parameters (C++ only)” on page 320
- “Template arguments (C++ only)” on page 322

Matching of class template partial specializations

The compiler determines whether to use the primary template or one of its partial specializations by matching the template arguments of the class template specialization with the template argument lists of the primary template and the partial specializations:

- If the compiler finds only one specialization, then the compiler generates a definition from that specialization.
- If the compiler finds more than one specialization, then the compiler tries to determine which of the specializations is the most specialized. A template *X* is more specialized than a template *Y* if every argument list that matches the one specified by *X* also matches the one specified by *Y*, but not the other way around. If the compiler cannot find the most specialized specialization, then the use of the class template is ambiguous; the compiler will not allow the program.
- If the compiler does not find any matches, then the compiler generates a definition from the primary template.

Name binding and dependent names (C++ only)

Name binding is the process of finding the declaration for each name that is explicitly or implicitly used in a template. The compiler may bind a name in the definition of a template, or it may bind a name at the instantiation of a template.

A *dependent name* is a name that depends on the type or the value of a template parameter. For example:

```
template<class T> class U : A<T>
{
    typename T::B x;
    void f(A<T>& y)
    {
        *y++;
    }
};
```

The dependent names in this example are the base class `A<T>`, the type name `T::B`, and the variable `y`.

The compiler binds dependent names when a template is instantiated. The compiler binds non-dependent names when a template is defined. For example:

```
void f(double) { cout << "Function f(double)" << endl; }

template<class T> void g(T a) {
    f(123);
    h(a);
}

void f(int) { cout << "Function f(int)" << endl; }
void h(double) { cout << "Function h(double)" << endl; }

void i() {
    extern void h(int);
    g<int>(234);
}

void h(int) { cout << "Function h(int)" << endl; }
```

The following is the output if you call function `i()`:

```
Function f(double)
Function h(double)
```

The *point of definition* of a template is located immediately before its definition. In this example, the point of definition of the function template `g(T)` is located immediately before the keyword `template`. Because the function call `f(123)` does not depend on a template argument, the compiler will consider names declared before the definition of function template `g(T)`. Therefore, the call `f(123)` will call `f(double)`. Although `f(int)` is a better match, it is not in scope at the point of definition of `g(T)`.

The *point of instantiation* of a template is located immediately before the declaration that encloses its use. In this example, the point of instantiation of the call to `g<int>(234)` is located immediately before `i()`. Because the function call `h(a)` depends on a template argument, the compiler will consider names declared before the instantiation of function template `g(T)`. Therefore, the call `h(a)` will call `h(double)`. It will not consider `h(int)`, because this function was not in scope at the point of instantiation of `g<int>(234)`.

Point of instantiation binding implies the following:

- A template parameter cannot depend on any local name or class member.
- An unqualified name in a template cannot depend on a local name or class member.

Related information

- “Template instantiation (C++ only)” on page 338

The typename keyword (C++ only)

Use the keyword `typename` if you have a qualified name that refers to a type and depends on a template parameter. Only use the keyword `typename` in template declarations and definitions. The following example illustrates the use of the keyword `typename`:

```
template<class T> class A
{
    T::x(y);
    typedef char C;
    A::C d;
}
```

The statement `T::x(y)` is ambiguous. It could be a call to function `x()` with a nonlocal argument `y`, or it could be a declaration of variable `y` with type `T::x`. C++ will interpret this statement as a function call. In order for the compiler to interpret this statement as a declaration, you would add the keyword `typename` to the beginning of it. The statement `A::C d;` is ill-formed. The class `A` also refers to `A<T>` and thus depends on a template parameter. You must add the keyword `typename` to the beginning of this declaration:

```
    typename A::C d;
```

You can also use the keyword `typename` in place of the keyword `class` in template parameter declarations.

Related information

- “Template parameters (C++ only)” on page 320

The template keyword as qualifier (C++ only)

Use the keyword `template` as a qualifier to distinguish member templates from other names. The following example illustrates when you must use `template` as a qualifier:

```
class A
{
    public:
        template<class T> T function_m() { };
};

template<class U> void function_n(U argument)
{
    char object_x = argument.function_m<char>();
}
```

The declaration `char object_x = argument.function_m<char>();` is ill-formed. The compiler assumes that the `<` is a less-than operator. In order for the compiler to recognize the function template call, you must add the `template` quantifier:

```
char object_x = argument.template function_m<char>();
```

If the name of a member template specialization appears after a `.`, `->`, or `::` operator, and that name has explicitly qualified template parameters, prefix the member template name with the keyword `template`. The following example demonstrates this use of the keyword `template`:

```
#include <iostream>
using namespace std;

class X {
    public:
        template <int j> struct S {
            void h() {
                cout << "member template's member function: " << j << endl;
            }
        };
        template <int i> void f() {
            cout << "Primary: " << i << endl;
        }
};

template<> void X::f<20>() {
    cout << "Specialized, non-type argument = 20" << endl;
}

template<class T> void g(T* p) {
    p->template f<100>();
    p->template f<20>();
    typename T::template S<40> s; // use of scope operator on a member template
    s.h();
}

int main()
{
    X temp;
    g(&temp);
}
```

The following is the output of the above example:

```
Primary: 100
Specialized, non-type argument = 20
member template's member function: 40
```

If you do not use the keyword `template` in these cases, the compiler will interpret the `<` as a less-than operator. For example, the following line of code is ill-formed:

```
p->f<100>();
```

The compiler interprets `f` as a non-template member, and the `<` as a less-than operator.

Chapter 16. Exception handling (C++ only)

Exception handling is a mechanism that separates code that detects and handles exceptional circumstances from the rest of your program. Note that an exceptional circumstance is not necessarily an error.

When a function detects an exceptional situation, you represent this with an object. This object is called an *exception object*. In order to deal with the exceptional situation you *throw the exception*. This passes control, as well as the exception, to a designated block of code in a direct or indirect caller of the function that threw the exception. This block of code is called a *handler*. In a handler, you specify the types of exceptions that it may process. The C++ run time, together with the generated code, will pass control to the first appropriate handler that is able to process the exception thrown. When this happens, an exception is *caught*. A handler may *rethrow* an exception so it can be caught by another handler.

The exception handling mechanism is made up of the following elements:

- try blocks
- catch blocks
- throw expressions
- Exception specifications (C++ only)

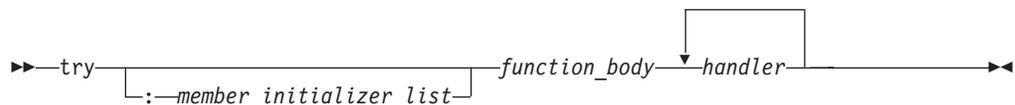
try blocks (C++ only)

You use a *try block* to indicate which areas in your program that might throw exceptions you want to handle immediately. You use a *function try block* to indicate that you want to detect exceptions in the entire body of a function.

try block syntax



Function try block syntax



The following is an example of a function try block with a member initializer, a function try block and a try block:

```
#include <iostream>
using namespace std;

class E {
public:
    const char* error;
    E(const char* arg) : error(arg) { }
};

class A {
public:
```

```

    int i;

    // A function try block with a member
    // initializer
    A() try : i(0) {
        throw E("Exception thrown in A()");
    }
    catch (E& e) {
        cout << e.error << endl;
    }
};

// A function try block
void f() try {
    throw E("Exception thrown in f()");
}
catch (E& e) {
    cout << e.error << endl;
}

void g() {
    throw E("Exception thrown in g()");
}

int main() {
    f();

    // A try block
    try {
        g();
    }
    catch (E& e) {
        cout << e.error << endl;
    }
    try {
        A x;
    }
    catch(...) { }
}

```

The following is the output of the above example:

```

Exception thrown in f()
Exception thrown in g()
Exception thrown in A()

```

The constructor of class A has a function try block with a member initializer. Function f() has a function try block. The main() function contains a try block.

Related information

- “Initialization of base classes and members (C++ only)” on page 304

Nested try blocks (C++ only)

When try blocks are nested and a throw occurs in a function called by an inner try block, control is transferred outward through the nested try blocks until the first catch block is found 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.
```

In the above example, if `spec_err` is thrown within the inner try block (in this case, from `func2()`), the exception is caught by the inner catch block, and, assuming this catch block does not transfer control, `func3()` is called. If `spec_err` is thrown after the inner try block (for instance, by `func3()`), it is not caught and the function `terminate()` is called. 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()`, because no appropriate catch block exists following the inner try block.

You can also nest a try block within a catch block.

catch blocks (C++ only)

catch block syntax

►►—catch—(—*exception_declaration*—)—{—*statements*—}—◄◄

You can declare a handler to catch many types of exceptions. The allowable objects that a function can catch are declared in the parentheses following the `catch` keyword (the *exception_declaration*). 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. The *exception_declaration* cannot be an incomplete type, or a reference or pointer to an incomplete type other than one of the following:

- `void*`
- `const void*`
- `volatile void*`
- `const volatile void*`

You cannot define a type in an *exception_declaration*.

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 any exception thrown can be handled by this handler.

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 caught must have an accessible copy constructor.

Related information

- “Type qualifiers” on page 67
- “Member access (C++ only)” on page 265

Function try block handlers (C++ only)

The scope and lifetime of the parameters of a function or constructor extend into the handlers of a function try block. The following example demonstrates this:

```
void f(int &x) try {
    throw 10;
}
catch (const int &i)
{
    x = i;
}

int main() {
    int v = 0;
    f(v);
}
```

The value of `v` after `f()` is called is 10.

A function try block on `main()` does not catch exceptions thrown in destructors of objects with static storage duration, or constructors of namespace scope objects.

The following example throws an exception from a destructor of a static object:

```
#include <iostream>
using namespace std;

class E {
public:
    const char* error;
    E(const char* arg) : error(arg) { }
};

class A {
public: ~A() { throw E("Exception in ~A()"); }
};

class B {
public: ~B() { throw E("Exception in ~B()"); }
};

int main() try {
    cout << "In main" << endl;
    static A cow;
    B bull;
}
catch (E& e) {
    cout << e.error << endl;
}
```

The following is the output of the above example:

```
In main
Exception in ~B()
```

The run time will not catch the exception thrown when object `cow` is destroyed at the end of the program.

The following example throws an exception from a constructor of a namespace scope object:

```
#include <iostream>
using namespace std;

class E {
public:
    const char* error;
    E(const char* arg) : error(arg) { }
};

namespace N {
    class C {
    public:
        C() {
            cout << "In C()" << endl;
            throw E("Exception in C()");
        }
    };

    C calf;
};

int main() try {
    cout << "In main" << endl;
}
catch (E& e) {
    cout << e.error << endl;
}
```

The following is the output of the above example:

```
In C()
```

The compiler will not catch the exception thrown when object `calf` is created.

In a function try block's handler, you cannot have a jump into the body of a constructor or destructor.

A return statement cannot appear in a function try block's handler of a constructor.

When the function try block's handler of an object's constructor or destructor is entered, fully constructed base classes and members of that object are destroyed. The following example demonstrates this:

```
#include <iostream>
using namespace std;

class E {
public:
    const char* error;
    E(const char* arg) : error(arg) { };
};

class B {
public:
    B() { };
    ~B() { cout << "~B() called" << endl; };
};

class D : public B {
public:
```

```

        D();
        ~D() { cout << "~D() called" << endl; };
};

D::~D() try : B() {
    throw E("Exception in D()");
}
catch(E& e) {
    cout << "Handler of function try block of D(): " << e.error << endl;
};

int main() {
    try {
        D val;
    }
    catch(...) { }
}

```

The following is the output of the above example:

```

~B() called
Handler of function try block of D(): Exception in D()

```

When the function try block's handler of D() is entered, the run time first calls the destructor of the base class of D, which is B. The destructor of D is not called because val is not fully constructed.

The run time will rethrow an exception at the end of a function try block's handler of a constructor or destructor. All other functions will return once they have reached the end of their function try block's handler. The following example demonstrates this:

```

#include <iostream>
using namespace std;

class E {
public:
    const char* error;
    E(const char* arg) : error(arg) { };
};

class A {
public:
    A() try { throw E("Exception in A()"); }
    catch(E& e) { cout << "Handler in A(): " << e.error << endl; }
};

int f() try {
    throw E("Exception in f()");
    return 0;
}
catch(E& e) {
    cout << "Handler in f(): " << e.error << endl;
    return 1;
}

int main() {
    int i = 0;
    try { A a; }
    catch(E& e) {
        cout << "Handler in main(): " << e.error << endl;
    }

    try { i = f(); }
    catch(E& e) {
        cout << "Another handler in main(): " << e.error << endl;
    }
}

```

```

    }
    cout << "Returned value of f(): " << i << endl;
}

```

The following is the output of the above example:

```

Handler in A(): Exception in A()
Handler in main(): Exception in A()
Handler in f(): Exception in f()
Returned value of f(): 1

```

Related information

- “The main() function” on page 205
- “The static storage class specifier” on page 44
- Chapter 9, “Namespaces (C++ only),” on page 217
- “Destructors (C++ only)” on page 308

Arguments of catch blocks (C++ only)

If you specify a class type for the argument of a catch block (the *exception_declaration*), the compiler uses a copy constructor to initialize that argument. If that argument does not have a name, the compiler initializes a temporary object and destroys it when the handler exits.

The ISO C++ specifications do not require the compiler to construct temporary objects in cases where they are redundant. The compiler takes advantage of this rule to create more efficient, optimized code. Take this into consideration when debugging your programs, especially for memory problems.

Matching between exceptions thrown and caught (C++ only)

An argument in the catch argument of a handler matches an argument in the *assignment_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 can be converted to the pointer type of the catch argument by standard 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, nonvolatile, or nonreference object type. A nonreference catch argument type matches a reference to an object of the same type.

Related information

- “Pointer conversions” on page 107
- “Type qualifiers” on page 67
- “References (C++ only)” on page 87

Order of catching (C++ only)

If the compiler encounters an exception in a try block, it will try each handler in order of appearance.

If a catch block for objects of a base class precedes a catch block for objects of a class derived from that base class, the compiler issues a warning and continues to compile the program despite the unreachable code in the derived class handler.

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.

If the run time cannot find a matching handler in the current scope, the run time will continue to find a matching handler in a dynamically surrounding try block. The following example demonstrates this:

```
#include <iostream>
using namespace std;

class E {
public:
    const char* error;
    E(const char* arg) : error(arg) { };
};

class F : public E {
public:
    F(const char* arg) : E(arg) { };
};

void f() {
    try {
        cout << "In try block of f()" << endl;
        throw E("Class E exception");
    }
    catch (F& e) {
        cout << "In handler of f()";
        cout << e.error << endl;
    }
};

int main() {
    try {
        cout << "In main" << endl;
        f();
    }
    catch (E& e) {
        cout << "In handler of main: ";
        cout << e.error << endl;
    };
    cout << "Resume execution in main" << endl;
}
```

The following is the output of the above example:

```
In main
In try block of f()
In handler of main: Class E exception
Resume execution in main
```

In function `f()`, the run time could not find a handler to handle the exception of type `E` thrown. The run time finds a matching handler in a dynamically surrounding try block: the try block in the `main()` function.

If the run time cannot find a matching handler in the program, it calls the `terminate()` function.

Related information

- “try blocks (C++ only)” on page 353

throw expressions (C++ only)

You use a *throw expression* to indicate that your program has encountered an exception.

throw expression syntax

```

▶▶ throw _____ ▶▶
      └──assignment_expression──┘
  
```

The type of *assignment_expression* cannot be an incomplete type, or a pointer to an incomplete type other than one of the following:

- void*
- const void*
- volatile void*
- const volatile void*

The *assignment_expression* is treated the same way as a function argument in a call or the operand of a return statement.

If the *assignment_expression* is a class object, the copy constructor and destructor of that object must be accessible. For example, you cannot throw a class object that has its copy constructor declared as private.

Related information

- “Incomplete types” on page 40

Rethrowing an exception (C++ only)

If a catch block cannot handle the particular exception it has caught, you can rethrow the exception. The rethrow expression (*throw without assignment_expression*) causes the originally thrown object to be rethrown.

Because the exception has already been caught at the scope in which the rethrow expression occurs, it is rethrown out to the next dynamically enclosing try block. Therefore, it cannot be handled by catch blocks at the scope in which the rethrow expression occurred. Any catch blocks for the dynamically enclosing try block have an opportunity to catch the exception.

The following example demonstrates rethrowing an exception:

```

#include <iostream>
using namespace std;

struct E {
    const char* message;
    E() : message("Class E") { }
};

struct E1 : E {
    const char* message;
    E1() : message("Class E1") { }
};

struct E2 : E {
    const char* message;
    E2() : message("Class E2") { }
}
  
```

```

};

void f() {
    try {
        cout << "In try block of f()" << endl;
        cout << "Throwing exception of type E1" << endl;
        E1 myException;
        throw myException;
    }
    catch (E2& e) {
        cout << "In handler of f(), catch (E2& e)" << endl;
        cout << "Exception: " << e.message << endl;
        throw;
    }
    catch (E1& e) {
        cout << "In handler of f(), catch (E1& e)" << endl;
        cout << "Exception: " << e.message << endl;
        throw;
    }
    catch (E& e) {
        cout << "In handler of f(), catch (E& e)" << endl;
        cout << "Exception: " << e.message << endl;
        throw;
    }
}

int main() {
    try {
        cout << "In try block of main()" << endl;
        f();
    }
    catch (E2& e) {
        cout << "In handler of main(), catch (E2& e)" << endl;
        cout << "Exception: " << e.message << endl;
    }
    catch (...) {
        cout << "In handler of main(), catch (...)" << endl;
    }
}

```

The following is the output of the above example:

```

In try block of main()
In try block of f()
Throwing exception of type E1
In handler of f(), catch (E1& e)
Exception: Class E1
In handler of main(), catch (...)

```

The try block in the main() function calls function f(). The try block in function f() throws an object of type E1 named myException. The handler catch (E1 &e) catches myException. The handler then rethrows myException with the statement throw to the next dynamically enclosing try block: the try block in the main() function. The handler catch(...) catches myException.

Stack unwinding (C++ only)

When an exception is thrown and control passes from a try block to a handler, the C++ run time calls destructors for all automatic objects constructed since the beginning of the try block. This process is called *stack unwinding*. The automatic objects are destroyed in reverse order of their construction. (Automatic objects are local objects that have been declared auto or register, or not declared static or extern. An automatic object x is deleted whenever the program exits the block in which x is declared.)

If an exception is thrown during construction of an object consisting of subobjects or array elements, destructors are only called for those subobjects or array elements successfully constructed before the exception was thrown. A destructor for a local static object will only be called if the object was successfully constructed.

If during stack unwinding a destructor throws an exception and that exception is not handled, the `terminate()` function is called. The following example demonstrates this:

```
#include <iostream>
using namespace std;

struct E {
    const char* message;
    E(const char* arg) : message(arg) { }
};

void my_terminate() {
    cout << "Call to my_terminate" << endl;
};

struct A {
    A() { cout << "In constructor of A" << endl; }
    ~A() {
        cout << "In destructor of A" << endl;
        throw E("Exception thrown in ~A()");
    }
};

struct B {
    B() { cout << "In constructor of B" << endl; }
    ~B() { cout << "In destructor of B" << endl; }
};

int main() {
    set_terminate(my_terminate);

    try {
        cout << "In try block" << endl;
        A a;
        B b;
        throw("Exception thrown in try block of main()");
    }
    catch (const char* e) {
        cout << "Exception: " << e << endl;
    }
    catch (...) {
        cout << "Some exception caught in main()" << endl;
    }

    cout << "Resume execution of main()" << endl;
}
```

The following is the output of the above example:

```
In try block
In constructor of A
In constructor of B
In destructor of B
In destructor of A
Call to my_terminate
```

In the try block, two automatic objects are created: `a` and `b`. The try block throws an exception of type `const char*`. The handler `catch (const char* e)` catches this exception. The C++ run time unwinds the stack, calling the destructors for `a` and `b`

in reverse order of their construction. The destructor for a throws an exception. Since there is no handler in the program that can handle this exception, the C++ run time calls `terminate()`. (The function `terminate()` calls the function specified as the argument to `set_terminate()`. In this example, `terminate()` has been specified to call `my_terminate()`.)

Exception specifications (C++ only)

C++ provides a mechanism to ensure that a given function is limited 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 throw only the exceptions contained in the exception specification.

For example, a function:

```
void translate() throw(unknown_word,bad_grammar) { /* ... */ }
```

explicitly states that it will only throw exception objects whose types are `unknown_word` or `bad_grammar`, or any type derived from `unknown_word` or `bad_grammar`.

Exception specification syntax

►► throw (type_id_list) ◀◀

The *type_id_list* is a comma-separated list of types. In this list you cannot specify an incomplete type, a pointer or a reference to an incomplete type, other than a pointer to `void`, optionally qualified with `const` and/or `volatile`. You cannot define a type in an exception specification.

A function with no exception specification allows all exceptions. A function with an exception specification that has an empty *type_id_list*, `throw()`, does not allow any exceptions to be thrown.

An exception specification is not part of a function's type.

An exception specification may only appear at the end of a function declarator of a function, pointer to function, reference to function, pointer to member function declaration, or pointer to member function definition. An exception specification cannot appear in a typedef declaration. The following declarations demonstrate this:

```
void f() throw(int);
void (*g)() throw(int);
void h(void i() throw(int));
// typedef int (*j)() throw(int); This is an error.
```

The compiler would not allow the last declaration, `typedef int (*j)() throw(int)`.

Suppose that class `A` is one of the types in the *type_id_list* of an exception specification of a function. That function may throw exception objects of class `A`, or any class publicly derived from class `A`. The following example demonstrates this:

```
class A { };
class B : public A { };
class C { };

void f(int i) throw (A) {
    switch (i) {
```

```

        case 0: throw A();
        case 1: throw B();
        default: throw C();
    }
}

void g(int i) throw (A*) {
    A* a = new A();
    B* b = new B();
    C* c = new C();
    switch (i) {
        case 0: throw a;
        case 1: throw b;
        default: throw c;
    }
}

```

Function `f()` can throw objects of types `A` or `B`. If the function tries to throw an object of type `C`, the compiler will call `unexpected()` because type `C` has not been specified in the function's exception specification, nor does it derive publicly from `A`. Similarly, function `g()` cannot throw pointers to objects of type `C`; the function may throw pointers of type `A` or pointers of objects that derive publicly from `A`.

A function that overrides a virtual function can only throw exceptions specified by the virtual function. The following example demonstrates this:

```

class A {
public:
    virtual void f() throw (int, char);
};

class B : public A{
public: void f() throw (int) { }
};

/* The following is not allowed. */
/*
class C : public A {
public: void f() { }
};

class D : public A {
public: void f() throw (int, char, double) { }
};
*/

```

The compiler allows `B::f()` because the member function may throw only exceptions of type `int`. The compiler would not allow `C::f()` because the member function may throw any kind of exception. The compiler would not allow `D::f()` because the member function can throw more types of exceptions (`int`, `char`, and `double`) than `A::f()`.

Suppose that you assign or initialize a pointer to function named `x` with a function or pointer to function named `y`. The pointer to function `x` can only throw exceptions specified by the exception specifications of `y`. The following example demonstrates this:

```

void (*f)();
void (*g)();
void (*h)() throw (int);

```

```

void i() {
    f = h;
    // h = g; This is an error.
}

```

The compiler allows the assignment `f = h` because `f` can throw any kind of exception. The compiler would not allow the assignment `h = g` because `h` can only throw objects of type `int`, while `g` can throw any kind of exception.

Implicitly declared special member functions (default constructors, copy constructors, destructors, and copy assignment operators) have exception specifications. An implicitly declared special member function will have in its exception specification the types declared in the functions' exception specifications that the special function invokes. If any function that a special function invokes allows all exceptions, then that special function allows all exceptions. If all the functions that a special function invokes allow no exceptions, then that special function will allow no exceptions. The following example demonstrates this:

```

class A {
public:
    A() throw (int);
    A(const A&) throw (float);
    ~A() throw();
};

class B {
public:
    B() throw (char);
    B(const A&);
    ~B() throw();
};

class C : public B, public A { };

```

The following special functions in the above example have been implicitly declared:

```

C::C() throw (int, char);
C::C(const C&); // Can throw any type of exception, including float
C::~~C() throw();

```

The default constructor of `C` can throw exceptions of type `int` or `char`. The copy constructor of `C` can throw any kind of exception. The destructor of `C` cannot throw any exceptions.

Related information

- “Incomplete types” on page 40
- “Function declarations and definitions” on page 183
- “Pointers to functions” on page 214
- Chapter 14, “Special member functions (C++ only),” on page 299

Special exception handling functions (C++ only)

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:

- The `unexpected()` function (C++ only)
- The `terminate()` function (C++ only)

The unexpected() function (C++ only)

When a function with an exception specification throws an exception that is not listed in its exception specification, the C++ run time does the following:

1. The `unexpected()` function is called.
2. The `unexpected()` function calls the function pointed to by `unexpected_handler`. By default, `unexpected_handler` points to the function `terminate()`.

You can replace the default value of `unexpected_handler` with the function `set_unexpected()`.

Although `unexpected()` cannot return, it may throw (or rethrow) an exception. Suppose the exception specification of a function `f()` has been violated. If `unexpected()` throws an exception allowed by the exception specification of `f()`, then the C++ run time will search for another handler at the call of `f()`. The following example demonstrates this:

```
#include <iostream>
using namespace std;

struct E {
    const char* message;
    E(const char* arg) : message(arg) { }
};

void my_unexpected() {
    cout << "Call to my_unexpected" << endl;
    throw E("Exception thrown from my_unexpected");
}

void f() throw(E) {
    cout << "In function f(), throw const char* object" << endl;
    throw("Exception, type const char*, thrown from f()");
}

int main() {
    set_unexpected(my_unexpected);
    try {
        f();
    }
    catch (E& e) {
        cout << "Exception in main(): " << e.message << endl;
    }
}
```

The following is the output of the above example:

```
In function f(), throw const char* object
Call to my_unexpected
Exception in main(): Exception thrown from my_unexpected
```

The `main()` function's try block calls function `f()`. Function `f()` throws an object of type `const char*`. However the exception specification of `f()` allows only objects of type `E` to be thrown. The function `unexpected()` is called. The function `unexpected()` calls `my_unexpected()`. The function `my_unexpected()` throws an object of type `E`. Since `unexpected()` throws an object allowed by the exception specification of `f()`, the handler in the `main()` function may handle the exception.

If `unexpected()` did not throw (or rethrow) an object allowed by the exception specification of `f()`, then the C++ run time does one of two things:

- If the exception specification of `f()` included the class `std::bad_exception`, `unexpected()` will throw an object of type `std::bad_exception`, and the C++ run time will search for another handler at the call of `f()`.
- If the exception specification of `f()` did not include the class `std::bad_exception`, the function `terminate()` is called.

Related information

- “Special exception handling functions (C++ only)” on page 366
- “The `set_unexpected()` and `set_terminate()` functions (C++ only)” on page 369

The `terminate()` function (C++ only)

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:

- The exception handling mechanism cannot find a handler for a thrown exception. The following are more specific cases of this:
 - During stack unwinding, a destructor throws an exception and that exception is not handled.
 - The expression that is thrown also throws an exception, and that exception is not handled.
 - The constructor or destructor of a nonlocal static object throws an exception, and the exception is not handled.
 - A function registered with `atexit()` throws an exception, and the exception is not handled. The following demonstrates this:

```
extern "C" printf(char* ...);
#include <exception>
#include <cstdlib>
using namespace std;

extern "C" void f() {
    printf("Function f()\n");
    throw "Exception thrown from f()";
}

extern "C" void g() { printf("Function g()\n"); }
extern "C" void h() { printf("Function h()\n"); }

void my_terminate() {
    printf("Call to my_terminate\n");
    abort();
}

int main() {
    set_terminate(my_terminate);
    atexit(f);
    atexit(g);
    atexit(h);
    printf("In main\n");
}
```

The following is the output of the above example:

```
In main
Function h()
Function g()
Function f()
Call to my_terminate
```

To register a function with `atexit()`, you pass a parameter to `atexit()` a pointer to the function you want to register. At normal program termination,

`atexit()` calls the functions you have registered with no arguments in reverse order. The `atexit()` function is in the `<cstdlib>` library.

- A throw expression without an operand tries to rethrow an exception, and no exception is presently being handled.
- A function `f()` throws an exception that violates its exception specification. The `unexpected()` function then throws an exception which violates the exception specification of `f()`, and the exception specification of `f()` did not include the class `std::bad_exception`.
- The default value of `unexpected_handler` is called.

The `terminate()` function calls the function pointed to by `terminate_handler`. By default, `terminate_handler` points to the function `abort()`, which exits from the program. You can replace the default value of `terminate_handler` with the function `set_terminate()`.

A terminate function cannot return to its caller, either by using `return` or by throwing an exception.

Related information

- “The `set_unexpected()` and `set_terminate()` functions (C++ only)”

The `set_unexpected()` and `set_terminate()` functions (C++ only)

The function `unexpected()`, when invoked, calls the function most recently supplied as an argument to `set_unexpected()`. If `set_unexpected()` has not yet been called, `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()`. The functions `set_unexpected()` and `set_terminate()` are included in the standard header files. 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 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 function should not return to its caller but terminate execution of the program.

Example using the exception handling functions (C++ only)

The following example shows the flow of control and special functions used in exception handling:

```
#include <iostream>
#include <exception>
using namespace std;
```

```

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;
    abort();
}

void my_unexpected() {
    cout << "Call to my_unexpected()" << endl;
    throw;
}

void f() throw(X,Y, bad_exception) {
    throw A();
}

void g() throw(X,Y) {
    throw A();
}

int main()
{
    pfv old_term = set_terminate(my_terminate);
    pfv old_unex = set_unexpected(my_unexpected);
    try {
        cout << "In first try block" << endl;
        f();
    }
    catch(X) {
        cout << "Caught X" << endl;
    }
    catch(Y) {
        cout << "Caught Y" << endl;
    }
    catch (bad_exception& e1) {
        cout << "Caught bad_exception" << endl;
    }
    catch (...) {
        cout << "Caught some exception" << endl;
    }

    cout << endl;

    try {
        cout << "In second try block" << endl;
        g();
    }
    catch(X) {
        cout << "Caught X" << endl;
    }
    catch(Y) {
        cout << "Caught Y" << endl;
    }
    catch (bad_exception& e2) {
        cout << "Caught bad_exception" << endl;
    }
    catch (...) {
        cout << "Caught some exception" << endl;
    }
}

```

The following is the output of the above example:

```
In first try block  
Call to my_unexpected()  
Caught bad_exception
```

```
In second try block  
Call to my_unexpected()  
Call to my_terminate
```

At run time, this program behaves as follows:

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 the first 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. The function `my_unexpected()` tries to rethrow the exception of type `A`. Because class `A` has not been specified in the exception specification of function `f()`, `my_unexpected()` throws an exception of type `bad_exception`.
4. Because `bad_exception` has been specified in the exception specification of function `f()`, the handler `catch (bad_exception& e1)` is able to handle the exception.
5. Within the second try block, function `g()` is called. Because `g()` throws an `unexpected` exception, a call to `unexpected()` is made. The `unexpected()` throws an exception of type `bad_exception`. Because `bad_exception` has not been specified in the exception specification of `g()`, `unexpected()` calls `terminate()`, which calls the function `my_terminate()`.
6. `my_terminate()` displays a message then calls `abort()`, which terminates the program.

Note that the catch blocks following the second try block are not entered, because the exception was handled by `my_unexpected()` as an `unexpected` throw, not as a valid exception.

Chapter 17. Preprocessor directives

The preprocessor is a program that is invoked by the compiler to process code before compilation. Commands for that program, known as *directives*, are lines of the source file beginning with the character #, which distinguishes them from lines of source program text. The effect of each preprocessor directive is a change to the text of the source code, and the result is a new source code file, which does not contain the directives. The preprocessed source code, an intermediate file, must be a valid C or C++ program, because it becomes the input to the compiler.

Preprocessor directives consist of the following:

- Macro definition directives, which replace tokens in the current file with specified replacement tokens
- File inclusion directives, which imbed files within the current file
- Conditional compilation directives, which conditionally compile sections of the current file
- Message generation directives, which control the generation of diagnostic messages
- The null directive (#), which performs no action
- Pragma directives, which apply compiler-specific rules to specified sections of code

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

A preprocessor directive ends at the new-line character unless the last character of the line is the \ (backslash) character. If the \ character appears as the last character in the preprocessor line, the preprocessor interprets the \ and the new-line character as a continuation marker. The preprocessor deletes the \ (and the following new-line character) and splices the physical source lines into continuous logical lines. White space is allowed between backslash and the end of line character or the physical end of record. However, this white space is usually not visible during editing.

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

Macro definition directives

Macro definition directives include the following directives and operators:

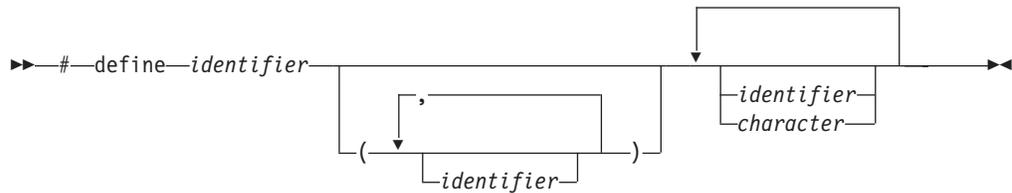
- The #define directive, which defines a macro
- The #undef directive, which removes a macro definition

“Standard predefined macro names” on page 380 describes the macros that are predefined by the ISO C standard.  Chapter 19, “Compiler predefined macros,” on page 455 describes the macros that are predefined for z/OS XL C/C++.

The #define directive

A *preprocessor define directive* directs the preprocessor to replace all subsequent occurrences of a macro with specified replacement tokens.

#define directive syntax



The #define directive can contain:

- Object-like macros
- Function-like macros

The following are some differences between #define and the const type qualifier:

- The #define directive can be used to create a name for a numerical, character, or string constant, whereas a const object of any type can be declared.
- A const object is subject to the scoping rules for variables, whereas a constant created using #define is not.
- Unlike a const object, the value of a macro does not appear in the intermediate source code used by the compiler because they are expanded inline. The inline expansion makes the macro value unavailable to the debugger.
- A macro can be used in a constant expression, such as an array bound, whereas a const object cannot.
-  The compiler does not type-check a macro, including macro arguments.

Related information

- “The const type qualifier” on page 69

Object-like macros

An *object-like macro definition* replaces a single identifier with the specified replacement tokens. The following object-like definition causes the preprocessor to replace all subsequent instances of the identifier COUNT with the constant 1000 :

```
#define COUNT 1000
```

If the statement

```
int arry[COUNT];
```

appears after this definition and in the same file as the definition, the preprocessor would change the statement to

```
int arry[1000];
```

in the output of the preprocessor.

Other definitions can make reference to the identifier COUNT:

```
#define MAX_COUNT COUNT + 100
```

The preprocessor replaces each subsequent occurrence of MAX_COUNT with COUNT + 100, which the preprocessor then replaces with 1000 + 100.

If a number that is partially built by a macro expansion is produced, the preprocessor does not consider the result to be a single value. For example, the following will not result in the value 10.2 but in a syntax error.

```
#define a 10
a.2
```

Identifiers that are partially built from a macro expansion may not be produced. Therefore, the following example contains two identifiers and results in a syntax error:

```
#define d efg
abcd
```

Function-like macros

More complex than object-like macros, a function-like macro definition declares the names of formal parameters within parentheses, separated by commas. An empty formal parameter list is legal: such a macro can be used to simulate a function that takes no arguments.  C99 adds support for function-like macros with a variable number of arguments.

Function-like macro definition:

An identifier followed by a parameter list in parentheses and the replacement tokens. The parameters are imbedded in the replacement code. White space cannot separate the identifier (which is the name of the macro) and the left parenthesis of the parameter list. A comma must separate each parameter.

For portability, you should not have more than 31 parameters for a macro. The parameter list may end with an ellipsis (...). In this case, the identifier `__VA_ARGS__` may appear in the replacement list.

Function-like macro invocation:

An identifier followed by a comma-separated list of arguments in parentheses. The number of arguments should match the number of parameters in the macro definition, unless the parameter list in the definition ends with an ellipsis. In this latter case, the number of arguments in the invocation should exceed the number of parameters in the definition. The excess are called *trailing arguments*. Once the preprocessor identifies a function-like macro invocation, argument substitution takes place. A parameter in the replacement code is replaced by the corresponding argument. If trailing arguments are permitted by the macro definition, they are merged with the intervening commas to replace the identifier `__VA_ARGS__`, as if they were a single argument. Any macro invocations contained in the argument itself are completely replaced before the argument replaces its corresponding parameter in the replacement code.

C only

A macro argument can be empty (consisting of zero preprocessing tokens). For example,

```
#define SUM(a,b,c) a + b + c
SUM(1,,3) /* No error message.
           1 is substituted for a, 3 is substituted for c. */
```

End of C only

If the identifier list does not end with an ellipsis, the number of arguments in a macro invocation must be the same as the number of parameters in the corresponding macro definition. During parameter substitution, any arguments remaining after all specified arguments have been substituted (including any separating commas) are combined into one argument called the variable argument.

The variable argument will replace any occurrence of the identifier `__VA_ARGS__` in the replacement list. The following example illustrates this:

```
#define debug(...)    fprintf(stderr, __VA_ARGS__)

debug("flag");      /*    Becomes fprintf(stderr, "flag");    */
```

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

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:

```
c = (x + y);
c = d * (x + y);
```

Use parentheses to ensure correct evaluation of replacement text. For example, the definition:

```
#define SQR(c) ((c) * (c))
```

requires parentheses around each parameter `c` in the definition in order to correctly evaluate an expression like:

```
y = SQR(a + b);
```

The preprocessor expands this statement to:

```
y = ((a + b) * (a + b));
```

Without parentheses in the definition, the correct order of evaluation is not preserved, and the preprocessor output is:

```
y = (a + b * a + b);
```

Arguments of the `#` and `##` operators are converted *before* replacement of parameters in a function-like macro.

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 does not end until a corresponding `#undef` directive is encountered. If there is no corresponding `#undef` directive, the scope of the macro definition lasts until the end of the translation unit.

A recursive macro is not fully expanded. For example, the definition

```
#define x(a,b) x(a+1,b+1) + 4
```

expands

```
x(20,10)
```

```
to
    x(20+1,10+1) + 4
```

rather than trying to expand the macro `x` over and over within itself. After the macro `x` is expanded, it is a call to function `x()`.

A definition is not required to specify replacement tokens. The following definition removes all instances of the token `debug` from subsequent lines in the current file:

```
#define debug
```

You can change the definition of a defined identifier or macro with a second preprocessor `#define` directive only if the second preprocessor `#define` directive is preceded by a preprocessor `#undef` directive. The `#undef` directive nullifies the first definition so that the same identifier can be used in a redefinition.

Within the text of the program, the preprocessor does not scan character constants or string constants for macro invocations.

The following example program contains two macro definitions and a macro invocation that refers to both of the defined macros:

CCNRAA8

```
/**
 ** This example illustrates #define directives.
 **/

#include <stdio.h>

#define SQR(s) ((s) * (s))
#define PRNT(a,b) \
    printf("value 1 = %d\n", a); \
    printf("value 2 = %d\n", b) ;

int main(void)
{
    int x = 2;
    int y = 3;

    PRNT(SQR(x),y);

    return(0);
}
```

After being interpreted by the preprocessor, this program is replaced by code equivalent to the following:

CCNRAA9

```
#include <stdio.h>

int main(void)
{
    int x = 2;
    int y = 3;

    printf("value 1 = %d\n", ( (x) * (x) ) );
    printf("value 2 = %d\n", y);

    return(0);
}
```

This program produces the following output:

```
value 1 = 4
value 2 = 3
```

Related information

- “Operator precedence and associativity” on page 156
- “Parenthesized expressions ()” on page 115

The #undef directive

A *preprocessor undef directive* causes the preprocessor to end the scope of a preprocessor definition.

#undef directive syntax

```
▶▶ #undef identifier ◀◀
```

If the identifier is not currently defined as a macro, #undef is ignored.

The following directives define BUFFER and SQR:

```
#define BUFFER 512
#define SQR(x) ((x) * (x))
```

The following directives nullify these definitions:

```
#undef BUFFER
#undef SQR
```

Any occurrences of the identifiers BUFFER and SQR that follow these #undef directives are not replaced with any replacement tokens. Once the definition of a macro has been removed by an #undef directive, the identifier can be used in a new #define directive.

The # operator

The # (single number sign) operator converts a parameter of a function-like macro into a character string literal. For example, if macro ABC is defined using the following directive:

```
#define ABC(x) #x
```

all subsequent invocations of the macro ABC would be expanded into a character string literal containing the argument passed to ABC. For example:

Invocation	Result of macro expansion
------------	---------------------------

ABC(1)	"1"
ABC>Hello there)	"Hello there"

The # operator should not be confused with the null directive.

Use the # operator in a function-like macro definition according to the following rules:

- A parameter following # operator in a function-like macro is converted into a character string literal containing the argument passed to the macro.
- White-space characters that appear before or after the argument passed to the macro are deleted.

- Multiple white-space characters imbedded within the argument passed to the macro are replaced by a single space character.
- If the argument passed to the macro contains a string literal and if a \ (backslash) character appears within the literal, a second \ character is inserted before the original \ when the macro is expanded.
- If the argument passed to the macro contains a " (double quotation mark) character, a \ character is inserted before the " when the macro is expanded.
- The conversion of an argument into a string literal occurs before macro expansion on that argument.
- If more than one ## operator or # operator appears in the replacement list of a macro definition, the order of evaluation of the operators is not defined.
- If the result of the macro expansion is not a valid character string literal, the behavior is undefined.

The following 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

- "The null directive (#)" on page 390

The ## operator

The ## (double number sign) operator concatenates two tokens in a macro invocation (text and/or arguments) given in a macro definition.

If a macro XY was defined using the following directive:

```
#define XY(x,y)    x##y
```

the last token of the argument for x is concatenated with the first token of the argument for y.

Use the ## operator according to the following rules:

- The ## operator cannot be the very first or very last item in the replacement list of a macro definition.
- The last token of the item in front of the ## operator is concatenated with first token of the item following the ## operator.
- Concatenation takes place before any macros in arguments are expanded.
- If the result of a concatenation is a valid macro name, it is available for further replacement even if it appears in a context in which it would not normally be available.
- If more than one ## operator and/or # operator appears in the replacement list of a macro definition, the order of evaluation of the operators is not defined.

The following examples demonstrate the use of the ## operator:

```

#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

- “The #define directive” on page 373

Standard predefined macro names

Both C and C++ provide the following predefined macro names as specified in the ISO C language standard. Except for `__FILE__` and `__LINE__`, the value of the predefined macros remains constant throughout the translation unit.

`__DATE__` A character string literal containing the date when the source file was compiled.

The value of `__DATE__` changes as the compiler processes any include files that are part of your source program. The date is in the form:

```
"Mmm dd yyyy"
```

where:

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.

yyyy Represents the year.

`__FILE__` A character string literal containing the name of the source file.

The value of `__FILE__` changes as the compiler processes include files that are part of your source program. It can be set with the `#line` directive.

`__LINE__` An integer representing the current source line number.

The value of `__LINE__` changes during compilation as the compiler processes subsequent lines of your source program. It can be set with the `#line` directive.

`__STDC__` For C, the integer 1 (one) indicates that the C compiler supports the ISO standard. If you set the language level to `COMMONC`, this macro is undefined. (When a macro is undefined, it behaves as if it had the integer value 0 when used in a `#if` statement.)

For C++, this macro is predefined to have the value 0 (zero). This indicates that the C++ language is not a proper superset of C, and

that the compiler does not conform to ISO C.

C only

__STDC_HOSTED__

The value of this C99 macro is 1, indicating that the C compiler is a hosted implementation. Note that this macro is only defined if `__STDC__` is also defined.

End of C only

C only

__STDC_VERSION__

The integer constant of type `long int`: 199409L for the C89 language level, 199901L for C99. Note that this macro is only defined if `__STDC__` is also defined.

End of C only

__TIME__

A character string literal containing 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.

C++ only

__cplusplus

For C++ programs, this macro expands to the long integer literal 199711L, indicating that the compiler is a C++ compiler. For C programs, this macro is not defined. Note that this macro name has no trailing underscores.

End of C++ only

Related information

- “The `#line` directive” on page 389
- “Object-like macros” on page 374

File inclusion directives

File inclusion directives consist of:

- The `#include` directive, which inserts text from another source file
-  The `#include_next` directive, which causes the compiler to omit the directory of the including file from the search path when searching for include files


```
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 that represents the name of the C standard I/O header file. A `#include` makes the header file available to the program.

```
#define C_IO_HEADER <stdio.h>

/* The following is equivalent to:
 * #include <stdio.h>
 */

#include C_IO_HEADER
```

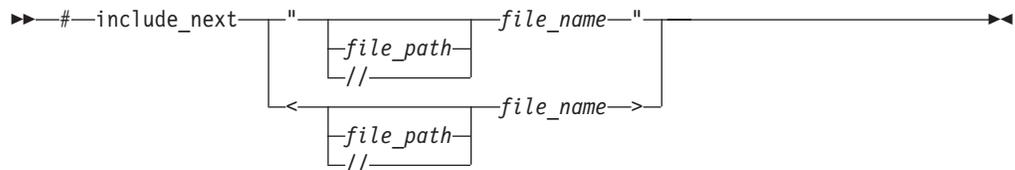
z/OS The z/OS implementation has specially defined behavior and compiler options for include file search paths, which are documented in greater detail in the descriptions of the `SEARCH` and `LSEARCH` options in the *z/OS XL C/C++ User's Guide*.

The `#include_next` directive

IBM extension

The preprocessor directive `#include_next` behaves like the `#include` directive, except that it specifically excludes the directory of the including file from the paths to be searched for the named file. All search paths up to and including the directory of the including file are omitted from the list of paths to be searched for the included file. This allows you to include multiple versions of a file with the same name in different parts of an application; or to include one header file in another header file with the same name (without the header including itself recursively). Provided that the different file versions are stored in different directories, the directive ensures you can access each version of the file, without requiring that you use absolute paths to specify the file name.

`#include_next` directive syntax



The directive must only be used in header files, and the file specified by the `file_name` must be a header file. There is no distinction between the use of double quotation marks and angle brackets to enclose the file name. **z/OS** You can specify a data set for `file_name`. Use double slashes (`//`) before the `file_name` to indicate that the file is a data set.

As an example of how search paths are resolved with the `#include_next` directive, assume that there are two versions of the file `t.h`: the first one, which is included in the source file `t.c`, is located in the subdirectory `path1`; the second one, which is

included in the first one, is located in the subdirectory path2. Both directories are specified as include file search paths when t.c is compiled.

```
/* t.c */

#include "t.h"

int main()
{
    printf("%d", ret_val);
}

/* t.h in path1 */

#include_next "t.h"

int ret_val = RET;

/* t.h in path2 */

#define RET 55;
```

The `#include_next` directive instructs the preprocessor to skip the path1 directory and start the search for the included file from the path2 directory. This directive allows you to use two different versions of t.h and it prevents t.h from being included recursively.

End of IBM extension

Conditional compilation directives

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

- The `#if` and `#elif` directives, which conditionally include or suppress portions of source code, depending on the result of a constant expression
- The `#ifdef` directive, which conditionally includes source text if a macro name is defined
- The `#ifndef` directive, which conditionally includes source text if a macro name is not defined
- The `#else` directive, which conditionally includes source text if the previous `#if`, `#ifdef`, `#ifndef`, or `#elif` test fails
- The `#endif` directive, which ends conditional text

The preprocessor conditional compilation directive spans several lines:

- The condition specification line (beginning with `#if`, `#ifdef`, or `#ifndef`)
- Lines containing code that the preprocessor passes on to the compiler if the condition evaluates to a nonzero value (optional)
- The `#elif` line (optional)
- 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. All the matching directives are considered to be at the same nesting level.

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

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

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

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

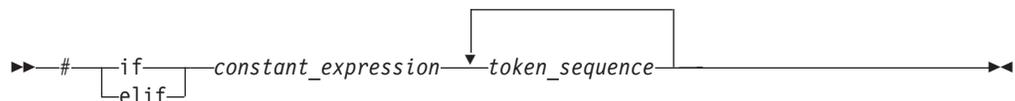
When a block following a preprocessor directive is to be ignored, the tokens are examined only to identify preprocessor directives within that block so that the conditional nesting level can be determined. All tokens other than the name of the directive are ignored.

Only the first block whose expression is nonzero is processed. The remaining blocks at that nesting level are ignored. If none of the blocks at that nesting level has been processed and there is a `#else` directive, the block following the `#else` directive is processed. If none of the blocks at that nesting level has been processed and there is no `#else` directive, the entire nesting level is ignored.

The `#if` and `#elif` directives

The `#if` and `#elif` directives compare the value of *constant_expression* to zero:

`#if` and `#elif` directive syntax



If the constant expression evaluates to a nonzero value, the lines of code that immediately follow the condition are passed on to the compiler.

If the expression evaluates to zero and the conditional compilation directive contains a preprocessor `#elif` directive, the source text located between the `#elif` and the next `#elif` or preprocessor `#else` directive is selected by the preprocessor to be passed on to the compiler. The `#elif` directive cannot appear after the preprocessor `#else` directive.

All macros are expanded, any `defined()` expressions are processed and all remaining identifiers are replaced with the token `0`.

The *constant_expression* that is tested must be integer constant expressions with the following properties:

- No casts are performed.
- Arithmetic is performed using long int values.
- The *constant_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 or #elif. The following expressions evaluate to 1 if the *identifier* is defined in the preprocessor, otherwise to 0:

```
defined identifier  
defined(identifier)
```

For example:

```
#if defined(TEST1) || defined(TEST2)
```

Note: If a macro is not defined, a value of 0 (zero) is assigned to it. In the following example, TEST must be a macro identifier:

```
#if TEST >= 1  
    printf("i = %d\n", i);  
    printf("array[i] = %d\n", array[i]);  
#elif TEST < 0  
    printf("array subscript out of bounds \n");  
#endif
```

The #ifdef directive

The #ifdef directive checks for the existence of macro definitions.

If the identifier specified is defined as a macro, the lines of code that immediately follow the condition are passed on to the compiler.

#ifdef directive syntax

The diagram illustrates the syntax of the #ifdef directive. It shows a sequence of tokens: #, ifdef, identifier, token_sequence, and newline_character. A horizontal line with arrows at both ends represents the entire line. A vertical line drops from the end of the identifier token to the start of the token_sequence token, with a horizontal line connecting them, indicating that the token_sequence is enclosed within the #ifdef block.

```
▶▶ # ifdef identifier token_sequence newline_character ▶▶
```

The following example defines MAX_LEN to be 75 if EXTENDED is defined for the preprocessor. Otherwise, MAX_LEN is defined to be 50.

```
#ifdef EXTENDED  
#   define MAX_LEN 75  
#else  
#   define MAX_LEN 50  
#endif
```

The #ifndef directive

The #ifndef directive checks whether a macro is not defined.

If the identifier specified is not defined as a macro, the lines of code immediately follow the condition are passed on to the compiler.

#ifndef directive syntax



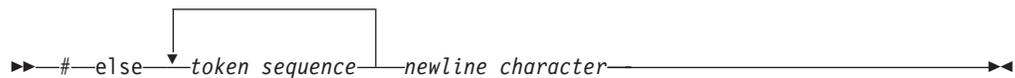
An identifier must follow the #ifndef keyword. The following example defines MAX_LEN to be 50 if EXTENDED is not defined for the preprocessor. Otherwise, MAX_LEN is defined to be 75.

```
#ifndef EXTENDED
#   define MAX_LEN 50
#else
#   define MAX_LEN 75
#endif
```

The #else directive

If the condition specified in the #if, #ifdef, or #ifndef directive evaluates to 0, and the conditional compilation directive contains a preprocessor #else directive, the lines of code located between the preprocessor #else directive and the preprocessor #endif directive is selected by the preprocessor to be passed on to the compiler.

#else directive syntax



The #endif directive

The preprocessor #endif directive ends the conditional compilation directive.

#endif directive syntax



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:

CCNRABC

```
/**
 ** 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 >= 1
        printf("i = %d\n", i);
        printf("array[i] = %d\n",
            array[i]);
#endif
    }
    return(0);
}
```

Message generation directives

Message generation directives include the following:

- The `#error` directive, which defines text for a compile-time error message
- The `#line` directive, which supplies a line number for compiler messages

Related information

- “Conditional compilation directives” on page 384

The `#error` directive

A *preprocessor error directive* causes the preprocessor to generate an error message and causes the compilation to fail.

`#error` directive syntax

```
▶▶ # error preprocessor_token ▶▶
```

The `#error` directive is often used in the `#else` portion of a `#if-#elif-#else` construct, as a safety check during compilation. For example, `#error` directives in the source file can prevent code generation if a section of the program is reached that should be bypassed.

For example, the directive

```
#define BUFFER_SIZE 255

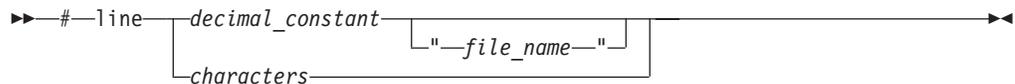
#ifdef BUFFER_SIZE < 256
#error "BUFFER_SIZE is too small."
#endif
```

generates the error message:
BUFFER_SIZE is too small.

The #line directive

A *preprocessor line control directive* supplies line numbers for compiler messages. It causes the compiler to view the line number of the next source line as the specified number.

#line directive syntax



In order for the compiler to produce meaningful references to line numbers in preprocessed source, the preprocessor inserts #line directives where necessary (for example, at the beginning and after the end of included text).

A file name specification enclosed in double quotation marks can follow the line number. If you specify a file name, the compiler views the next line as part of the specified file. If you do not specify a file name, the compiler views the next line as part of the current source file.

At the C99 language level, the maximum value of the #line preprocessing directive is 2147483647.

For z/OS XL C and C++ compilers, the *file_name* should be:

- A fully qualified sequential data set
- A fully qualified PDS or PDSE member
- A z/OS UNIX System Services path name

The entire string is taken unchanged as the alternate source file name for the translation 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 *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.

In all C and C++ implementations, the token sequence on a #line directive is subject to macro replacement. After macro replacement, the resulting character sequence must consist of a decimal constant, optionally followed by a file name enclosed in double quotation marks.

You can use #line control directives to make the compiler provide more meaningful error messages. The following example program uses #line control directives to give each function an easily recognizable line number:

CCNRABD

```
/**  
 ** 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

```

The null directive (#)

The *null directive* performs no action. It consists of a single # on a line of its own.

The null directive should not be confused with the # operator or the character that starts a preprocessor directive.

In the following example, if MINVAL is a defined macro name, no action is performed. If MINVAL is not a defined identifier, it is defined 1.

```

#ifdef MINVAL
#
#else
#define MINVAL 1
#endif

```

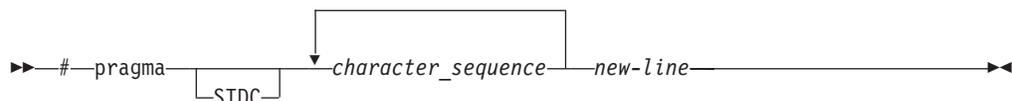
Related information

- “The # operator” on page 378

Pragma directives

A *pragma* is an implementation-defined instruction to the compiler. It has the general form:

#pragma directive syntax



The *character_sequence* is a series of characters giving a specific compiler instruction and arguments, if any. The token STDC indicates a standard pragma; consequently, no macro substitution takes place on the directive. The *new-line* character must terminate a pragma directive.

The *character_sequence* on a pragma is not subject to macro substitutions.

Note:  You can also use the `_Pragma` operator syntax to specify a pragma directive; for details, see “The `_Pragma` preprocessing operator (C only).”

More than one pragma construct can be specified on a single pragma directive. The compiler ignores unrecognized pragmas.

Standard C pragmas are described in “Standard pragmas (C only).”  Pragmas available for z/OS XL C/C++ are described in Chapter 18, “z/OS XL C/C++ pragmas,” on page 393.

The `_Pragma` preprocessing operator (C only)

The unary operator `_Pragma`, which is a C99 feature, allows a preprocessor macro to be contained in a pragma directive.

`_Pragma` operator syntax

►► `_Pragma`—(—“—*string_literal*—”—)—————►◄

The *string_literal* may be prefixed with `L`, making it a wide-string literal.

The string literal is destringized and tokenized. The resulting sequence of tokens is processed as if it appeared in a pragma directive. For example:

```
_Pragma ( "pack(full)" )
```

would be equivalent to

```
#pragma pack(full)
```

Standard pragmas (C only)

A *standard pragma* is a pragma preprocessor directive for which the C Standard defines the syntax and semantics and for which no macro replacement is performed. A standard pragma must be one of the following:

►► `#pragma`—STDC—

FP_CONTRACT	ON
FENV_ACCESS	OFF
CX_LIMITED_RANGE	DEFAULT

—*new-line*—————►◄

These pragmas are recognized and ignored.

Chapter 18. z/OS XL C/C++ pragmas

The following sections describe the pragmas available in z/OS XL C/C++:

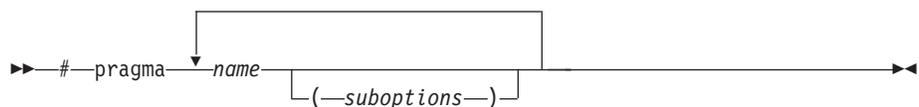
- “Pragma directive syntax”
- “Scope of pragma directives”
- “IPA considerations” on page 394
- “Summary of compiler pragmas by functional category” on page 394
- “Individual pragma descriptions” on page 398

Pragma directive syntax

z/OS XL C/C++ supports the following pragma directive:

#pragma *name*

This form uses the following syntax:



The *name* is the pragma directive name, and the *suboptions* are any required or optional suboptions that can be specified for the pragma, where applicable.

You can specify more than one *name* and *suboptions* in a single **#pragma** statement.

The compiler ignores unrecognized pragmas, issuing an informational message indicating this.

If you have any pragmas that are not common to both C and C++ in code that will be compiled by both compilers, you may add conditional compilation directives around the pragmas. (This is not strictly necessary since unrecognized pragmas are ignored.) For example, **#pragma object_model** is only recognized by the C++ compiler, so you may decide to add conditional compilation directives around the pragma.

```
#ifdef __cplusplus
#pragma object_model(pop)
#endif
```

Scope of pragma directives

Many pragma directives can be specified at any point within the source code in a compilation unit; others must be specified before any other directives or source code statements. In the individual descriptions for each pragma, the "Usage" section describes any constraints on the pragma's placement.

In general, if you specify a pragma directive before any code in your source program, it applies to the entire compilation unit, including any header files that are included. For a directive that can appear anywhere in your source code, it applies from the point at which it is specified, until the end of the compilation unit.

You can further restrict the scope of a pragma's application by using complementary pairs of pragma directives around a selected section of code. For example, using **#pragma checkout (suspend)** and **#pragma checkout (resume)** directives as follows requests that the selected parts of your source code be excluded from being diagnosed by the CHECKOUT compiler option:

```
#pragma checkout (suspend)

/*Source code between the suspend and resume pragma
  checkout is excluded from CHECKOUT analysis*/

#pragma checkout (resume)
```

Many pragmas provide "pop" or "reset" suboptions that allow you to enable and disable pragma settings in a stack-based fashion; examples of these are provided in the relevant pragma descriptions.

IPA considerations

Interprocedural Analysis (IPA), through the IPA compiler option, is a mechanism for performing optimizations across the translation units of your C or C++ program. IPA also performs optimizations not otherwise available with the z/OS XL C/C++ compiler.

You may see changes during the IPA link step, due to the effect of a pragma. The IPA link step detects and resolves the conflicting effects of pragmas, and the conflicting effects of pragmas and compiler options that you specified for different translation units. There may also be conflicting effects between pragmas 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 pragmas, along with the resolutions.

Summary of compiler pragmas by functional category

The z/OS XL C/C++ pragmas available on the z/OS platform are grouped into the following categories:

- Language element control
- C++ template pragmas
- Floating-point and integer control
- Error checking and debugging
- Listings, messages and compiler information
- Optimization and tuning
- Object code control
- Portability and migration

For descriptions of these categories, see "Summary of compiler options" in the *z/OS XL C/C++ User's Guide*.

Language element control

Table 30. Language element control pragmas

Pragma	Description
 #pragma extension	Enables extended language features for the code that follows it.

Table 30. Language element control pragmas (continued)

Pragma	Description
z/OS #pragma filetag	Specifies the code set in which the source code was entered.
#pragma langlvl directive (C only)	Determines whether source code and compiler options should be checked for conformance to a specific language standard, or subset or superset of a standard.
z/OS #pragma margins, nomargins	Specifies the columns in the input line to scan for input to the compiler.
#pragma options (C only)	Specifies a list of compiler 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 catalogued procedures.
z/OS #pragma runopts	Specifies a list of runtime options for the compiler to use at execution time.
z/OS #pragma sequence	Defines the section of the input record that is to contain sequence numbers.
z/OS #pragma XOPTS	Passes suboptions directly to the CICS integrated translator for processing CICS statements embedded in C/C++ source code.

C++ template pragmas

Table 31. C++ template pragmas

Pragma	Description
#pragma define (C++ only)	Provides an alternative method for explicitly instantiating a template class.
#pragma implementation (C++ only)	For use with the TEMPINC compiler option, supplies the name of the file containing the template definitions corresponding to the template declarations contained in a header file.

Floating-point and integer control

Table 32. Floating-point and integer control pragmas

Pragma	Description
#pragma chars	Determines whether all variables of type char are treated as either signed or unsigned.
#pragma enum	Specifies the amount of storage occupied by enumerations.

Error checking and debugging

Table 33. Error checking and debugging pragmas

Pragma	Description
#pragma operator_new (C++ only)	Determines whether the new and new[] operators throw an exception if the requested memory cannot be allocated.

Listings, messages and compiler information

Table 34. Listings, messages and compiler information pragmas

Pragma	Description
 #pragma checkout	Controls the diagnostic messages that are generated by the compiler.
 #pragma info (C++ only)	Controls the diagnostic messages that are generated by the compiler.
 #pragma page (C only)	Specifies that the code following the pragma begins at the top of the page in the generated source listing.
 #pragma pagesize (C only)	Sets the number of lines per page for the generated source listing.
"#pragma report (C++ only)" on page 442	Controls the generation of diagnostic messages.
 #pragma skip (C only)	Skips lines of the generated source listing.
 #pragma subtitle (C only)	Places subtitle text on all subsequent pages of the generated source listing.
 #pragma title (C only)	Places title text on all subsequent pages of the generated source listing.

Optimization and tuning

Table 35. Optimization and tuning pragmas

Pragma	Description
#pragma disjoint	Lists identifiers that are not aliased to each other within the scope of their use.
 #pragma inline (C only) / noline	Specifies that a C function is to be inlined, or that a C or C++ function is not to be inlined.
#pragma isolated_call	Specifies functions in the source file that have no side effects other than those implied by their parameters.
#pragma leaves	Informs the compiler that a named function never returns to the instruction following a call to that function.
#pragma option_override	Allows you to specify optimization options at the subprogram level that override optimization options given on the command line.
#pragma reachable	Informs the compiler that the point in the program after a named function can be the target of a branch from some unknown location.
#pragma unroll	Controls loop unrolling, for improved performance.

Object code control

Table 36. Object code control pragmas

Pragma	Description
#pragma comment	Places a comment into the object module.

Table 36. Object code control pragmas (continued)

Pragma	Description
➤ z/OS #pragma csect	Identifies the name for the code, static, or test control section (CSECT) of the object module.
➤ z/OS #pragma environment (C only)	Uses C code as an assembler substitute.
➤ z/OS #pragma export	Declares that an external function or variable is to be exported.
➤ z/OS #pragma linkage (C only)	Identifies the entry point of modules that are used in interlanguage calls from C programs as well as the linkage or calling convention that is used on a function call.
➤ z/OS #pragma longname/nolongname, nolongname	Specifies whether the compiler is to generate mixed-case names that can be longer than 8 characters in the object module.
The #pragma map directive	Converts all references to an identifier to another, externally defined identifier.
#pragma pack	Sets the alignment of all aggregate members to a specified byte boundary.
#pragma priority (C++ only)	Specifies the priority level for the initialization of static objects.
➤ z/OS #pragma prolog (C only), #pragma epilog (C only)	When used with the METAL option, inserts High-Level Assembly (HLASM) prolog or epilog code for a specified function.
#pragma strings	Specifies the storage type for string literals.
➤ z/OS #pragma target (C only)	Specifies the operating system or runtime environment for which the compiler creates the object module.
➤ z/OS #pragma variable	Specifies whether the compiler is to use a named external object in a reentrant or non-reentrant fashion.

Portability and migration

Table 37. Portability and migration pragmas

Pragma	Description
➤ z/OS #pragma convert	Provides a way to specify more than one coded character set in a single compilation unit to convert string literals.
➤ z/OS #pragma convlit	Suspends the string literal conversion that the CONVLIT compiler option performs during specific portions of your program.
#pragma namemangling (C++ only)	Chooses the name mangling scheme for external symbol names generated from C++ source code.
#pragma namemanglingrule (C++ only)	Provides fined-grained control over the name mangling scheme in effect for selected portions of source code, specifically with respect to the mangling of cv-qualifiers in function parameters.
#pragma object_model (C++ only)	Sets the object model to be used for structures, unions, and classes.

Table 37. Portability and migration pragmas (continued)

Pragma	Description
z/OS #pragma wsizeof	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 z/OS XL C/C++ feature.

Individual pragma descriptions

This section contains descriptions of individual pragmas available in z/OS XL C/C++.

For each pragma, the following information is given:

Category

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

Purpose

This section provides a brief description of the effect of the pragma, and why you might want to use it.

Syntax

This section provides the syntax for the pragma. For convenience, the **#pragma name** form of the directive is used in each case. However (in C), it is perfectly valid to use the alternate C99-style `_Pragma` operator syntax; see “Pragma directive syntax” on page 393 for details.

Parameters

This section describes the suboptions that are available for the pragma, where applicable.

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

IPA considerations

For those pragmas where there are special considerations for IPA, the pragma descriptions include IPA-related information.

Examples

Where appropriate, examples of pragma directive use are provided in this section.

#pragma chars

Category

Floating-point and integer control

Purpose

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

Syntax

```
▶▶ #pragma chars ( unsignedsigned ) ▶▶
```

Defaults

-qchars=unsigned

Parameters

unsigned

Variables of type char are treated as unsigned char.

signed

Variables of type char are treated as signed char.

Usage

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

If the pragma is specified more than once in the source file, the first one will take precedence. Once specified, the pragma applies to the entire file and cannot be disabled; if a source file contains any functions that you want to compile without **#pragma chars**, place these functions in a different file.  The pragma must appear before any source statements, except for the pragmas **filetag**, **longname**, **langlvl** or **target**, which may precede it.  The pragma must appear before any source statements.

Related information

- The CHARS option in the *z/OS XL C/C++ User's Guide*

#pragma checkout

z/OS only

Category

Listings, messages, and compiler information

Purpose

Controls the diagnostic messages that are generated by the compiler.

You can suspend the diagnostics that the INFO or CHECKOUT compiler options perform during specific portions of your program. You can then resume the same level of diagnostics later in the file.  You can use this pragma directive in place of the INFO option or **#pragma info** directive.

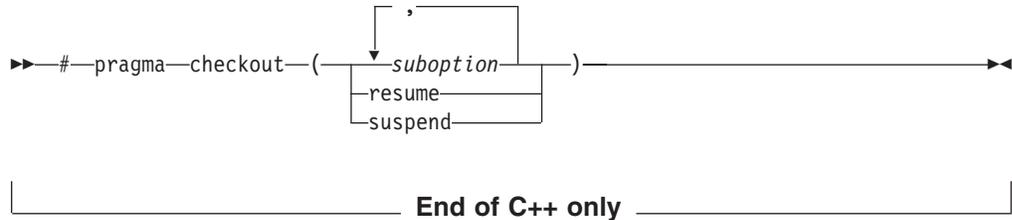
Syntax

C only

```
▶ #pragma checkout ( [resume] | [suspend] ) ▶
```

End of C only

C++ only



Defaults

See the INFO and CHECKOUT options in the *z/OS XL C/C++ User's Guide*.

Parameters

C++ *suboption*

Any suboption supported by the INFO compiler option. For details, see the INFO option in the *z/OS XL C/C++ User's Guide*.

suspend

Instructs the compiler to suspend all diagnostic operations for the code following the directive.

resume

Instructs the compiler to resume all diagnostic operations for the code following the directive.

Usage

This pragma can appear anywhere that a preprocessor directive is valid.

Related information

- “#pragma info (C++ only)” on page 413
- The INFO and CHECKOUT options in the *z/OS XL C/C++ User's Guide*

End of z/OS only

#pragma comment

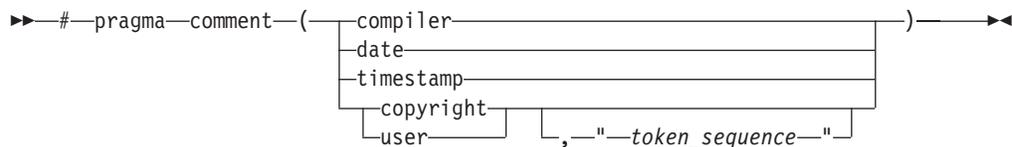
Category

Object code control

Purpose

Places a comment into the object module.

Syntax



Parameters

compiler

Appends the name and version of the compiler in an END information record at

the end of the generated object module. The name and version are not included in the generated executable, nor are they loaded into memory when the program is run.

date

The date and time of the compilation are appended in an END information record at the end of the generated object module. The date and time are not included in the generated executable, nor are they loaded into memory when the program is run.

timestamp

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. The date and time are not included in the generated executable, nor are they loaded into memory when the program is run.

If the compiler cannot find the timestamp for a source file, the directive returns Mon Jan 1 0:00:01 1990.

copyright

Places the text specified by the *token_sequence*, if any, into the generated object module. The *token_sequence* is included in the generated executable and loaded into memory when the program is run.

user

Places the text specified by the *token_sequence*, if any, into the generated object module. The characters are placed in two locations in the generated object module. One copy of the string is placed in the code image so that the string will be included in the executable load module. This copy is not necessarily loaded into memory when the program is run. A second copy of the string is placed on the END records in columns 34 to 71 for XOBJ-format object modules, or in columns 4 to 80 for GOFF-format object modules.

token_sequence

The characters in this field, if specified, must be enclosed in double quotation marks ("). This field has a 1024-byte limit.

Usage

More than one **comment** directive can appear in a translation unit, and each type of **comment** directive can appear more than once, with the exception of **copyright**, which can appear only once.

You can display the object-file comments by using the MAP option for the C370LIB utility.

IPA considerations

This directive affects the IPA compile step only if the OBJECT suboption of the IPA compiler option is in effect. With the IPA(OBJONLY) option, the pragma has the same effect as if IPA were not specified.

During the partitioning process in the IPA link step, the compiler places the text string information #pragma comment at the beginning of partition 0, the initialization partition.

#pragma convert

z/OS only

Category

Portability and migration

Purpose

Provides a way to specify more than one coded character set in a single compilation unit to convert string literals.

Unlike the related CONVLIT, ASCII/EBCDIC, and LOCALE compiler options, it allows for more than one character encoding to be used for string literals in the same compilation unit.

Syntax

```
▶▶ #pragma convert ( ccsid )  
                    " code_set_name "  
                    base  
                    source  
                    pop  
                    pop_all
```

Defaults

See the *z/OS XL C/C++ User's Guide* for information about default code pages.

Parameters

ccsid

Represents the Coded Character Set Identifier, which is an integer value between 0 and 65535 inclusive. The coded character set can be based on either EBCDIC or ASCII.

code_set_name

Is a string that specifies an ASCII or EBCDIC based codepage.

base

Represents the codepage determined by the current locale or the LOCALE compiler option. If the ASCII option has been specified, then **base** is the ISO8859-1 codepage; if the CONVLIT option has been specified, then **base** is the codepage indicated by that option. If both ASCII and CONVLIT options have been specified, then **base** is the codepage indicated by the CONVLIT option.

source

Represents the codepage the source file is written in; that is, the filetag. If there is no filetag, then **source** is the codepage indicated by the LOCALE option specified at compile time.

pop

Resets the code set to that which was previously in effect immediately before the current codepage.

pop_all

Resets the codepage to that which was in effect before the introduction of any **convert** pragmas.

Usage

The compiler options CONVLIT, ASCII/EBCDIC, and LOCALE determine the code set in effect before any **#pragma convert** directives are introduced, and after all **#pragma convert** directives are popped from the stack.

The conversion continues from the point of placement of the first **#pragma convert** directive until another **#pragma convert** directive is encountered, or until the end of the source file is reached. For every **#pragma convert** directive in your program, it is good practice to have a corresponding **#pragma convert(pop)** as well. This will prevent one file from potentially changing the codepage of another file that is included.

#pragma convert takes precedence over **#pragma convlit**.

The following are not converted:

- A string or character constant specified in hexadecimal or octal escape sequence format (because it represents the *value* of the desired character on output).
- A string literal that is part of a `#include` or `pragma` directive.
-  String literals that are used to specify linkage (for example, `extern "C"`).

Related information

- “#pragma convlit”
- The LOCALE, ASCII, and CONVLIT options in the *z/OS XL C/C++ User's Guide*

End of z/OS only

#pragma convlit

z/OS only

Category

Portability and migration

Purpose

Suspends 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.

Syntax

```
▶▶ #pragma convlit ( [resume] [suspend] ) ▶▶
```

Defaults

See the *z/OS XL C/C++ User's Guide* for information about default code pages.

Parameters

suspend

Instructs the compiler to suspend all string literal conversions for the code following the directive.

resume

Instructs the compiler to resume all string literal conversions for the code following the directive.

Usage

If you use the PPOONLY option, the compiler echoes the **convlit** pragma to the expanded source file.

Related information

- “#pragma convert” on page 402
- The CONVLIT option in the *z/OS XL C/C++ User's Guide*

End of z/OS only

#pragma csect

z/OS only

Category

Object code control

Purpose

Identifies the name for the code, static, or test control section (CSECT) of the object module.

Syntax

```
▶▶ #pragma csect ( [ CODE  
                  [ STATIC  
                  [ TEST ] ] , "name" ) ▶▶
```

Defaults

See the CSECT option in the *z/OS XL C/C++ User's Guide*.

Parameters

CODE

Specifies the CSECT that contains the executable code (C functions) and constant data.

STATIC

Designates the CSECT that contains all program variables with the `static` storage class and all character strings.

TEST

Designates the CSECT that contains debug information. You must also specify the TEST compiler option.

name

The name that is used for the applicable CSECT. The compiler does not map the name in any way. If the name is greater than 8 characters, the LONGNAME option must be in effect and you must use the binder. 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.

Usage

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

When both **#pragma csect** and the CSECT compiler option are specified, the compiler first uses the option to generate the CSECT names, and then the **#pragma csect** overrides the names generated by the option.

Examples

Suppose that you compile the following code with the option CSECT(abc) and program name foo.c.

```
#pragma csect (STATIC, "blah")
int main ()
{
    return 0;
}
```

First, the compiler generates the following csect names:

```
STATIC: abc#foo#S
CODE: abc#foo#C
TEST: abc#foo#T
```

Then the **#pragma csect** overrides the static CSECT name, which renders the final CSECT name to be:

```
STATIC: blah
CODE: abc#foo#C
TEST: abc#foo#T
```

IPA considerations

Use the **#pragma csect** directive when naming regular objects only if the OBJECT suboption of the IPA compiler option is in effect. Otherwise, the compiler discards the CSECT names that **#pragma csect** generated. With the IPA(OBJONLY) option, the pragma has the same effect as if the IPA option were not specified.

Related information

- CSECT option in the *z/OS XL C/C++ User's Guide*.

End of z/OS only

#pragma define (C++ only)

Category

Template control

Purpose

Provides an alternative method for explicitly instantiating a template class.

Syntax

```
▶▶ #pragma define (—template_class_name—) ▶▶
```

Parameters

template_class_name

The name of the template class to be instantiated.

Usage

This pragma provides equivalent functionality to standard C++ explicit instantiation, and is provided for backwards compatibility purposes only. New applications should use standard C++ explicit instantiation.

The pragma can appear anywhere an explicit instantiation statement can appear.

Examples

The following directive:

```
#pragma define(Array<char>)
```

is equivalent to the following explicit instantiation:

```
template class Array<char>;
```

Related information

- "Explicit instantiation"

#pragma disjoint

Category

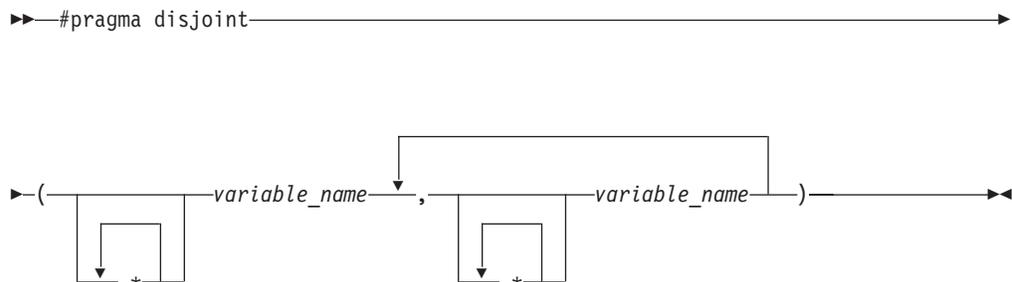
Optimization and tuning

Purpose

Lists identifiers that are not aliased to each other within the scope of their use.

By informing the compiler that none of the identifiers listed in the pragma shares the same physical storage, the pragma provides more opportunity for optimizations.

Syntax



Parameters

variable_name

The name of a variable. It must not refer to any of the following:

- A member of a structure, class, or union
- A structure, union, or enumeration tag
- An enumeration constant
- A typedef name
- A label

Usage

The **#pragma disjoint** directive asserts that none of the identifiers listed in the pragma share physical storage; if any the identifiers *do* actually share physical storage, the pragma may give incorrect results.

The pragma can appear 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.

You must declare the identifiers before using them in the pragma. Your program must not dereference a pointer in the identifier list nor use it as a function argument before it appears in the directive.

Examples

The following example shows the use of **#pragma disjoint**.

```
int a, b, *ptr_a, *ptr_b;

#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.

#pragma enum

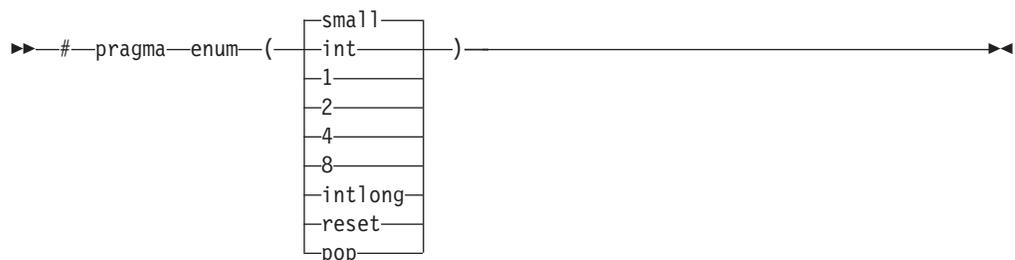
Category

Floating-point and integer control

Purpose

Specifies the amount of storage occupied by enumerations.

Syntax



Defaults

small

Parameters

small

Specifies that an enumeration occupies the minimum amount of storage required by the smallest predefined type that can represent that range of enumeration constants: either 1, 2, or 4 bytes of storage. If the specified storage size is smaller than that required by the range of the enumeration constants, the compiler issues a diagnostic message. For example:

```
#pragma enum(1)
enum e_tag {
    a=0,
    b=SHRT_MAX /* diagnostic message */
} e_var;
#pragma enum(reset)
```

int Specifies that an enumeration occupies 4 bytes of storage and is represented by `int`.

- 1 Specifies that an enumeration occupies 1 byte of storage.
- 2 Specifies that an enumeration occupies 2 bytes of storage.
- 4 Specifies that an enumeration occupies 4 bytes of storage.
- 8 Specifies that an enumeration occupies 8 bytes of storage. This suboption is only valid with LP64.

intlong

Specifies that an enumeration occupies 8 bytes of storage and is represented as a `long` if the range of the enumeration constants exceeds the limit for type `int`. Otherwise, enumerations occupy 4 bytes of storage and are of type `int`. This suboption is only valid with LP64.

reset

pop

Sets the enum setting to that which was in effect before the current setting.

Usage

The directive is in effect until the next valid **#pragma enum** directive is encountered. For every **#pragma enum** directive in your program, it is good practice to have a corresponding **#pragma enum(reset)** or **#pragma enum(pop)** as well. This is the only way to prevent one file from potentially changing the enum setting of another file that is included.

You cannot have **#pragma enum** directives within the declaration of an enumeration. The following code segment generates a warning message and the second occurrence of the pragma is ignored:

```
#pragma enum(small)
enum e_tag {
    a,
    b,
#pragma enum(int) /*cannot be within a declaration */
    c
} e_var;
```

Related information

For detailed information on the preferred sign and type for each range of enumeration constants, see the description of the `ENUMSIZE` compiler option in the *z/OS XL C/C++ User's Guide*

#pragma environment (C only)

z/OS only

Category

Object code control

Purpose

Uses C code as an assembler substitute.

The directive allows you to do the following:

- Specify a function as an entry point other than `main`
- Omit setting up a C environment on entry to the named function
- Specify several system exits that are written in C code in the same executable

Syntax

```
▶▶ #pragma environment (—identifier— [ , nolib ] )▶▶
```

Defaults

Not applicable.

Parameters

identifier

The name of the function that is to be the alternate entry point.

nolib

The Language Environment is established, but the LE runtime library is not loaded at run time. If you omit this argument, the library is loaded.

Usage

If you specify any other value than **nolib** after the function name, behavior is not defined.

End of z/OS only

#pragma export

z/OS only

Category

Object code control

Purpose

Declares that an external function or variable is to be exported.

The pragma 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.

Syntax

#pragma export

►► #pragma export (—*identifier*—) ◀◀

Defaults

Not applicable.

Parameters

identifier

The name of a variable or function to be exported.

Usage

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.

If the specification for a `const` variable in a **#pragma export** directive conflicts with the `ROCONST` option, the pragma directive takes precedence over the compiler option, and the compiler issues an informational message. The `const` variable gets exported and it is considered reentrant.

The `main` function can not be exported.

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.

Related information

- “The `_Export` qualifier (C++ only)” on page 100
- “The `_Export` function specifier (C++ only)” on page 197

End of z/OS only

#pragma extension

z/OS only

Category

Language element control

Purpose

Enables extended language features for the code that follows it.

Syntax

►► #pragma extension [(—*pop*—)] ◀◀

Defaults

See the description of the `LANGLVL` compiler option in the *z/OS XL C/C++ User's Guide*.

Parameters

pop

Reverts the language level setting to the previous one defined for the file (if any).

Usage

The directive must only occur outside external declarations.

Multiple **#pragma extension** and **#pragma extension(pop)** directive pairs can appear in the same file. You should place a **#pragma extension(pop)** directive at the end of the section of code to which the **#pragma extension** applies within the same file; if you do not do so, the compiler will insert a pop directive.

Do not pop a **#pragma extension** directive from within a nested include file.

C When **#pragma langlvl** is embedded in **#pragma extension** and **#pragma extension (pop)** directives, an informational message is issued and **#pragma langlvl** is ignored.

Examples

The following example shows how **#pragma extension** is applied to an included file, assuming that the default language level setting is ANSI:

```
#pragma extension
#include_next <stddef.h> /* C++: langlvl = extended; C: langlvl = extc89 */
#pragma extension(pop)
#include <pthread.h> /* langlvl = ansi */
```

The following example shows how **#pragma extension** is applied to a section of code, assuming that the default language level setting is ANSI:

```
int alignofChar()
{
return __alignof__(char); /* langlvl = ansi, __alignof__ is treated as an
                           identifier and an error message
                           is issued */
}
#pragma extension

int alignofInt() /* C++: langlvl = extended; C: langlvl = extc89 */
{
return __alignof__(int); /* __alignof__ is treated as a keyword
                           and no error message is issued */
}
```

Related information

- “#pragma langlvl directive (C only)” on page 417
- The LANGLVL option in the *z/OS XL C/C++ User's Guide*

End of z/OS only

#pragma filetag

z/OS only

Category

Language element control

Purpose

Specifies the code set in which the source code was entered.

Syntax

```
▶▶ #pragma filetag (—"code_set_name"—) ▶▶
```

Defaults

See the *z/OS XL C/C++ User's Guide* for information about default code pages.

Parameters

code_set_name

The name of the source code set.

Usage

Since the # character is variant between code sets, use the trigraph representation **??=** instead of # as illustrated below.

The **#pragma filetag** directive can appear only once per source file. It must appear before the first statement or directive, except for conditional compilation directives or the #line directive. For example:

```
??=line 42
??=ifdef COMPILER_VER      /* This is allowed. */
??=pragma filetag ("code set name")
??=endif
```

The **#pragma filetag** directive should not appear in combination with any other **#pragma** directives. For example, the following is incorrect:

```
??=pragma filetag ("IBM-1047") export (baffle_1)
```

If there are comments before the pragma, the compiler does not translate them to the code page that is associated with the LOCALE option.

Related information

- The LOCALE, ASCII, and CONVLIT options in the *z/OS XL C/C++ User's Guide*

End of z/OS only

#pragma implementation (C++ only)

Category

Template control

Purpose

For use with the TEMPINC compiler option, supplies the name of the file containing the template definitions corresponding to the template declarations contained in a header file.

Syntax

```
▶▶ #pragma implementation (—"file_name"—) ▶▶
```

Parameters

file_name

The name of the file containing the definitions for members of template classes declared in the header file.

Usage

This pragma is not normally required if your template implementation file has the same name as the header file containing the template declarations, and a .c extension. You only need to use the pragma if the template implementation file does not conform to this file-naming convention. For more information about using template implementation files, see "Using C++ Templates".

#pragma implementation is only effective if the TEMPINC option is in effect. Otherwise, the pragma has no meaning and is ignored.

The pragma can appear in the header file containing the template declarations, or in a source file that includes the header file. It can appear anywhere that a declaration is allowed.

Related information

- The TEMPINC option in the *z/OS XL C/C++ User's Guide*
- "Using C++ Templates"

#pragma info (C++ only)

Category

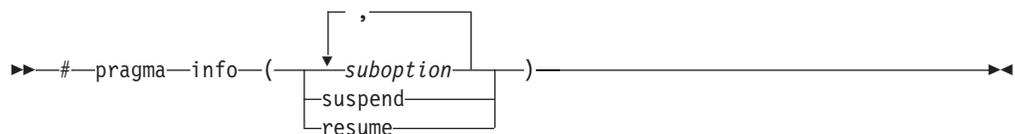
Listings, messages, and compiler information

Purpose

Controls the diagnostic messages that are generated by the compiler.

You can use this pragma directive in place of the INFO option.

Syntax



Defaults

See the INFO option in the *z/OS XL C/C++ User's Guide*.

Parameters

suboption

Any suboption allowed by the INFO compiler option. For details, see the description of the INFO option in the *z/OS XL C/C++ User's Guide*.

suspend

Suspends the diagnostics that the pragma or INFO compiler option performs during specific portions of your program.

resume

Resumes the same level of diagnostics in effect before the **suspend** pragma was specified.

Related information

- “#pragma checkout” on page 399
- The INFO option in the *z/OS XL C/C++ User's Guide*

#pragma inline (C only) / noinline

z/OS only

Category

Optimization and tuning

Purpose

Specifies that a C function is to be inlined, or that a C or C++ function is not to be inlined.

Syntax

C only

```
▶▶ #pragma inline  
noinline (—identifier—)▶▶
```

End of C only

C++ only

```
▶▶ #pragma noinline (—identifier—)▶▶
```

End of C++ only

Defaults

noinline

Parameters

C only

inline

Inlines the named function on every call, provided you have specified the INLINE or the OPT compiler options (otherwise it has no effect). The function is inlined in both selective (NOAUTO) and automatic (AUTO) mode.

End of C only

noinline

Prevents the named function from being inlined on any call, disabling any effects of the INLINE or OPT compiler options (the pragma has no effect in selective (NOAUTO) mode). It also takes precedence over the C/C++ keyword inline.

identifier

The name of a function to be included or excluded for inlining.

Usage

 The directive must be at file scope.  The directive can be placed anywhere.

IPA considerations

If you use either the **#pragma inline** or the **#pragma noline** directive in your source, you can later override them with an appropriate IPA link control file directive during the IPA link step. 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.

Related information

- “The inline function specifier” on page 190
- The `INLINE` option in the *z/OS XL C/C++ User's Guide*

End of z/OS only

#pragma isolated_call

Category

Optimization and tuning

Purpose

Specifies functions in the source file that have no side effects other than those implied by their parameters.

Essentially, any change in the state of the runtime environment is considered a side effect, including:

- Accessing a volatile object
- Modifying an external object
- Modifying a static object
- Modifying a file
- Accessing a file that is modified by another process or thread
- Allocating a dynamic object, unless it is released before returning
- Releasing a dynamic object, unless it was allocated during the same invocation
- Changing system state, such as rounding mode or exception handling
- Calling a function that does any of the above

Marking a function as isolated indicates to the optimizer that external and static variables cannot be changed by the called function and that pessimistic references to storage can be deleted from the calling function where appropriate. Instructions can be reordered with more freedom, resulting in fewer pipeline delays and faster execution in the processor. Multiple calls to the same function with identical parameters can be combined, calls can be deleted if their results are not needed, and the order of calls can be changed.

Syntax

```
▶▶ #pragma isolated_call (—function—) ▶▶
```

Defaults

Not applicable.

Parameters

function

The name of a function that does not have side effects or does not rely on functions or processes that have side effects. *function* is a primary expression that can be an identifier, operator function, conversion function, or qualified name. An identifier must be of type function or a typedef of function. 

If the name refers to an overloaded function, all variants of that function are marked as isolated calls.

Usage

The only side effect that is allowed for a function named in the pragma is modifying the storage pointed to by any pointer arguments passed to the function, that is, calls by reference. The function is also permitted to examine non-volatile external objects and return a result that depends on the non-volatile state of the runtime environment. Do not specify a function that causes any other side effects; that calls itself; or that relies on local static storage. If a function is incorrectly identified as having no side effects, the program behavior might be unexpected or produce incorrect results.

The **#pragma isolated_call** directive can be placed at any point in the source file, before or after calls to the function named in the pragma.

Predefined macros

None.

Examples

The following example shows you when to use the **#pragma isolated_call** directive (on the `addmult` function). It also shows you when not to use it (on the `same` and `check` functions):

```
#include <stdio.h>
#include <math.h>

int addmult(int op1, int op2);
#pragma isolated_call(addmult)

/* This function 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 function '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 function '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 effects

If you specify this pragma in your source code in the IPA compile step, you cannot override the effects of the pragma on the IPA link step.

#pragma langlvl directive (C only)

Category

Language element control

Purpose

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

This pragma is equivalent to the LANGLVL compiler option.

Syntax

```

▶▶ #pragma langlvl (ansi | common | extc89 | extc99 | extended | saa | saa12 | stdc89 | stdc99)

```

Defaults

See the LANGLVL option in the *z/OS XL C/C++ User's Guide*.

Parameters

ansi

Allows only language constructs that support the ISO C standards. It does not permit packed decimal types and issues an error message when it detects assignment between integral types and pointer types.

extc89

Allows language constructs that support the ISO C89 standard, and accepts implementation-specific language extensions.

extc99

Allows language constructs that support the ISO C99 standard, and accepts implementation-specific language extensions.

extended

The option permits packed decimal types and it issues a warning message when it detects assignment between integral types and pointer types.

commonc

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 K&R C.

saa

Compilation conforms to the SAA[®] C Level 2 CPI language definition.

saal2

Compilation conforms to the SAA C Level 2 CPI language definition, with some exceptions.

stdc89

Allows language constructs that support the ISO C89 standard.

stdc99

Allows language constructs that support the ISO C99 standard.

Usage

You can only specify this pragma only once in a source file, and must appear before any statements. The compiler uses predefined macros in the header files to make declarations and definitions available that define the specified language level. When both the pragma and the compiler option are specified, the compiler option takes precedence over the pragma. Note that if you would like to specify the **extc89** language level, you can also do by using **#pragma extension**.

Note: In z/OS UNIX System Services, if the **c89** environment variable `{_NOCMDOPTS}` is set to 1, **#pragma langlvl** has no effect. You must use the compiler option `LANGLVL` instead of the pragma.

Related information

- “Standard pragmas (C only)” on page 391
- “#pragma extension” on page 410
- The `LANGLVL` option in the *z/OS XL C/C++ User's Guide*

#pragma leaves**Category**

Optimization and tuning

Purpose

Informs the compiler that a named function never returns to the instruction following a call to that function.

By informing the compiler that it can ignore any code after the function, the directive allows for additional opportunities for optimization.

This pragma is commonly used for custom error-handling functions, in which programs can be terminated if a certain error is encountered.

Syntax

```
▶▶ #pragma leaves ( function_name ) ▶▶
```

Parameters

function_name

The name of the function that does not return to the instruction following the call to it.

Defaults

Not applicable.

Usage

If you specify the LIBANSI compiler option (which informs the compiler that function names that match functions in the C standard library are in fact C library functions), the compiler checks whether the longjmp family of functions (longjmp, _longjmp, siglongjmp, and _siglongjmp) contain **#pragma leaves**. If the functions do not contain this pragma directive, the compiler will insert this directive for the functions. This is not shown in the listing.

Examples

```
#pragma leaves(handle_error_and_quit)
void test_value(int value)
{
    if (value == ERROR_VALUE)
    {
        handle_error_and_quit(value);
        TryAgain(); // optimizer ignores this because
                   // never returns to execute it
    }
}
```

IPA considerations

If you specify the **#pragma leaves** directive in your source code in the IPA compile step, you cannot override the effects of this directive in the IPA link step.

Related information

- “#pragma reachable” on page 441.

#pragma linkage (C only)

z/OS only

Category

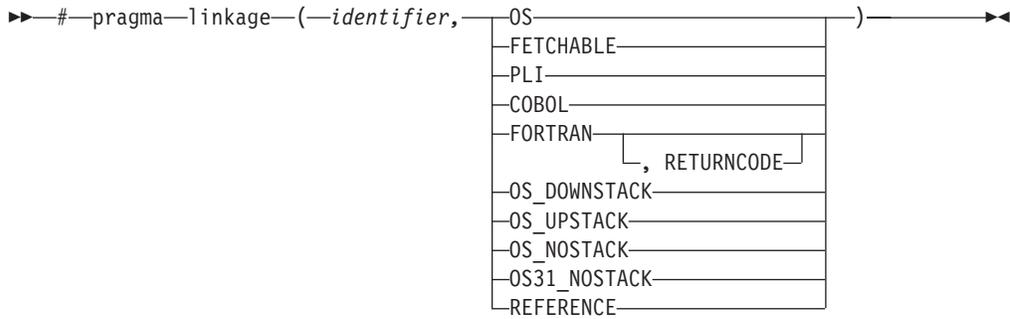
Object code control

Purpose

Identifies the entry point of modules that are used in interlanguage calls from C programs as well as the linkage or calling convention that is used on a function call.

The directive also designates other entry points within a program that you can use in a fetch operation.

Syntax



Defaults

C linkage.

Parameters

identifier

The name of a function that is to be the entry point of the module, or a typedef name that will be used to define the entry point. (See below for an example.)

FETCHABLE

Indicates that *identifier* can be used in a fetch operation. A fetched XPLINK module must have its entry point defined with a **#pragma linkage(..., fetchable)** directive.

OS

Designates *identifier* as an OS linkage entry point. OS linkage is the basic linkage convention that is used by the operating system. If the caller is compiled with NOXPLINK, on entry to the called routine, its register 13 points to a standard Language Environment stack frame, beginning with a 72-byte save area. The stack frame is compatible with Language Environment languages that expect by-reference calling conventions and with the Language Environment-supplied assembler prologue macro. If the caller is compiled with XPLINK, the behavior depends on the OSCALL suboption of the XPLINK compiler option. This suboption selects the behavior for linkage OS from among OS_DOWNSTACK, OS_UPSTACK, and OS_NOSTACK (the default). This means that applications which use linkage OS to communicate among C or C++ functions will need source changes when recompiled with XPLINK. See the description that follows for REFERENCE.

PLI

Designates *identifier* as a PL/I linkage entry point.

COBOL

Designates *identifier* as a COBOL linkage entry point.

FORTRAN

Designates *identifier* as a FORTRAN linkage entry point.

RETURNCODE indicates to the compiler that the routine named by *identifier* is a FORTRAN routine, which returns an alternate return code. It also indicates that the routine is defined outside the translation unit. You can retrieve the return code by using the `fortrc` function. If the compiler finds the function definition inside the translation unit, it issues an error message. Note that you can define functions outside the translation unit, even if you do not specify the RETURNCODE keyword.

OS_DOWNSTACK

Designates *identifier* as an OS linkage entry point in XPLINK mode with a downward growing stack frame.

If the function identified by *identifier* is defined within the translation unit and is compiled using the NOXPLINK option, the compiler issues an error message.

OS_UPSTACK

Designates *identifier* as an OS linkage entry point in XPLINK mode with a traditional upward growing stack frame.

This linkage is required for a new XPLINK downward-stack routine to be able to call a traditional upward-stack OS routine. This linkage explicitly invokes compatibility code to swap the stack between the calling and the called routines.

If the function identified by *identifier* is defined within the translation unit and is compiled using the XPLINK option, the compiler issues an error message. Typically, the *identifier* will not be defined in a compilation. This is acceptable. In this case, it is a reference to an external procedure that is separately compiled with NOXPLINK.

OS_NOSTACK

Designates *identifier* as an OS linkage entry point in XPLINK mode with no preallocated stack frame. An argument list is constructed containing the addresses of the actual arguments. The last item in this list has its high order bit set. Register 1 is set to point to this argument list. Register 13 points to a 72-byte save area that may not be followed by z/OS Language Environment control structures, such as the NAB. Register 14 contains the return address. Register 15 contains the entry point of the called function. This is synonymous with OS31_NOSTACK.

OS31_NOSTACK

Designates *identifier* as an OS linkage entry point in XPLINK mode with no preallocated stack frame.

REFERENCE

This is synonymous with OS_UPSTACK in non-XPLINK mode and synonymous with OS_DOWNSTACK in XPLINK mode. Unlike the linkage OS, this is not affected by the OSCALL suboption of XPLINK. Consider using this option instead to make the source code portable between XPLINK and non-XPLINK.

Usage

You can use a typedef in a **#pragma linkage** directive to associate a specific linkage convention with a function type. In the following example, the directive associates the OS linkage convention with the typedef `func_t`:

```
typedef void func_t(void);
#pragma linkage (func_t,OS)
```

This typedef can then be used in C declarations wherever a function should have OS linkage. In the following example:

```
func_t myfunction;
```

`myfunction` is declared as having type `func_t`, which is associated with OS linkage; `myfunction` would therefore have OS linkage.

Related information

- The XPLINK option in the *z/OS XL C/C++ User's Guide*

End of z/OS only

#pragma longname/nolongname

z/OS only

Category

Object code control

Purpose

Specifies whether the compiler is to generate mixed-case names that can be longer than 8 characters in the object module.

Syntax

```
▶▶ #pragma longname  
          |  
          └─ nolongname
```

Defaults

- **C** nolongname
- **C++** longname

Parameters

longname

The compiler generates mixed-case names in the object module. Names can be up to 1024 characters in length.

nolongname

The compiler generates truncated and uppercase names in the object module.

C++ Only functions that do not have C++ linkage are given truncated and uppercase names.

Usage

If you use the **#pragma longname** directive, 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 or LONGNAME compiler option, the compiler ignores the **#pragma longname** directive. If you specify either **#pragma nolongname** or the NOLONGNAME compiler 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.

C only

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.

End of C only

C++ only

You must specify **#pragma longname/nolongname** before any code. Otherwise, the compiler issues a warning message. **#pragma nolongname** must also precede any other preprocessing directive.

End of C++ only

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** or **cc** utility from z/OS UNIX System Services, both of which already specify the LONGNAME compiler option). Otherwise, you will receive an unrecoverable compiler error.

Related information

- “The #pragma map directive”

End of z/OS only

The #pragma map directive

Category

Object code control

Purpose

The **#pragma map** directive instructs the compiler to convert all references to an identifier to another, externally defined identifier. It is typically used to map an identifier for a data object or function that is declared and used inside a program module to an existing external name defined outside of the program module. For example, it can be used to allow a program to reference a function or data object defined in an external module written in a different language, such as assembler or Fortran, without needing to follow the naming conventions required by the other language.

Syntax

#pragma map syntax – C

```
▶▶ #pragma map (—identifier—, —"—name—"—) ▶▶
```

#pragma map syntax – C++

```
▶▶ #pragma map (—identifier—(—argument_list—), —"—name—"—) ▶▶
```

Parameters

identifier

The name used in the source code.  The *identifier* can represent a data object or function with external linkage.  The *identifier* can represent a data object, a non-overloaded or overloaded function, or overloaded operator, with external linkage. If the name to be mapped is not in the global namespace, it must be fully qualified.

The *identifier* should be declared in the same compilation unit in which it is referenced, but should not be defined in any other compilation unit. The *identifier* must not be used in another **#pragma map** directive anywhere in the program.

▶ C++ *argument_list*

The list of arguments for the overloaded function or operator function designated by the *identifier*. If the *identifier* designates an overloaded function, the function must be parenthesized and must include its argument list if it exists. If the *identifier* designates a non-overloaded function, only *identifier* is required, and the parentheses and argument list are optional.

name

The name that will appear in the object code. ▶ C The *name* can represent a data object or function with external linkage. The compiler preserves mixed-case names. If the name is longer than 8 characters, you must use the binder and specify the LONGNAME compiler option. ▶ C++ The *name* can represent a data object, a non-overloaded or overloaded function, or overloaded operator, with external linkage. If the name is longer than 8 characters you must use the binder. If the name designates a function with C++ linkage, you must specify the mangled name. Alternatively, if the function is declared with extern "C" linkage, the mangled name is not required in the pragma.

If the name exceeds 65535 bytes, an informational message is emitted and the pragma is ignored.

The *name* may or may not be declared in the same compilation unit in which the *identifier* is referenced, but must not be defined in the same compilation unit. Also, the *name* should not be referenced anywhere in the compilation unit where *identifier* is referenced. The *name* must not be the same as that used in another **#pragma map** directive or **#pragma csect** directive in the same compilation unit. The map *name* is not affected by the CONVLIT or the ASCII compiler options.

Usage

The **#pragma map** directive can appear anywhere in the program. Note that in order for a function to be actually mapped, the map target function (*name*) must have a definition available at link time (from another compilation unit), and the map source function (*identifier*) must be called in your program.

You cannot use **#pragma map** with compiler built-in functions.

Examples

The following is an example of **#pragma map** used to map a function name (using the mangled name for the map name in C++):

```
/* Compilation unit 1: */  
  
#include <stdio.h>  
  
void foo();  
extern void bar(); /* optional */  
  
#if __cplusplus  
#pragma map (foo, "bar__Fv")  
#else  
#pragma map (foo, "bar")  
#endif
```


You can also use an asterisk (*) to indicate 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 input record.

Usage

The pragmas override the MARGINS or NOMARGINS compiler options. The 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.

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.

Related information

- “#pragma sequence” on page 445
- The MARGINS compiler options in the *z/OS XL C/C++ User's Guide*

End of z/OS only

#pragma namemangling (C++ only)

Category

Portability and migration

Purpose

Chooses the name mangling scheme for external symbol names generated from C++ source code.

Syntax

```

▶▶ #pragma namemangling ( [ansi | zosv1r5_default | zosv1r5_ansi | zosv1r2 | osv2r10 | compat] [, num_chars] )

```

Defaults

See the NAMEMANGLING option in the *z/OS XL C/C++ User's Guide*.

Parameters

ansi

Indicates that the name mangling scheme complies with the C++ standard. The default value for *num_chars* is 32767 characters, which is the maximum.

zosv1r5_ansi

Use this scheme for compatibility with link modules from z/OS C++ Version 1 Release 5 that were created with the **#pragma namemangling(ansi)** directive or with the NAMEMANGLING(ANSI) compiler option in effect. The default value for *num_chars* is 32767 characters, which is the maximum.

zosv1r5_default

Use this scheme for compatibility with link modules from z/OS C++ Version 1 Release 5 that were created with the default mangling for that compiler. This suboption uses the same name mangling scheme as modules created with z/OS C++ Version 1 Release 2 that were created with the **#pragma namemangling(ansi)** directive or with the NAMEMANGLING(ANSI) compiler option in effect. The default value for *num_chars* is 1024 characters, which is the maximum.

zosv1r2

Same semantics as the **zosv1r5_default** suboption: instructs the compiler that the name mangling scheme is compatible with z/OS V1R2 link modules that were created with NAMEMANGLING(ANSI).

osv2r10

Use this scheme for compatibility with link modules created with the OS/390 C++ Version 2 Release 10 compiler or earlier versions, or with link modules that were created with the **#pragma namemangling(compat)** directive or with the NAMEMANGLING(COMPAT) compiler option in effect. The default value for *num_chars* is 255 characters, which is the maximum.

compat

Same semantics as the **osv2r10** suboption: instructs the compiler that the name mangling scheme is compatible with that in OS/390 V2R10 and earlier versions.

num_chars

Represents a maximum for the length of the mangled name.

pop

Restores the name mangling scheme to that which was in effect immediately before the current setting. If no previous name mangling scheme was specified in the file, the scheme specified by the NAMEMANGLING compiler option is used.

Usage

For every **#pragma namemangling** directive in your program, it is good practice to have a corresponding **#pragma namemangling(pop)** as well. In this way, it is possible to prevent one file from potentially changing the name mangling setting of another file that is included.

Related information

- “#pragma namemanglingrule (C++ only)”
- The NAMEMANGLING option in the *z/OS XL C/C++ User's Guide*

#pragma namemanglingrule (C++ only)

Category

Portability and migration

Purpose

Provides fined-grained control over the name mangling scheme in effect for selected portions of source code, specifically with respect to the mangling of cv-qualifiers in function parameters.

When a function name is mangled, repeated function arguments of the same type are encoded according to the following compression scheme:

parameter → T *param number* [] #single repeat of a previous parameter
 → N *repetition digit param number* [] #2 to 9 repetitions

where:

param number

Indicates the number of the previous parameter which is repeated. It is followed by an underscore () if *param number* contains multiple digits.

repetition digit

Must be greater than 1 and less than 10. If an argument is repeated more than 9 times, this rule is applied multiple times. For example, a sequence of 38 parameters that are the same as parameter 1 mangles to N91N91N91N91N21.

The **#pragma namemanglingrule** directive allows you to control whether top-level cv-qualifiers are mangled in function parameters.

Syntax

▶ #pragma namemanglingrule ((fparmtype ,

off
on
pop

) ▶

Defaults

- **fparmtype, on** when the **#pragma namemangling(ansi)** directive or the **NAMEMANGLING(ANSI)** compiler option is in effect. Otherwise the default is **fparmtype,off**.

Parameters

fparmtype, on

Top-level cv-qualifiers are not encoded in the mangled name of a function parameter. Also, top-level cv-qualifiers are ignored when repeated function parameters are compared for equivalence; function parameters that differ only by the use of a top-level cv-qualifier are considered equivalent and are mangled according to the compressed encoding scheme.

fparmtype, off

Top-level cv-qualifiers are encoded in the mangled name of a function parameter. Also, repeated function parameters that differ by the use of cv-qualifiers are not considered equivalent and are mangled as separate parameters. This setting is compatible with z/OS C++ Version 1 Release 2.

fparmtype, pop

Reverts to the previous **fparmtype** setting in effect. If no previous settings are in effect, the default **fparmtype** setting is used.

Note: This pragma fixes function signature ambiguities in 32-bit mode but it is not needed in 64-bit mode since those ambiguities do exist in 64-bit mode.

Usage

#pragma namemanglingrule is allowed in global, class, and function scopes. It has no effect on a block scope function declaration with external linkage.

Different pragma settings can be specified in front of function declarations and definitions. If **#pragma namemanglingrule** settings in subsequent declarations and definitions conflict, the compiler ignores those settings and issues a warning message.

Examples

The following table shows the effects of this pragma applied to different function signatures.

Table 38. Mangling of function parameters with top-level cv-qualifiers

Source name	Mangled name	
	fnparmttype, off	fnparmttype, on
void foo (const int)	foo__FCi	foo__Fi
void foo (int* const)	foo__FCPi	foo__FPi
void foo (int** const)	foo__FCPPi	foo__FPPi
void foo (int, const int)	foo__FiCi	foo__FIT1

Related information

- “#pragma namemangling (C++ only)” on page 426
- The NAMEMANGLING option in the *z/OS XL C/C++ User's Guide*

#pragma object_model (C++ only)

Category

Portability and migration

Purpose

Sets the object model to be used for structures, unions, and classes.

The object models differ in the areas of layout for the virtual function table, support for virtual base classes, and name mangling scheme.

Syntax

```
▶▶ #pragma object_model ( compat | ibm | pop ) ▶▶
```

Defaults

compat

Parameters

compat

Is compatible with name mangling and the virtual function table that was available with the previous releases of the z/OS C++ compiler.

ibm

Provides improved performance. Class hierarchies with many virtual base classes can benefit from this option because the size of the derived class is smaller and access to the virtual function table is faster. You must use XPLINK when specifying this option.

pop

Sets the object model setting to that which was in effect before the current setting.

Usage

Classes implicitly inherit the object model of their parent, overriding any local object model specification. All classes in the same inheritance hierarchy must have the same object model. An error is generated if, through multiple inheritance, the object models for base classes are mixed.

Examples

The following example illustrates how the inheritance mechanism supersedes the **#pragma object_model** specification.

```
#pragma object_model(ibm)
class A{}; // ibm model: pragma is used
#pragma object_model(compat)
class B: A{}; // ibm model: pragma is ignored because of
// inheritance
// (A is "ibm", therefore B is "ibm")
#pragma object_model(ibm)
class C: B{}; // ibm model: pragma is ignored because of
// inheritance
// (B is "ibm", therefore C is "ibm")
#pragma object_model(compat)
class D{}; // compat model: no inheritance, pragma is
// used
class E: A, D{}; // ERROR: A and D have differing object
// models.
#pragma object_model(compat)
class F: B, C{}; // ibm model: pragma is ignored because
// of inheritance
// (B and C are "ibm", therefore F is "ibm")
// Same object model for the inheritance hierarchy
```

Related information

- The OBJECTMODEL option in the *z/OS XL C/C++ User's Guide*

#pragma operator_new (C++ only)

Category

Error checking and debugging

Purpose

Determines whether the `new` and `new[]` operators throw an exception if the requested memory cannot be allocated.

This pragma is equivalent to the `LANGLVL(NEWEXCP)` option.

Syntax

```
▶▶ #pragma operator_new ( ( returnsnull ) throwexception ) ▶▶
```

Defaults

returnsnull

Parameters

returnsnull

If the memory requested by the `new` operator cannot be allocated, the compiler returns 0, the null pointer. Use this option for compatibility with versions of the XL C++ compiler previous to V1R7.

throwexception

If the memory requested by the new operator cannot be allocated, the compiler throws a standard exception of type `std::bad_alloc`. Use this option in new applications, for conformance with the C++ standard.

Usage

The pragma can be specified only once in a source file. It must appear before any statements in the source file. This pragma takes precedence over the `LANGLVL(NEWEXCP)` compiler option.

Restrictions

This pragma applies only to versions of the new operator that throw exceptions; it does not apply to the `nothrow` or empty `throw` versions of the new operator (for the prototypes of all the new operator versions, see the description of the `<new>` header in the *Standard C++ Library Reference*). It also does not apply to class-specific new operators, user-defined new operators, and new operators with placement arguments.

Related information

- “new expressions (C++ only)” on page 151
- “Allocation and deallocation functions (C++ only)” on page 210
- The `LANGLVL` compiler option in the *z/OS XL C/C++ User's Guide*

#pragma option_override

Category

Optimization and tuning

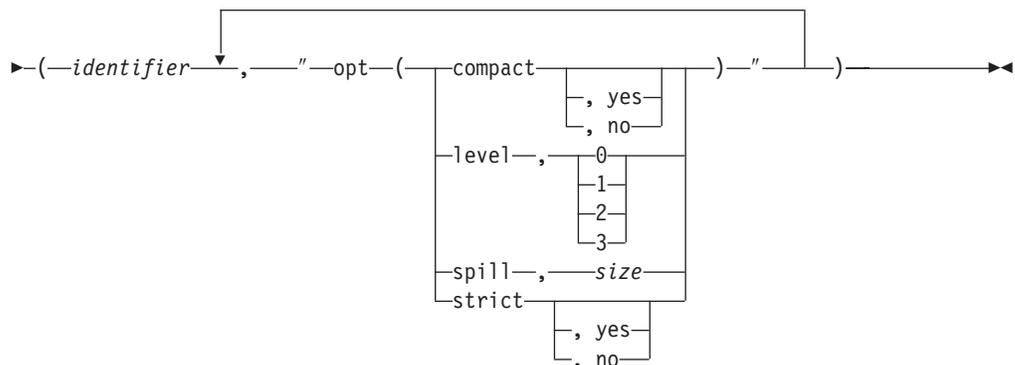
Purpose

Allows you to specify optimization options at the subprogram level that override optimization options given on the command line.

This enables finer control of program optimization, and can help debug errors that occur only under optimization.

Syntax

▶▶ #pragma option_override



Parameters

identifier

The name of a function for which optimization options are to be overridden.

The following table shows the equivalent command line option for each pragma suboption.

#pragma option_override value	Equivalent compiler option
compact	COMPACT
compact, yes	
compact, no	NOCOMPACT
level, 0	OPT(0)
level, 1	OPT(1)
level, 2	OPT(2)
level, 3	OPT(3)
spill, <i>size</i>	SPILL(<i>size</i>)
strict	STRICT
strict, yes	
strict, no	NOSTRICT

Defaults

See the descriptions in the *z/OS XL C/C++ User's Guide* of the options listed in the table above for default settings.

Usage

The pragma takes effect only if optimization is already enabled by a command-line option. You can only specify an optimization level in the pragma *lower* than the level applied to the rest of the program being compiled.

The **#pragma option_override** directive only affects functions that are defined in the same compilation unit. The pragma directive can appear anywhere in the translation unit. That is, it can appear before or after the function definition, before or after the function declaration, before or after the function has been referenced, and inside or outside the function definition.

 This pragma cannot be used with overloaded member functions.

Examples

Suppose you compile the following code fragment containing the functions `foo` and `faa` using `OPT(1)`. Since it contains the `#pragma option_override(faa, "opt(level, 0)")`, function `faa` will not be optimized.

```
foo(){  
    .  
    .  
    .  
}  
  
#pragma option_override(faa, "opt(level, 0)")  
  
faa(){
```

```
.  
. .  
}
```

IPA considerations

You cannot specify the IPA compiler option for **#pragma option_override**.

During IPA compile processing, subprogram-specific options will be used to control IPA compile-time optimizations.

During IPA link processing, subprogram-specific options will be used to control IPA link-time optimizations, as well as program partitioning. They will be retained, even if the related IPA link command line option is specified.

Related information

- The OPT, COMPACT, SPILL and STRICT options in the *z/OS XL C/C++ User's Guide*

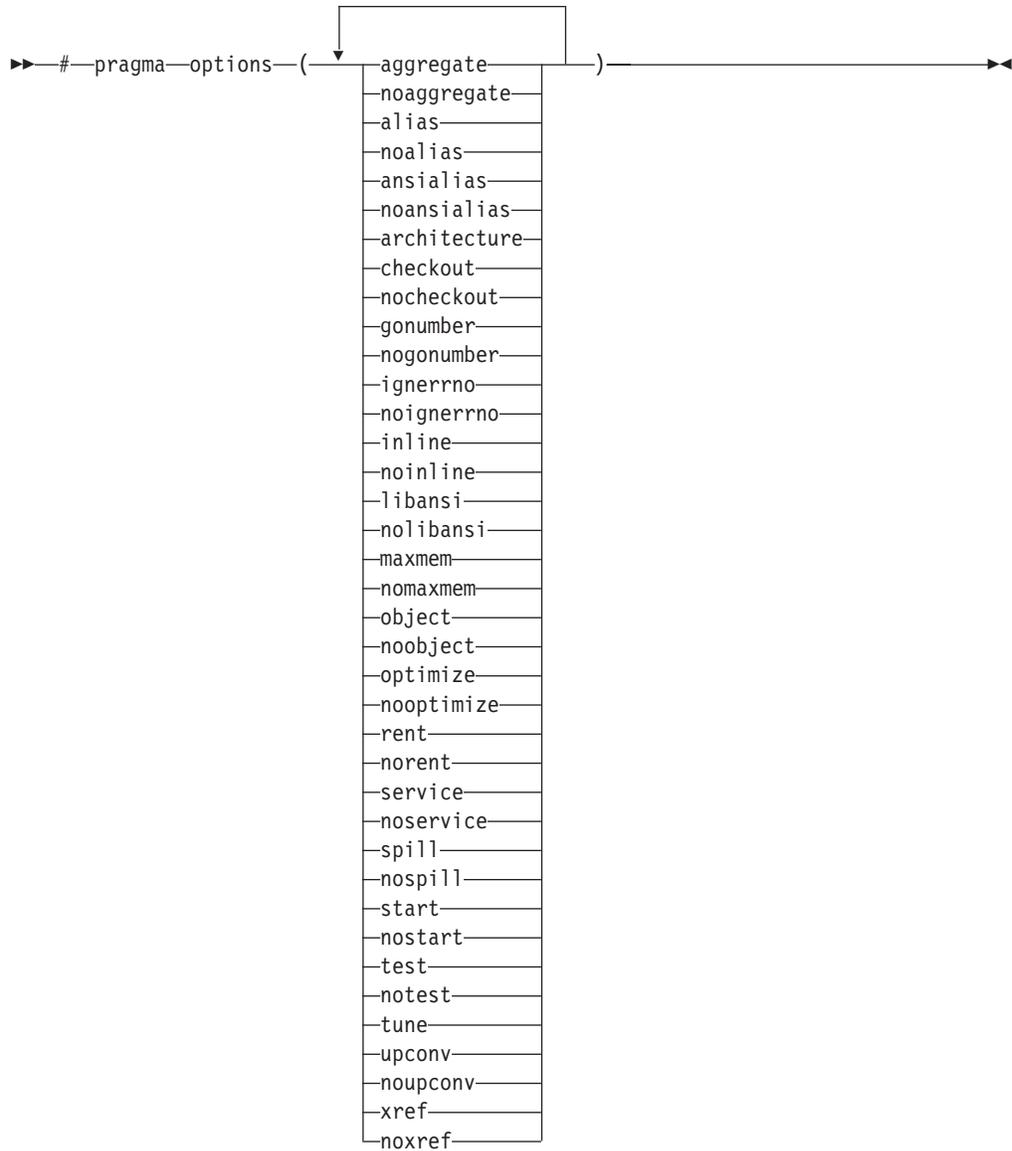
#pragma options (C only)

Category

Language element control

Purpose

Syntax



Defaults

See the *z/OS XL C/C++ User's Guide* for the default settings for these options.

Parameters

See the *z/OS XL C/C++ User's Guide* for descriptions of these options.

Usage

If you use a compile option that contradicts the options that are specified on the `#pragma options` directive, the compiler 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.

IPA considerations

You cannot specify the IPA compiler option for `#pragma options`.

Related information

- *z/OS XL C/C++ User's Guide*

#pragma pack

Category

Object code control

Purpose

Sets the alignment of all aggregate members to a specified byte boundary.

If the byte boundary number is smaller than the natural alignment of a member, padding bytes are removed, thereby reducing the overall structure or union size.

Syntax

```
▶▶ #pragma pack ( number ) ▶▶
```

<i>number</i>
full
packed
twobyte
reset

Defaults

Members of aggregates (structures, unions, and classes) are aligned on their natural boundaries and a structure ends on its natural boundary. The alignment of an aggregate is that of its strictest member (the member with the largest alignment requirement).

Parameters

number

is one of the following:

- 1 Aligns structure members on 1-byte boundaries, or on their natural alignment boundary, whichever is less.
- 2 Aligns structure members on 2-byte boundaries, or on their natural alignment boundary, whichever is less.
- 4 Aligns structure members on 4-byte boundaries, or on their natural alignment boundary, whichever is less.
- 8 Reserved for possible future use.
- 16 Reserved for possible future use.

▶ **z/OS** *z/OS only full*

Aligns structure members on 4-byte boundaries, or on their natural alignment boundary, whichever is less. This is the same as **#pragma pack()** and **#pragma pack(4)**.

▶ **z/OS** *z/OS only packed*

Aligns structure members on 1-byte boundaries, or on their natural alignment boundary, whichever is less. This is the same as **#pragma pack(1)**.

▶ **z/OS** *z/OS only twobyte*

Aligns structure members on 2-byte boundaries, or on their natural alignment boundary, whichever is less. This is the same as **#pragma pack(2)**.

reset

Sets the packing rule to that which was in effect before the current setting.

Usage

The **#pragma pack** directive applies to the definition of an aggregate type, rather than to the declaration of an instance of that type; it therefore automatically applies to all variables declared of the specified type.

The **#pragma pack** directive modifies the current alignment rule for only the members of structures whose declarations follow the directive. It does not affect the alignment of the structure directly, but by affecting the alignment of the members of the structure, it may affect the alignment of the overall structure.

The **#pragma pack** directive cannot increase the alignment of a member, but rather can decrease the alignment. For example, for a member with data type of short, a **#pragma pack(1)** directive would cause that member to be packed in the structure on a 1-byte boundary, while a **#pragma pack(4)** directive would have no effect.

The **#pragma pack** directive applies only to complete declarations of structures or unions; this excludes forward declarations, in which member lists are not specified. For example, in the following code fragment, the alignment for struct S is 4, since this is the rule in effect when the member list is declared:

```
#pragma pack(1)
struct S;
#pragma pack(4)
struct S { int i, j, k; };
```

A nested structure has the alignment that precedes its declaration, not the alignment of the structure in which it is contained, as shown in the following example:

```
#pragma pack (4)                // 4-byte alignment
    struct nested {
        int x;
        char y;
        int z;
    };

    #pragma pack(1)              // 1-byte alignment
    struct packedcxx{           char a;
        short b;
        struct nested s1;      // 4-byte alignment
    };
```

If more than one **#pragma pack** directive appears in a structure defined in an inlined function, the **#pragma pack** directive in effect at the beginning of the structure takes precedence.

Examples

The following example shows how the **#pragma pack** directive can be used to set the alignment of a structure definition:

```
// header file file.h

#pragma pack(1)

struct jeff{                    // this structure is packed
    short bill;                 // along 1-byte boundaries
    int *chris;
};
#pragma pack(reset)            // reset to previous alignment rule
```

```
// source file anyfile.c

#include "file.h"

struct jeff j;           // uses the alignment specified
                        // by the pragma pack directive
                        // in the header file and is
                        // packed along 1-byte boundaries
```

This example shows how a **#pragma pack** directive can affect the size and mapping of a structure:

```
struct s_t {
  char a;
  int b;
  short c;
  int d;
}S;
```

Default mapping:

```
size of s_t = 16
offset of a = 0
offset of b = 4
offset of c = 8
offset of d = 12
alignment of a = 1
alignment of b = 4
alignment of c = 2
alignment of d = 4
```

With #pragma pack(1):

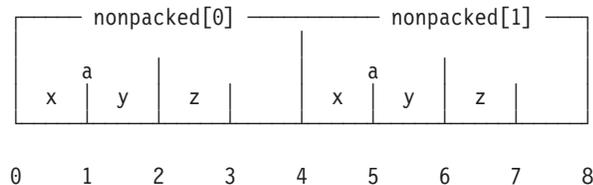
```
size of s_t = 11
offset of a = 0
offset of b = 1
offset of c = 5
offset of d = 7
alignment of a = 1
alignment of b = 1
alignment of c = 1
alignment of d = 1
```

The following example defines a union `uu` containing a structure as one of its members, and declares an array of 2 unions of type `uu`:

```
union uu {
  short a;
  struct {
    char x;
    char y;
    char z;
  } b;
};

union uu nonpacked[2];
```

Since the largest alignment requirement among the union members is that of short `a`, namely, 2 bytes, one byte of padding is added at the end of each union in the array to enforce this requirement:



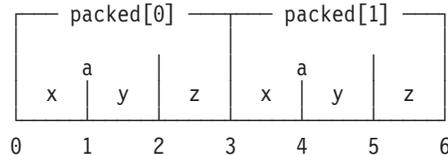
The next example uses **#pragma pack(1)** to set the alignment of unions of type `uu` to 1 byte:

```
#pragma pack(1)

union uu {
    short a;
    struct {
        char x;
        char y;
        char z;
    } b;
};

union uu pack_array[2];
```

Now, each union in the array packed has a length of only 3 bytes, as opposed to the 4 bytes of the previous case:



Related information

- “Compatibility of structures, unions, and enumerations (C only)” on page 64
- “The `_Packed` qualifier (C only)” on page 99

#pragma page (C only)

z/OS only

Category

Listings, messages and compiler information

Purpose

Specifies that the code following the pragma begins at the top of the page in the generated source listing.

Syntax

```
▶▶ #pragma page ( [number] ) ▶▶
```

Defaults

Not applicable.

Parameters

number

The number of pages from the current page on which to begin writing the line of source code that follows the pragma. **#pragma page()** is the same as **#pragma page(1)**: the source line that follows the pragma will start on a new page. If you specify **#pragma page(2)**, the listing will skip one blank page and the source line following the pragma will start on the second page after the current page. In all cases, the listing continues.

End of z/OS only

#pragma pagesize (C only)

z/OS only

Category

Listings, messages and compiler information

Purpose

Sets the number of lines per page for the generated source listing.

Syntax

```
▶▶ #pragma pagesize ( number ) ▶▶
```

Defaults

The default page size is 66 lines.

Parameters

number

The number of lines per page for the generated source listing.

Usage

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.

End of z/OS only

#pragma priority (C++ only)

Category

Object code control

Purpose

Specifies the priority level for the initialization of static objects.

The C++ standard requires that all global objects within the same translation unit be constructed from top to bottom, but it does not impose an ordering for objects declared in different translation units. The **#pragma priority** directive allows you to impose a construction order for all static objects declared within the same load module. Destructors for these objects are run in reverse order during termination.

Syntax

```
▶▶ #pragma priority ( number ) ▶▶
```

Defaults

The default priority level is 0.

Parameters

number

An integer literal in the range of -2 147 482 624 to 2147483647. A lower value indicates a higher priority; a higher value indicates a lower priority. Numbers from -214 783 648 to -214 782 623 are reserved for system use. If you do not specify a *number*, the compiler assumes .

Usage

More than one **#pragma priority** can be specified within a translation unit. The priority value specified in one pragma applies to the constructions of all global objects declared after this pragma and before the next one. However, in order to be consistent with the Standard, priority values specified within the same translation unit must be strictly increasing. Objects with the same priority value are constructed in declaration order.

The effect of a **#pragma priority** exists only within one load module. Therefore, **#pragma priority** cannot be used to control the construction order of objects in different load modules. Refer to *z/OS XL C/C++ Programming Guide* for further discussions on techniques used in handling DLL static object initialization.

#pragma prolog (C only), #pragma epilog (C only)

Category

Object code control

Purpose

When used with the METAL option, inserts High-Level Assembly (HLASM) prolog or epilog code for a specified function.

Prologs are inserted after function entry and epilogs are inserted before function return in the generated HLASM code. These directives allow you to provide your own function entry and exit code for system programming.

Syntax

```
▶▶ #pragma [epilog] (—function_name— ,—"text-string"—) ▶▶  
           └──prolog──┘
```

Defaults

Not applicable.

Parameters

function_name

The name of a function to which the epilog or prolog is to be inserted in the generated HLASM code.

text-string

text-string is a C string, which must contain valid HLASM statements. If the *text-string* consists of white-space characters only, or if the *text-string* is not provided, then the compiler ignores the option specification. If the *text-string* does not contain any white-space characters, then the compiler will insert leading spaces in front. Otherwise, the compiler will insert the *text-string* into the function prolog location of the generated assembler source. The compiler does not understand or validate the contents of the *text-string*. In order to satisfy the

assembly step later, the given *text-string* must form valid HLASM code with the surrounding code generated by the compiler.

Note: Special characters like newLine and quote are shell (or command line) meta characters, and maybe preprocessed before reaching the compiler. It is advisable to avoid using them.

Usage

These directives are only recognized when the z/OS XL C METAL compiler option is in effect.

Only one **#pragma epilog** or **#pragma prolog** directive is allowed for a specific function.

Related information

- EPILOG, PROLOG, and METAL options in the *z/OS XL C/C++ User's Guide*
- Default prolog and epilog code information in the *z/OS Metal C Programming Guide and Reference*

#pragma reachable

Category

Optimization and tuning

Purpose

Informs the compiler that the point in the program after a named function can be the target of a branch from some unknown location.

By informing the compiler that the instruction after the specified function can be reached from a point in your program other than the return statement in the named function, the pragma allows for additional opportunities for optimization.

Syntax

```
▶▶ #pragma reachable ( function_name ) ▶▶
```

Parameters

function_name

The name of a function preceding the instruction which is reachable from a point in the program other than the function's return statement.

Defaults

Not applicable.

Usage

Unlike the **#pragma leaves**, **#pragma reachable** is required by the compiler optimizer whenever the instruction following the call may receive control from some program point other than the return statement of the called function. If this condition is true and **#pragma reachable** is not specified, then the subprogram containing the call should not be compiled with OPT(1), OPT(2), OPT(3), or IPA. See also “#pragma leaves” on page 418.

If you specify the LIBANSI compiler option (which informs the compiler that function names that match functions in the C standard library are in fact C library functions), the compiler checks whether the setjmp family of functions (setjmp, _setjmp, sigsetjmp, and _sigsetjmp) contain **#pragma reachable**. If the functions do not contain this pragma directive, the compiler will insert this directive for the functions. This is not shown in the listing.

IPA considerations

If you specify the **#pragma reachable** directive in your source code in the IPA compile step, you cannot override the effects of this directive in the IPA link step.

If you specify the LIBANSI compile option for any translation unit in the IPA compile step, the compiler generates information which indicates the setjmp family of functions contain the **reachable** status. If you specify the NOLIBANSI option for the IPA link step, the attribute remains in effect.

Related information

- “#pragma leaves” on page 418

#pragma report (C++ only)

Category

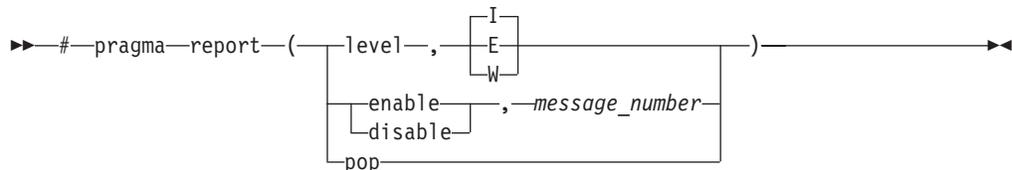
Listings, messages and compiler information

Purpose

Controls the generation of diagnostic messages.

The pragma allows you to specify a minimum severity level for a message for it to display, or allows you to enable or disable a specific message regardless of the prevailing report level.

Syntax



Defaults

The default report level is Informational (I), which displays messages of all types.

Parameters

level

Indicates that the pragma is set according to the minimum severity level of diagnostic messages to display.

E Indicates that only error messages will display. Error messages are of the highest severity. This is equivalent to the FLAG(E) compiler option.

W Indicates that warning and error messages will display. This is equivalent to the FLAG(W) compiler option.

I Indicates that all diagnostic messages will display: warning, error and informational messages. Informational messages are of the lowest severity. This is equivalent to the FLAG(I) compiler option.

enable

Enables the specified *message_number*.

disable

Disables the specified *message_number*.

message_number

Represents a message identifier, which consists of a prefix followed by the message number; for example, CCN1004.

pop

Reverts the report level to that which was previously in effect. If no previous report level has been specified, a warning is issued, and the report level remains unchanged.

Usage

The pragma takes precedence over most compiler options. For example, if you use **#pragma report** to disable a compiler message, that message will not be displayed with any FLAG compiler option setting. Similarly, if you specify the SUPPRESS compiler option for a message but also specify **#pragma report(enable)** for the same message, the pragma will prevail.

Related information

- The FLAG option in the *z/OS XL C/C++ User's Guide*

#pragma runopts

z/OS only

Category

Language element control

Purpose

Specifies a list of runtime options for the compiler to use at execution time.

Syntax

→ #pragma runopts (*suboption*) →

Defaults

Not applicable.

Parameters*suboption*

A z/OS Language Environment run-time option or any of the compiler run-time options ARGPARSE, ENV, PLIST, REDIR, or EXECOPS. For more information on z/OS run-time options, see the *z/OS XL C/C++ User's Guide* and *z/OS Language Environment Programming Guide*.

Usage

Specify your **#pragma runopts** directive in the translation unit that contains `main`. If more than one translation unit contains a **#pragma runopts** directive, unpredictable results can occur; the **#pragma runopts** directive only affects translation units containing `main`.

If a suboption to **#pragma runopts** is not a valid C or C++ token, you can surround the suboptions 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:*)
)
```

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 ARGPARSEINOARGPARSE, EXECOPSINOEXECOPS, PLIST, or REDIRINOREDİR 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**.
- 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 translation unit that it finds that contains CEEUOPT.
 - b. If you have not defined the CEEUOPT variable in any translation unit, the IPA link step uses the **#pragma runopts** value from the first translation unit that it processes.

The sequence of translation 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.

Related information

- “#pragma target (C only)” on page 447
- The TARGET option in the *z/OS XL C/C++ User's Guide*

End of z/OS only

#pragma sequence

z/OS only

Category

Language element control

Purpose

Defines 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.

Syntax

```
▶▶ #pragma {sequence (—left_column_margin—, —right_column_margin—) | nosequence} ▶▶
```

Defaults

C Sequence numbers are assigned to columns 73 through 80 of the input record for fixed-length-records, and no sequence numbers are assigned for variable-length records. **C++** No sequence numbers are assigned for fixed or variable-length records.

Parameters

left_column_margin

The column number of the first (left-hand) margin. The value of *left_column_margin* must be greater than 0, and less than 32767.

Also, *left_column_margin* must be less than or equal to the value of *right_column_margin*.

right_column_margin

The column number of the last (right-hand) margin. The value of *right_column_margin* must be greater than that of *left_column_margin*, and less than 32767.

You can also use an asterisk (*) to indicate 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.

Usage

C++ In C++ only, you can specify the **sequence** option with no parameters, which instructs the compiler to number columns 73 through 80 of the input record (fixed or variable length).

If you use the compiler options SEQUENCE or NOSEQUENCE with the **#pragma sequence/nosequence** directive, the directive overrides the compiler options. The compiler option is in effect up to the first **#pragma sequence/nosequence** directive. 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.

You can use **#pragma sequence** and **#pragma margins** together. If they reserve the same columns, **#pragma sequence** has priority, and the compiler 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 “#pragma margins” on page 425.

Related information

- The SEQUENCE option in the *z/OS XL C/C++ User's Guide*

End of z/OS only

#pragma skip (C only)

z/OS only

Category

Listings, messages and compiler information

Purpose

Skips lines of the generated source listing.

Syntax

#pragma skip directive syntax

```
▶▶ #pragma skip ( number ) ▶▶
```

Defaults

Not applicable.

Parameters

number

The number of lines to skip in the generated source listing. The value of *number* must be a positive integer less than 255. If you omit *number*, the compiler skips one line.

End of z/OS only

#pragma strings

Category

Object code control

Purpose

Specifies the storage type for string literals.

Syntax

```
▶▶ #pragma strings ( readonly  
                  writable  
                  writable ) ▶▶
```

Defaults

C and C++ strings are read-only.

Parameters

readonly

String literals are to be placed in read-only memory.

writable

String literals are to be placed in read-write memory.

Usage

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

The pragma must appear before any source statements in a file.

IPA effects

During the IPA link step, the compiler compares the **#pragma strings** specifications for individual translation units. If it finds differences, it treats the strings as if you specified **#pragma strings(writable)** for all translation units.

Related information

- The ROSTRINGS option in the *z/OS XL C/C++ User's Guide*

#pragma subtitle (C only)

z/OS only

Category

Listings, messages and compiler information

Purpose

Places subtitle text on all subsequent pages of the generated source listing.

Syntax

```
▶▶ #pragma subtitle ("text") ▶▶
```

Defaults

Not applicable.

Parameters

text

The subtitle to appear in on all pages of the generated source listing.

End of z/OS only

#pragma target (C only)

z/OS only

Category

Object code control

Purpose

Specifies the operating system or runtime environment for which the compiler creates the object module.

Syntax

►► #pragma target (LE
IMS) ►►

Defaults

LE

Parameters

LE

Generates code to run under the z/OS Language Environment run-time library. This is equivalent to specifying **#pragma target()**. This suboption 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.

IMS

Generates object code to run under IMS. This suboption 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.

Usage

Note that you cannot specify the release suboptions using the **#pragma target** directive as you can with the TARGET compiler option.

The only pragma directives that can precede **#pragma target** are **filetag**, **chars**, **langlvl**, and **longname**.

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 to the IPA link step.

If you specify different **#pragma target** directives for different translation units, the IPA link step uses the ENV and PLIST information from the translation unit containing `main`. If there is no `main`, it uses information from the first translation unit it finds. If you specify the TARGET compile option for the IPA link step, it overrules the **#pragma target** directive.

Related information

- “#pragma runopts” on page 443
- The TARGET option in the *z/OS XL C/C++ User's Guide*

#pragma title (C only)

z/OS only

Category

Listings, messages and compiler information

Purpose

Places title text on all subsequent pages of the generated source listing.

Syntax

```
▶▶ #pragma title ("text") ▶▶
```

Defaults

Not applicable.

Parameters

text

The title to appear in on all pages of the generated source listing.

End of z/OS only

#pragma unroll

Category

Optimization and tuning

Purpose

Controls loop unrolling, for improved performance.

When **unroll** is in effect, the optimizer determines and applies the best unrolling factor for each loop; in some cases, the loop control may be modified to avoid unnecessary branching. The compiler remains the final arbiter of whether the loop is actually unrolled.

Syntax

```
▶▶ #pragma {nounroll | unroll} [(-number-)] ▶▶
```

Defaults

See the description of the UNROLL option in the *z/OS XL C/C++ User's Guide*.

Parameters

number

Forces *number* – 1 replications of the designated loop body or full unrolling of the loop, whichever occurs first. The value of *number* is unbounded and must

be a positive integer. Specifying **#pragma unroll(1)** effectively disables loop unrolling, and is equivalent to specifying **#pragma nounroll**.

Usage

The pragma overrides the [NO]UNROLL compiler option setting for a designated loop. However, even if **#pragma unroll** is specified for a given loop, the compiler remains the final arbiter of whether the loop is actually unrolled.

Only one pragma may be specified on a loop. The pragma must appear immediately before the loop to have effect.

The pragma affects only the loop that follows it. An inner nested loop requires a **#pragma unroll** directive to precede it if the desired loop unrolling strategy is different from that of the prevailing [NO]UNROLL option.

The **#pragma unroll** and **#pragma nounroll** directives can only be used on for loops. They cannot be applied to do while and while loops.

The loop structure must meet the following conditions:

- There must be only one loop counter variable, one increment point for that variable, and one termination variable. These cannot be altered at any point in the loop nest.
- Loops cannot have multiple entry and exit points. The loop termination must be the only means to exit the loop.
- Dependencies in the loop must not be "backwards-looking". For example, a statement such as $A[i][j] = A[i - 1][j + 1] + 4$ must not appear within the loop.

Predefined macros

None.

Examples

In the following example, the **#pragma unroll(3)** directive on the first for loop requires the compiler to replicate the body of the loop three times. The **#pragma unroll** on the second for loop allows the compiler to decide whether to perform unrolling.

```
#pragma unroll(3)
for( i=0; i < n; i++)
{
    a[i] = b[i] * c[i];
}

#pragma unroll
for( j=0; j < n; j++)
{
    a[j] = b[j] * c[j];
}
```

In this example, the first **#pragma unroll(3)** directive results in:

```
i=0;
if (i>n-2) goto remainder;
for (; i<n-2; i+=3) {
    a[i]=b[i] * c[i];
    a[i+1]=b[i+1] * c[i+1];
    a[i+2]=b[i+2] * c[i+2];
}
if (i<n) {
```

```

remainder:
for (; i<n; i++) {
    a[i]=b[i] * c[i];
}
}

```

Related information

- The UNROLL option in the *z/OS XL C/C++ User's Guide*

#pragma variable

z/OS only

Category

Object code control

Purpose

Specifies whether the compiler is to use a named external object in a reentrant or non-reentrant fashion.

Syntax

```

▶▶ #pragma variable ( identifier , rent  
noent )

```

Defaults

▶ **C++** Variables are reentrant (**rent**). ▶ **C** Variables are not reentrant (**noent**).

Parameters

identifier

The name of an external variable.

rent

Specifies that the variable's references or definition will be in the writable static area that is in modifiable storage.

noent

Specifies that the variable's references or its definition is in the code area and is in potentially read-only storage. This suboption does not apply to, and has no effect on, program variables with `static` storage class.

Usage

If an identifier is defined in one translation unit and used in another, the reentrant or non-reentrant status of the variable must be the same in all translation units.

▶ **C** To specify that variables declared as `const` not be placed into the writeable static area, you must use the `ROCONST` and `RENT` compiler options.

If the specification for a `const` variable in a **#pragma variable** directive conflicts with the `ROCONST` option, the pragma directive takes precedence over the compiler option, and the compiler issues an informational message.

If you use the **noent** suboption 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.

The following code fragment leads to undefined behavior when compiled with the RENT option.

```
int i;
int *p = &i;
#pragma variable(p, norent)
```

The variable `i` is reentrant, but the pointer `p` is non-reentrant. If the code is in a DLL, there will only be one copy of `p` but multiple copies of `i`, one for each caller of the DLL.

A non-reentrant pointer variable cannot take an address as an initializer: the compiler will treat the variable as reentrant if necessary (in other words, it will ignore the pragma). Initializers for non-reentrant variables should be compile-time constants. Due to code relocation during execution time, an address in a program that has both reentrant and non-reentrant variables is never considered a compile-time constant. This restriction includes the addresses of string literals.

Examples

The following program fragment shows how to use the **#pragma variable** directive to force an external program variable to be part of a program that includes executable code and constant data.

```
#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** because **#pragma variable(rates, norent)** is specified. The writeable static area 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.

Related information

- The RENT option in the *z/OS XL C/C++ User's Guide*

End of z/OS only

#pragma wsizeof

z/OS only

Category

Portability and migration

Purpose

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 z/OS XL C/C++ feature.

When the `sizeof` operator was applied to a function return type, older z/OS C and C++ compilers returned the size of the widened type instead of the original type. For example, in the following code fragment, using the older compilers, `i` has a value of 4.

```
char foo();
i = sizeof foo();
```

Using the z/OS XL C/C++ compiler, `i` has a the value of 1, which is the size of the original type, `char`.

Syntax

```
▶▶ #pragma wsizeof ( [ on ] ) ▶▶
```

└── resume ─┘

Defaults

The `sizeof` operator returns the original type for function return types.

Parameters

on Enables the old compiler behavior of the `sizeof` operator, so that the widened size is returned for function return types.

resume

Re-enables the normal behavior of the `sizeof` operator.

Usage

You can use this 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.

The compiler will match **on** and **resume** throughout the entire compilation unit. That is, the effect of the pragma can extend beyond a header file. Ensure the **on** and **resume** pragmas are matched in your compilation unit.

The pragma only affects the use of `sizeof` on function return types. Other behaviors of `sizeof` remain the same.

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.

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 translation units with different **wsizeof** settings is merged together during the IPA link step.

Related information

- The **WSIZEOF** option in the *z/OS XL C/C++ User's Guide*

End of z/OS only

#pragma XOPTS

z/OS only

Category

Language element control

Purpose

Passes suboptions directly to the CICS integrated translator for processing CICS statements embedded in C/C++ source code.

Syntax

```
▶▶ #pragma XOPTS (—suboptions—) ▶▶
```

Defaults

Not applicable.

Parameters

XOPTS

Must be specified in all uppercase.

suboptions

Are options to be passed to the CICS integrated translator.

Usage

The directive is only valid when the CICS compiler option is in effect. It must appear before any C/C++ or CICS statements in the source, and must appear at file scope (C) or global namespace scope (C++).

Note that if you invoke the compiler with any of the preprocessing-only options, the directive will be preserved in the preprocessed output.

Related information

For detailed information on acceptable embedded CICS statements and preprocessed output, see the description of the CICS compiler option in the *z/OS XL C/C++ User's Guide*.

End of z/OS only

Chapter 19. Compiler predefined macros

Predefined macros can be used to conditionally compile code for specific compilers, specific versions of compilers, specific environments and/or specific language features.

Predefined macros fall into several categories:

- “General macros”
- “Macros related to the platform” on page 458
- “Macros related to compiler features” on page 459

“Examples of predefined macros” on page 467 show how you can use compiler macros in your code.

General macros

The following predefined macros are always predefined by the compiler. Unless noted otherwise, all the following macros are *protected*, which means that the compiler will issue a warning if you try to undefine or redefine them.

Table 39. General predefined macros

Predefined macro name	Description	Predefined value
__FUNCTION__	Indicates the name of the function currently being compiled.	A character string containing the name of the function currently being compiled.
 __LIBREL__	Indicates the Language Environment library level under which the compiler is running.	The return value of a compiler call to the <code>librel</code> library function.
 __ptr31__	Expands to the pointer qualifier <code>__ptr32</code> . Not protected.	__ptr32
 __PTR32	Indicates that the pointer qualifier <code>__ptr32</code> is recognized. Not protected.	1

Table 39. General predefined macros (continued)

Predefined macro name	Description	Predefined value
<p>z/OS <code>__TARGET_LIB__</code></p>	<p>Indicates the version of the target library.</p>	<p>A hexadecimal string literal representing the version number of the target library. The format of the version number is hex <i>PVRRMMMM</i>, where:</p> <p><i>P</i> Represents the z/OS XL C or C/C++ library product. The possible values are:</p> <ul style="list-style-type: none"> • 0 for C/370™ • 1 for Language Environment/370 and Language Environment for MVS & VM • 2 for OS/390 • 4 for z/OS Release 2 and later <p><i>V</i> Represents the version number</p> <p><i>RR</i> Represents the release number</p> <p><i>MMMM</i> Represents the modification number</p> <p>The value of the <code>__TARGET_LIB__</code> macro depends on the setting of the TARGET compiler option, which allows you to specify the run-time environment and release for the generated object module. The <code>__TARGET_LIB__</code> macro is set as follows:</p> <ul style="list-style-type: none"> • 0x41090000 (for zOSV1R9 TARGET suboption) • 0x41080000 (for zOSV1R8 TARGET suboption) • 0x41070000 (for zOSV1R7 TARGET suboption) • 0x41060000 (for zOSV1R6 TARGET suboption) • 0x41050000 (for zOSV1R5 TARGET suboption) • 0x41040000 (for zOSV1R4 TARGET suboption) • 0x41030000 (for zOSV1R3 TARGET suboption) • 0x41020000 (for zOSV1R2 TARGET suboption) • 0x41010000 (for zOSV1R1 TARGET suboption) • 0x220A0000 (for OSV2R10 TARGET suboption) • 0xnnnnnnnn (for 0xnnnnnnnn TARGET suboption) <p>If the TARGET suboption is specified as a hexadecimal string literal, this macro is also defined to that literal.</p>

Table 39. General predefined macros (continued)

Predefined macro name	Description	Predefined value
__TIMESTAMP__	<p>Indicates the date and time when the source file was last modified. The value changes as the compiler processes any include files that are part of your source program.</p> <p>This macro is available for Partitioned Data Sets (PDSs/PDSEs) and z/OS UNIX System Services 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 data sets, or in stream source in JCL, the compiler returns a dummy timestamp. For z/OS UNIX System Services files, the compiler uses the system timestamp on a source file. Otherwise, it returns a dummy timestamp, "Mon Jan 1 0:00:01 1990".</p>	<p>A character string literal in the form "<i>Day Mmm dd hh:mm:ss yyyy</i>", where::</p> <p><i>Day</i> Represents the day of the week (Mon, Tue, Wed, Thu, Fri, Sat, or Sun).</p> <p><i>Mmm</i> Represents the month in an abbreviated form (Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, or Dec).</p> <p><i>dd</i> Represents the day. If the day is less than 10, the first d is a blank character.</p> <p><i>hh</i> Represents the hour.</p> <p><i>mm</i> Represents the minutes.</p> <p><i>ss</i> Represents the seconds.</p> <p><i>yyyy</i> Represents the year.</p>

Macros indicating the z/OS XL C/C++ compiler product

Macros related to the z/OS XL C/C++ compiler are always predefined, and are protected (the compiler will issue a warning if you try to undefine or redefine them).

Table 40. Compiler product predefined macros

Predefined macro name	Description	Predefined value
 __COMPILER_VER__	Indicates the version of the compiler.	<p>A hexadecimal integer in the format <i>PVRRMMMM</i>, where :</p> <p><i>P</i> Represents the compiler product</p> <ul style="list-style-type: none"> • 0 for C/370 • 1 for AD/Cycle® C/370 and C/C++ for MVS/ESA • 2 for OS/390 C/C++ • 4 for z/OS Release 2 and later <p><i>V</i> Represents the version number</p> <p><i>RR</i> Represents the release number</p> <p><i>MMMM</i> Represents the modification number</p> <p>In z/OS Version 1 Release 9, the value of the macro is X'41090000'.</p>

Table 40. Compiler product predefined macros (continued)

Predefined macro name	Description	Predefined value
 <code>__IBMC__</code>	Indicates the level of the XL C compiler.	An integer in the format <i>PVRRM</i> , where : <i>P</i> Represents the compiler product <ul style="list-style-type: none"> • 0 for C/370 • 1 for AD/Cycle C/370 and C/C++ for MVS/ESA • 2 for OS/390 C/C++ • 4 for z/OS Release 2 and later <i>V</i> Represents the version number <i>RR</i> Represents the release number <i>M</i> Represents the modification number In z/OS XL C/C++ Version 1 Release 9, the value of the macro is 41090.
 <code>__IBMCP__</code>	Indicates the level of the XL C++ compiler.	An integer in the format <i>PVRRM</i> , where : <i>P</i> Represents the compiler product <ul style="list-style-type: none"> • 0 for C/370 • 1 for AD/Cycle C/370 and C/C++ for MVS/ESA • 2 for OS/390 C/C++ • 4 for z/OS Release 2 and later <i>V</i> Represents the version number <i>RR</i> Represents the release number <i>M</i> Represents the modification number In z/OS XL C/C++ Version 1 Release 9, the value of the macro is 41090.

Macros related to the platform

The following predefined macros are provided to facilitate porting applications between platforms. All platform-related predefined macros are unprotected and may be undefined or redefined without warning unless otherwise specified.

Table 41. Platform-related predefined macros

Predefined macro name	Description	Predefined value	Predefined under the following conditions
 <code>__370__</code>	Indicates that the program is compiled or targeted to run on System/370System/370.	1	Always predefined for z/OS.
 <code>__HHW_370__</code>	Indicates that the host hardware is System/370™.	1	Always predefined for z/OS.
 <code>__HOS_MVS__</code>	Indicates that the host operating system is z/OS.	1	Always predefined for z/OS.
 <code>__MVS__</code>	Indicates that the host operating system is z/OS.	1	Always predefined for z/OS.

Table 41. Platform-related predefined macros (continued)

Predefined macro name	Description	Predefined value	Predefined under the following conditions
 <code>__THW_370__</code>	Indicates that the target hardware is System/370.	1	Always predefined for z/OS.
 <code>__TOS_MVS__</code>	Indicates that the host operating system is z/OS.	1	Always predefined for z/OS.

Macros related to compiler features

Feature-related macros are predefined according to the setting of specific compiler options or pragmas. Unless noted otherwise, all feature-related macros are protected (the compiler will issue a warning if you try to undefine or redefine them).

Feature-related macros are discussed in the following sections:

- “Macros related to compiler option settings”
- “Macros related to language levels” on page 464

Macros related to compiler option settings

The following macros can be tested for various features, including source input characteristics, output file characteristics, optimization, and so on. All of these macros are predefined by a specific compiler option or suboption, or any invocation or pragma that implies that suboption. If the suboption enabling the feature is not in effect, then the macro is undefined.

See the description of each option in the *z/OS XL C/C++ User's Guide* for detailed information about the option.

Table 42. General option-related predefined macros

Predefined macro name	Description	Predefined value	Predefined when the following compiler option or equivalent pragma is in effect:
 <code>__ARCH__</code>	Indicates the target architecture for which the source code is being compiled.	The integer value specified in the ARCH compiler option.	ARCH(<i>integer value</i>)
 <code>__BFP__</code>	Indicates that binary floating point (BFP) mode is in effect.	1	FLOAT(IEEE)
<code>__64BIT__</code>	Indicates that 64-bit compilation mode is in effect.	1	LP64
<code>_CHAR_SIGNED</code>	Indicates that the default character type is signed char.	1	CHARS(SIGNED)
<code>_CHAR_UNSIGNED</code>	Indicates that the default character type is unsigned char.	1	CHARS(UNSIGNED)

Table 42. General option-related predefined macros (continued)

Predefined macro name	Description	Predefined value	Predefined when the following compiler option or equivalent pragma is in effect:
z/OS  __CHARSET_LIB		0	NOASCII
		1	ASCII
z/OS  __CICS__	Indicates that embedded CICS statements are accepted	1	CICS
z/OS  __CODESET__	Indicates the character code set in effect.	The compiler uses the following runtime function to determine the compile-time character code set: nl_langinfo(CODESET)	LOCALE
C++  _CPPUNWIND	Indicates that C++ exception handling is enabled.	1	EXH
__DIGRAPHS__	Indicates support for digraphs.	1	DIGRAPH
z/OS  __DLL__	Indicates that the program is compiled as DLL code.	1	C  DLL C++  Always predefined.
z/OS  __ENUM_OPT	Indicates that the compiler supports the ENUMSIZE option and the #pragma enum directive.	1	Always predefined.
z/OS  __FILETAG__	Indicates the character code set of the current file.	The string literal specified by the #pragma filetag directive. The value changes as the compiler processes include files that make up the source program.	#pragma filetag (<i>string literal</i>)
z/OS  C  __IBM_INLINE_ASM_SUPPORT	Indicates that inline assembly statements are supported.	1	Always predefined.
z/OS  C  __IBM_FAR_IS_SUPPORTED__	Indicates that the __far type qualifier is supported.	1	METAL
z/OS  __GOFF__		1	GOFF

Table 42. General option-related predefined macros (continued)

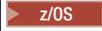
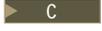
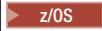
Predefined macro name	Description	Predefined value	Predefined when the following compiler option or equivalent pragma is in effect:
<code>__IBM_DFP__</code>	Indicates support for decimal floating-point types.	1	DFP
  <code>__IBM_METAL__</code>	Indicates that LE-independent HLASM code is to be generated by the compiler.	1	METAL
<code>__IGNERRNO__</code>	Indicates that system calls do not modify <code>errno</code> , thereby enabling certain compiler optimizations.	1	IGNERRNO
 <code>__ILP32</code>	Indicates that 32-bit compilation mode is in effect.	1	ILP32
<code>__INITAUTO__</code>	Indicates the value to which automatic variables which are not explicitly initialized in the source program are to be initialized.	A hexadecimal constant in the form <code>(0xnnU)</code> , including the parentheses, where <code>nn</code> represents the value specified in the <code>INITAUTO</code> compiler option.	<code>INITAUTO(hex value)</code>

Table 42. General option-related predefined macros (continued)

Predefined macro name	Description	Predefined value	Predefined when the following compiler option or equivalent pragma is in effect:
<code>__INITAUTO_W__</code>	Indicates the value to which automatic variables which are not explicitly initialized in the source program are to be initialized.	A hexadecimal constant in the form <code>(0xnnnnnnnnU)</code> , including the parentheses, where <code>nnnnnnnn</code> represents one of the following: <ul style="list-style-type: none"> If you specified an eight-digit ("word") value in the <code>INITAUTO</code> compiler option, <code>nnnnnnnn</code> is the value you specified. If you specified a two-digit ("byte") value in the <code>INITAUTO</code> compiler option, <code>nnnnnnnn</code> is the two-digit value repeated 4 times. 	<code>INITAUTO(hex value)</code>
 <code>_LARGE_FILES</code>	Indicates that large file support is enabled, which allows access to hierarchical file system files that are larger than 2 gigabytes.	1	<code>DEFINE(_LARGE_FILES)</code>
<code>__LIBANSI__</code>	Indicates that calls to functions whose names match those in the C Standard Library are in fact the C library functions, enabling certain compiler optimizations.	1	<code>LIBANSI</code>

Table 42. General option-related predefined macros (continued)

Predefined macro name	Description	Predefined value	Predefined when the following compiler option or equivalent pragma is in effect:
 <code>__LOCALE__</code>	Contains a string literal that represents the locale of the LOCALE compiler option. The following example shows how to use the <code>__LOCALE__</code> macro: <pre>int main() { #ifdef __LOCALE__ /* If the locale option is not specified, */ /* we can just follow the default locale */ setlocale(LC_ALL, __LOCALE__); #endif ... }</pre>	The compiler uses the following runtime function to determine the compile-time locale: <pre>setlocale (LC_ALL, "string literal");</pre>	<code>LOCALE(string literal)</code>
 <code>__LONGNAME__</code>	Indicates that identifiers longer than 8 characters are allowed.	1	 <code>LONGNAME</code>  Always predefined.
 <code>__LP64</code>	Indicates that 64-bit compilation mode is in effect.	1	<code>LP64</code>
 <code>__OBJECT_MODEL_CLASSIC__</code>	Indicates that the "classic" object model is in effect.	1	<code>OBJECTMODEL (CLASSIC)</code>
 <code>__OBJECT_MODEL_IBM__</code>	Indicates that the IBM object is in effect.	1	<code>OBJECTMODEL(ibm)</code>
<code>__OPTIMIZE__</code>	Indicates the level of optimization in effect.	The integer value specified in the OPT compiler option.	<code>OPT(integer value)</code>
 <code>__RTTI_DYNAMIC_CAST__</code>	Indicates that runtime type identification information for the <code>typeid</code> and <code>dynamic_cast</code> operator is generated.	1	<code>RTTI RTTI(ALL DYNAMICCAST)</code>
 <code>__SQL__</code>	Indicates that processing of embedded SQL statements is enabled.	1	<code>SQL</code>
 <code>__TEMPINC__</code>	Indicates that the compiler is using the template-implementation file method of resolving template functions.	1	<code>TEMPINC</code>

Table 42. General option-related predefined macros (continued)

Predefined macro name	Description	Predefined value	Predefined when the following compiler option or equivalent pragma is in effect:
 <code>__TUNE__</code>		The integer value specified in the TUNE compiler option.	TUNE(<i>integer value</i>)
 <code>__XPLINK__</code>		1	XPLINK

Macros related to language levels

The following macros can be tested for C99 features, features related to GNU C or C++, and other IBM language extensions. All of these macros are predefined to a value of 1 by a specific language level, represented by a suboption of the LANGLVL compiler option, or any invocation or pragma that implies that suboption. If the suboption enabling the feature is not in effect, then the macro is undefined. For descriptions of the features related to these macros, see the *z/OS XL C/C++ Language Reference*.

Table 43. Predefined macros for language features

Predefined macro name	Description	Predefined when the following language level is in effect
  <code>__ANSI__</code>	Indicates that only language constructs that support the ISO C89 Standard are allowed.	ansi stdc89 stdc99
 <code>__BOOL__</code>	Indicates that the <code>bool</code> keyword is accepted.	Always defined except when NOKEYWORD(<code>bool</code>) is in effect.
 <code>__C99_BOOL</code>	Indicates support for the <code>_Bool</code> data type.	stdc99 extc99 extc89 extended
 <code>__C99_COMPLEX</code>	Indicates support for complex data types.	stdc99 extc99 extc89 extended
 <code>__C99_CPLUSCMT</code>	Indicates support for C++ style comments	stdc99 extc99 (also SSCOM)
 <code>__C99_COMPOUND_LITERAL</code>	Indicates support for compound literals.	stdc99 extc99 extc89 extended
 <code>__C99_DESIGNATED_INITIALIZER</code>	Indicates support for designated initialization.	stdc99 extc99 extc89 extended
 <code>__C99_DUP_TYPE_QUALIFIER</code>	Indicates support for duplicated type qualifiers.	stdc99 extc99 extc89 extended
 <code>__C99_EMPTY_MACRO_ARGUMENTS</code>	Indicates support for empty macro arguments.	stdc99 extc99 extc89 extended
 <code>__C99_FLEXIBLE_ARRAY_MEMBER</code>	Indicates support for flexible array members.	stdc99 extc99 extc89 extended

Table 43. Predefined macros for language features (continued)

Predefined macro name	Description	Predefined when the following language level is in effect
<code>__C99_FUNC__</code>	Indicates support for the <code>__func__</code> predefined identifier.	 <code>stdc99</code> <code>extc99</code> <code>extc89</code> extended  <code>extended</code> <code>c99__func__</code>
 <code>__C99_HEX_FLOAT_CONST</code>	Indicates support for hexadecimal floating constants.	<code>stdc99</code> <code>extc99</code> <code>extc89</code> extended
 <code>__C99_INLINE</code>	Indicates support for the <code>inline</code> function specifier.	<code>stdc99</code> <code>extc99</code>
 <code>__C99_LLONG</code>	Indicates support for C99-style long long data types.	<code>stdc99</code> <code>extc99</code>
 <code>__C99_MACRO_WITH_VA_ARGS</code>	Indicates support for function-like macros with variable arguments.	<code>stdc99</code> <code>extc99</code> <code>extc89</code> extended
 <code>__C99_MAX_LINE_NUMBER</code>	Indicates that the maximum line number is 2147483647.	<code>stdc99</code> <code>extc99</code> <code>extc89</code> extended
 <code>__C99_MIXED_DECL_AND_CODE</code>	Indicates support for mixed declaration and code.	<code>stdc99</code> <code>extc99</code> <code>extc89</code> extended
 <code>__C99_MIXED_STRING_CONCAT</code>	Indicates support for concatenation of wide string and non-wide string literals.	<code>stdc99</code> <code>extc99</code> <code>extc89</code> extended
 <code>__C99_NON_LVALUE_ARRAY_SUB</code>	Indicates support for non-lvalue subscripts for arrays.	<code>stdc99</code> <code>extc99</code> <code>extc89</code> extended
 <code>__C99_NON_CONST_AGGR_INITIALIZER</code>	Indicates support for non-constant aggregate initializers.	<code>stdc99</code> <code>extc99</code> <code>extc89</code> extended
 <code>__C99_PRAGMA_OPERATOR</code>	Indicates support for the <code>_Pragma</code> operator.	<code>stdc99</code> <code>extc99</code> <code>extc89</code> extended
 <code>__C99_REQUIRE_FUNC_DECL</code>	Indicates that implicit function declaration is not supported.	<code>stdc99</code>
<code>__C99_RESTRICT</code>	Indicates support for the C99 restrict qualifier.	 <code>stdc99</code> <code>extc99</code>
 <code>__C99_STATIC_ARRAY_SIZE</code>	Indicates support for the <code>static</code> keyword in array parameters to functions.	<code>stdc99</code> <code>extc99</code> <code>extc89</code> extended
 <code>__C99_STD_PRAGMAS</code>	Indicates support for standard pragmas.	<code>stdc99</code> <code>extc99</code> <code>extc89</code> extended
 <code>__C99_TGMATH</code>	Indicates support for type-generic macros in <code>tgmath.h</code>	<code>stdc99</code> <code>extc99</code> <code>extc89</code> extended
<code>__C99_UCN</code>	Indicates support for universal character names.	 <code>stdc99</code> <code>extc99</code> <code>extc89</code> extended 
 <code>__C99_VAR_LEN_ARRAY</code>	Indicates support for variable length arrays.	<code>stdc99</code> <code>extc99</code> <code>extc89</code> extended

Table 43. Predefined macros for language features (continued)

Predefined macro name	Description	Predefined when the following language level is in effect
  <code>__COMMONC__</code>	Indicates that language constructs defined by XPG are allowed.	commonc
  <code>__COMPATMATH__</code>	Indicates that the newer C++ function declarations are not to be introduced by the math.h header file.	oldmath
 <code>__EXT</code>	Used in features.h to control the availability of extensions to the general ISO run-time libraries.	LIBEXT, or any LANGLVL suboption that implies it. (See the description of the LANGLVL option in the <i>z/OS XL C/C++ User's Guide</i> for a list of suboptions that imply LIBEXT.)
<code>__EXTENDED__</code>	Indicates that language extensions are supported.	extended
<code>__IBM_INCLUDE_NEXT</code>	Indicates support for the <code>#include_next</code> preprocessing directive.	Always defined.
<code>__IBM_TYPEOF__</code>	Indicates support for the <code>__typeof__</code> or <code>typeof</code> keyword.	 always defined  extended (Also KEYWORD(TYPEOF))
<code>__LONG_LONG</code>	Indicates support for IBM1ong 1ong data types.	 extended extc89 longlong  extended longlong
 <code>__MI_BUILTIN</code>	Indicates that the machine instruction built-in functions are available.	LIBEXT, or any LANGLVL suboption that implies it. (See the description of the LANGLVL option in the <i>z/OS XL C/C++ User's Guide</i> for a list of suboptions that imply LIBEXT.)
  <code>__RESTRICT__</code>	Indicates that the <code>__restrict__</code> or <code>__restrict</code> keywords are supported.	Predefined at all language levels.
 <code>__SAA__</code>	Indicates that only language constructs that support the most recent level of SAA C standards are allowed.	saa
 <code>__SAA_L2__</code>	Indicates that only language constructs that conform to SAA Level 2 C standards are allowed.	saal2

Examples of predefined macros

This example illustrates use of the `__FUNCTION__` and the `__C99_FUNC__` macros to test for the availability of the C99 `__func__` identifier to return the current function name:

```
#include <stdio.h>

#if defined(__C99_FUNC__)
#define PRINT_FUNC_NAME() printf (" In function %s \n", __func__);
#elif defined(__FUNCTION__)
#define PRINT_FUNC_NAME() printf (" In function %s \n", __FUNCTION__);
#else
#define PRINT_FUNC_NAME() printf (" Function name unavailable\n");
#endif

void foo(void);

int main(int argc, char **argv)
{
    int k = 1;
    PRINT_FUNC_NAME();
    foo();
    return 0;
}

void foo (void)
{
    PRINT_FUNC_NAME();
    return;
}
```

The output of this example is:

```
In function main
In function foo
```

 This example illustrates use of the `__FUNCTION__` macro in a C++ program with virtual functions.

CCNX08C

```
#include <stdio.h>
class X { public: virtual void func() = 0;};

class Y : public X {
    public: void func() { printf("In function %s \n", __FUNCTION__);}
};

int main() {
    Y aaa;
    aaa.func();
}
```

The output of this example is:

```
In function Y::func()
```

Appendix A. C and C++ compatibility on the z/OS platform

This appendix pertains to the differences between C and C++ that apply specifically to the z/OS platform. The contents describe the constructs that are found in both ISO C and ISO C++, but which are treated differently in the two languages.

String 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.

In 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 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).

Class/structure and typedef names

In C++, a class or structure 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 or structure 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:

```
int main ()
{
    typedef double db;
    struct db; /* error in C++, valid in C */

    typedef struct st st; /* valid C and C++ */
}
```

The same distinction applies within class/structure declarations. For example:

```
int main ()
{
    typedef double db;
    struct st
    {
        db x;
        double db; /* error in C++, valid in C */
    };
}
```

Class/structure and scope declarations

In C++, a class declaration introduces the class or structure name into the scope where it is declared and hides any object, function, or other declaration of that name in an outer scope. In 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;
int main ()
{
    struct db /* hides double object db in C++ */
```

```
    { char* str; };
    int x = sizeof(db);    /* size of struct in C++ */
                          /* size of double in C */
}
```

const object initialization

In C++, const objects must be initialized. In C, they can be left uninitialized.

Definitions

An object declaration is a definition in C++. In C, it is a declaration (also known as a tentative definition). For example:

```
int i;
```

In C++, a global data object must be defined only once. In 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 declarations for a variable.

Definitions within return or argument types

In C++, types may not be defined in return or argument types. C allows such definitions. For example, the following declarations produce errors in C++, but are valid declarations in C:

```
void print(struct X { int i;} x);    /* error in C++ */
enum count{one, two, three} counter(); /* error in C++ */
```

Enumerator type

An enumerator has the same type as its enumeration in C++. In C, an enumeration has type `int`.

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 C, an object of enumeration type can be assigned values of any integral type.

Function declarations

In C++, all declarations of a function must match the unique definition of a function. C has no such restriction.

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 C, it could take any number of arguments, of any type.

Global constant linkage

In C++, an object declared `const` has internal linkage, unless it has previously been given external linkage. In C, it has external linkage.

Jump statements

C++ does not allow you to jump over declarations containing initializations. C does allow you to use jump statements for this purpose.

Keywords

C++ contains some additional keywords not found in C. C programs that use these keywords as identifiers are not valid C++ programs:

Table 44. C++ keywords

<code>bool</code>	<code>export</code>	<code>private</code>	<code>true</code>
<code>catch</code>	<code>false</code>	<code>protected</code>	<code>try</code>
<code>class</code>	<code>friend</code>	<code>public</code>	<code>typeid</code>
<code>const_cast</code>	<code>inline</code>	<code>reinterpret_cast</code>	<code>typename</code>
<code>delete</code>	<code>mutable</code>	<code>static_cast</code>	<code>using</code>
<code>dynamic_cast</code>	<code>namespace</code>	<code>template</code>	<code>virtual</code>
<code>explicit</code>	<code>new</code>	<code>this</code>	<code>wchar_t</code>
	<code>operator</code>	<code>throw</code>	

main() recursion

In C++, `main()` cannot be called recursively and cannot have its address taken. C allows recursive calls and allows pointers to hold the address of `main()`.

Names of nested classes/structures

In C++, the name of a nested class is local to its enclosing class. In C, the name of the nested structure belongs to the same scope as the name of the outermost enclosing structure.

Pointers to void

C++ allows `void` pointers to be assigned only to other `void` pointers. In C, a pointer to `void` can be assigned to a pointer of any other type without an explicit cast.

Prototype declarations

C++ requires full prototype declarations. C allows nonprototyped functions.

Return without declared value

In both C and C++, the function `main()` must be declared to return a value of type `int`. In C++, if no value is explicitly returned from function `main()` by means of a return statement and if program execution reaches the end of function `main()` (that is, the program does not terminate due to a call to `exit()`, `std::terminate()`, or a similar function), then the value 0 is implicitly returned. A return (either explicit or implicit) from all other functions that are declared to return a value *must* return a value. In C, a function that is declared to return a value can return with no value, with unspecified results.

__STDC__ macro

The predefined macro variable `__STDC__` is defined for C++, and it has the integer value 0 when it is used in an `#if` statement, indicating that the C++ language is not a proper superset of C, and that the compiler does not conform to C. In C, `__STDC__` has the integer value 1.

Appendix B. Common Usage C language level for the z/OS Platform

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 ISO C language definition. It is based on various C implementations that predate the ISO standard.

Common Usage C is supported with the `LANGLVL(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:

```
"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 ISO token concatenation operator `##`.

The `LANGLVL` compiler option is described in *z/OS XL C/C++ User's Guide*. The `#pragma langlvl` is described in “`#pragma langlvl` directive (C only)” on page 417.

Appendix C. Conforming to POSIX 1003.1

The implementation resulting from the combination of z/OS UNIX System Services and the z/OS Language Environment supports 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 z/OS UNIX System Services implementation meets the criteria.

Appendix D. Implementation-defined behavior

The following sections describe how the z/OS XL C/C++ compilers define some of the implementation-specific behavior from the ISO C and C++ standards. In-depth usage information is provided in *z/OS XL C/C++ User's Guide* and *z/OS XL C/C++ Programming Guide*.

- “Identifiers”
- “Characters”
- “String conversion” on page 478
- “Integers” on page 478
- “Floating-point numbers” on page 479
- “C/C++ data mapping” on page 480
- “Arrays and pointers” on page 480
- “Registers” on page 480
- “Structures, unions, enumerations, bit fields” on page 481
- “Declarators” on page 481
- “Statements” on page 481
- “Preprocessing directives” on page 481
- “Translation limits” on page 482

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.

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 the `NOLONGNAME` option is used, 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 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 (|) 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).

The number of bits that represent a single-byte 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

Sequence of white-space characters (excluding the new-line):

- Any spaces or comments in the source program are 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

Type	Amount of storage	Range (in limits.h)
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

Type	Amount of storage	Range (in limits.h)
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	8 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

The result of converting an integer from 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 numbers

Type	Amount of storage	Range (approximate)	
		IBM z/Architecture hexadecimal format	IEEE binary format
float	4 bytes	5.5×10^{-79} to 7.2×10^{75}	1.2×10^{-38} to 3.4×10^{38}
double	8 bytes	5.5×10^{-79} to 7.2×10^{75}	2.2×10^{-308} to 1.8×10^{308}
long double	16 bytes	5.5×10^{-79} to 7.2×10^{75}	3.4×10^{-4932} to 1.2×10^{4932}

The following is the direction of truncation (or rounding) when you convert an integer number to an IBM z/Architecture hexadecimal floating-point number, or to an IEEE binary floating-point number:

- IBM z/Architecture 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 z/OS XL C/C++ compiler can evaluate at compile time. It has no effect on rounding at run time.

C/C++ data mapping

The z/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`, `_Bool/bool`, and `decimal(n,p)`
- Halfword boundary for `short int`
- Fullword boundary for `int`, `long int`, pointers, `float`, and `float _Complex`
- Doubleword boundary for `double`, `long double`, `double _Complex`, and `long double _Complex`

For each external defined variable, the z/OS XL 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 compile with the NORENT compiler option, in separate csects that are based on their names.

Arrays and pointers

The type of `size_t`:

- unsigned int in 32-bit mode
- unsigned long in 64-bit mode

The type of `ptrdiff_t`:

- int in 32-bit mode
- long in 64-bit mode

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:

- The register storage class indicates to the compiler that a variable in a block scope data definition or a parameter declaration is heavily used (such as a loop control variable). It is equivalent to `auto`, except that the compiler might, if possible, place the variable into a machine register for faster access.

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.

The padding at the end of structure/union:

- Padding is added to end the structure on its natural boundary. The alignment of the structure or union is that of its strictest member.

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

Value of a single-character constant in a constant expression that controls conditional inclusion:

- Matches the value of the character constant in the execution character set.

Such a constant may have a negative value:

- Yes

The method of searching include source files (`<...>`):

- See *z/OS XL C/C++ User's Guide*.

The method of searching quoted source files:

- User include files can be specified in double quotes. See *z/OS XL C/C++ User's Guide*.

The mapping between the name specified in the include directive and the external source file name:

- See *z/OS XL C/C++ User's Guide*.

The definitions of `__DATE__` and `__TIME__` when date and time of translation is not available:

- For *z/OS XL C/C++*, the date and time of translation are always available.

Translation limits

System-determined means that the limit is determined by your system resources.

Table 45. 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

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 | • 32760 under MVS |
| • Characters in a logical source line | • 32K minus 1 |
| • Characters in a string literal | • LONG_MAX (See 1) |
| • Bytes in an object | • SHRT_MAX |
| • Nested include files | • System-determined |
| • Enumeration constants in an enumeration | • System-determined |
| • Levels in nested structure or union | • System-determined |
-

Table 45. Translation Limits (continued)

Notes:

1. LONG_MAX is the limit for automatic variables only. For all other variables, the limit is 16 megabytes.
-

Appendix E. Accessibility

Accessibility features help a user who has a physical disability, such as restricted mobility or limited vision, to use software products successfully. The major accessibility features in z/OS enable users to:

- Use assistive technologies such as screen readers and screen magnifier software
- Operate specific or equivalent features using only the keyboard
- Customize display attributes such as color, contrast, and font size

Using assistive technologies

Assistive technology products, such as screen readers, function with the user interfaces found in z/OS. Consult the assistive technology documentation for specific information when using such products to access z/OS interfaces.

Keyboard navigation of the user interface

Users can access z/OS user interfaces using TSO/E or ISPF. Refer to *z/OS TSO/E Primer*, *z/OS TSO/E User's Guide*, and *z/OS ISPF User's Guide Vol I* for information about accessing TSO/E and ISPF interfaces. These guides describe how to use TSO/E and ISPF, including the use of keyboard shortcuts or function keys (PF keys). Each guide includes the default settings for the PF keys and explains how to modify their functions.

z/OS information

z/OS information is accessible using screen readers with the BookServer/Library Server versions of z/OS books in the Internet library at:

<http://www.ibm.com/servers/eserver/zseries/zos/bkserv/>

Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106-0032, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

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

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

This information contains 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.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Programming interface information

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

Trademarks

The following are trademarks or registered trademarks of International Business Machines Corporation in the United States or other countries or both:

IBM	IBM logo	ibm.com
AD/Cycle	AIX	BookManager
BookMaster	C/370	CICS
DB2	DB2 Universal Database	DFSMS
GDDM	Hiperspace	IMS
Language Environment	MVS	MVS/ESA
Open Class	OS/390	OS/400

QMF	REXX	SAA
System/370	System z	VSE/ESA
WebSphere	z/Architecture	z/OS
zSeries	z/VM	

Adobe, Acrobat, PostScript and all Adobe-based trademarks are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States and/or other countries.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

Other company, product, and service names may be trademarks or service marks of others.

Standards

The following standards are supported in combination with the Language Environment element:

- The C language is consistent with *Programming languages - C (ISO/IEC 9899:1999)*. For more information on ISO, visit their web site at: www.iso.org
- The C++ language is consistent with *Programming languages - C++ (ISO/IEC 14882:2003(E))* and *Programming languages - C++ (ISO/IEC 14882:1998)*.

The following standards are supported in combination with the Language Environment and z/OS UNIX System Services elements:

- A subset of *IEEE Std. 1003.1-2001 (Single UNIX Specification, Version 3)*. For more information on IEEE, visit their web site at: www.ieee.org.
- *IEEE Std 1003.1—1990, IEEE Standard Information Technology—Portable Operating System Interface (POSIX)—Part 1: System Application Program Interface (API) [C language]*, copyright 1990 by the Institute of Electrical and Electronic Engineers, Inc.
- The core features of *IEEE P1003.1a Draft 6 July 1991, Draft Revision to Information Technology—Portable Operating System Interface (POSIX), Part 1: System Application Program Interface (API) [C Language]*, copyright 1992 by the Institute of Electrical and Electronic Engineers, Inc.
- *IEEE Std 1003.2—1992, IEEE Standard Information Technology—Portable Operating System Interface (POSIX)—Part 2: Shells and Utilities*, copyright 1990 by the Institute of Electrical and Electronic Engineers, Inc.
- The core features of *IEEE Std P1003.4a/D6—1992, IEEE Draft Standard Information Technology—Portable Operating System Interface (POSIX)—Part 1: System Application Program Interface (API)—Amendment 2: Threads Extension [C language]*, copyright 1990 by the Institute of Electrical and Electronic Engineers, Inc.

- The core features of *IEEE 754-1985 (R1990) IEEE Standard for Binary Floating-Point Arithmetic (ANSI)*, copyright 1985 by the Institute of Electrical and Electronic Engineers, Inc.
- *X/Open CAE Specification, System Interfaces and Headers, Issue 4 Version 2*, copyright 1994 by The Open Group
- *X/Open CAE Specification, Networking Services, Issue 4*, copyright 1994 by The Open Group
- *X/Open Specification Programming Languages, Issue 3, Common Usage C*, copyright 1988, 1989, and 1992 by The Open Group
- United States Government's *Federal Information Processing Standard (FIPS) publication for the programming language C, FIPS-160*, issued by National Institute of Standards and Technology, 1991

Index

Special characters

`__callback` 69
`__cdecl` 194, 214
`__far` 70
`__func__` 18
`__ptr32` 72
`__VA_ARGS__` 375
`_Export` 100, 197
`_Pragma` 391
- (subtraction operator) 131
- (unary minus operator) 120
-- (decrement operator) 119
-> (arrow operator) 117
, (comma operator) 139
:: (scope resolution operator) 116
! (logical negation operator) 120
!= (not equal to operator) 133
?: (conditional operators) 141
/ (division operator) 130
/= (compound assignment operator) 129
. (dot operator) 117
\$ 31
* (indirection operator) 122
* (multiplication operator) 130
*= (compound assignment operator) 129
\ continuation character 28, 373
\ escape character 33
[] (array subscript operator) 138
% (remainder) 131
> (greater than operator) 132
>> (right-shift operator) 132
>>= (compound assignment operator) 129
>= (greater than or equal to operator) 132
< (less than operator) 132
<< (left-shift operator) 132
<<= (compound assignment operator) 129
<= (less than or equal to operator) 132
| (bitwise inclusive OR operator) 136
| (vertical bar), locale 31
|| (logical OR operator) 137
& (address operator) 121
& (bitwise AND operator) 135
& (reference declarator) 87
&& (logical AND operator) 136
&= (compound assignment operator) 129
preprocessor directive character 373
preprocessor operator 378
(macro concatenation) 379
+ (addition operator) 131
+ (unary plus operator) 120
++ (increment operator) 118
+= (compound assignment operator) 129
= (simple assignment operator) 128
== (equal to operator) 133
^ (bitwise exclusive OR operator) 135
^ (caret), locale 31
^= (compound assignment operator) 129

~ (bitwise negation operator) 120

A

aborting functions 369
abstract classes 294, 296
access rules
 base classes 279
 class types 241, 265
 friends 271
 members 265
 multiple access 286
 protected members 278
 virtual functions 296
access specifiers 252, 265, 275, 283
 in class derivations 279
accessibility 265, 286, 485
addition operator (+) 131
address operator (&) 121
aggregate types 39, 302
 initialization 92, 302
alias 87
 type-based aliasing 82
aliasing 406
 pragma disjoint 406
alignment 101, 435
 bit fields 58
 pragma pack 435
 structure members 56
 structures 101
allocation
 expressions 151
 functions 210
ambiguities
 base and derived member names 287
 base classes 285
 resolving 162, 289
 virtual function calls 294
AND operator, bitwise (&) 135
AND operator, logical (&&) 136
argc (argument count) 205
 example 205
 restrictions 206
arguments
 command-line 206
 default 211
 evaluation 213
 macro 375
 main function 205
 of catch blocks 359
 passing 183, 207
 restrictions 206
 passing by reference 209
 passing by value 208
 trailing 375
argv (argument vector) 205
 example 205
 restrictions 206

- arithmetic conversions 103
- arithmetic types
 - type compatibility 55
- armode
 - function attribute 75, 204
- arrays
 - array-to-pointer conversions 108
 - as function parameter 45, 202
 - declaration 45, 202, 252
 - description 84
 - flexible array member 56, 57
 - initialization 90
 - initializing 95
 - ISO support 480
 - multidimensional 85
 - subscripting operator 138
 - type compatibility 87
 - variable length 80, 86
 - zero-extent 56
- ASCII character codes 33
- asm 13
 - keyword 48
 - statements 178
- assembly
 - statements 178
- assignment operator (=)
 - compound 129
 - pointers 83
 - simple 128
- associativity of operators 156
- atexit function 368
- auto storage class specifier 44

B

- base classes
 - abstract 296
 - access rules 279
 - ambiguities 285, 287
 - direct 284
 - indirect 274, 285
 - initialization 304
 - multiple access 286
 - pointers to 277
 - virtual 285, 290
- base list 285
- best viable function 236
- binary expressions and operators 127
- binding xxxii, 98
 - direct 99
 - dynamic 291
 - static 291
 - virtual functions 291
- bit fields 57
 - as structure member 56
 - ISO support 481
 - type name 125
- bitwise negation operator (~) 120
- block statement 163
- block visibility 2
- BookManager documents xxii

- Boolean
 - conversions 104
 - data types 50
 - literals 21
- boundaries, data 480
- break statement 173
- built-in data types 39

C

- C language xxvii
- C++ language xxvii
- candidate functions 225, 236
- case label 166
- cast expressions 143
- catch blocks 353, 355
 - argument matching 359
 - order of catching 359
- CCSID (coded character set identifier) 402
- char type specifier 54
- character
 - data types 54
 - literals 26
 - multibyte 29, 32
- character set
 - extended 32
 - source 31
- checkout pragma 399
- CICS 454
- class libraries xxix
- class members
 - access operators 117
 - access rules 265
 - class member list 251
 - declaration 252
 - initialization 304
 - order of allocation 252
- class templates
 - declaration and definition 327
 - distinction from *template class* 326
 - explicit specialization 344
 - member functions 329
 - static data members 328
- classes 244
 - abstract 296
 - access rules 265
 - aggregate 242
 - base 275
 - base list 275
 - class objects 39
 - class specifiers 242
 - class templates 326
 - declarations 242
 - incomplete 246, 252
 - derived 275
 - friends 267
 - inheritance 273
 - keywords 241
 - local 248
 - member functions 253
 - member lists 251

- classes (*continued*)
 - member scope 255
 - nested 246, 269
 - overview 241
 - polymorphic 241
 - scope of names 245
 - static members 260
 - this pointer 257
 - using declaration 280
 - virtual 285, 291
- COBOL linkage 420
- comma 139
 - in enumerator list 62
- comment pragma 400
- comments 36
- common features of z/OS XL C and XL C++
 - compilers xxvii
- compatibility
 - data types 41
 - user-defined types 64
- compatible types
 - across source files 65
 - arithmetic types 55
 - arrays 87
 - in conditional expressions 141
- complex types 52
- composite types 41
 - across source files 65
- compound
 - assignment 129
 - expression 130
 - literal 151
 - statement 163
 - types 39
- concatenation
 - macros 379
 - multibyte characters 29
- conditional compilation directives 384
 - elif preprocessor directive 385
 - else preprocessor directive 387
 - endif preprocessor directive 387
 - examples 387
 - if preprocessor directive 385
 - ifdef preprocessor directive 386
 - ifndef preprocessor directive 386
- conditional expression (? :) 130, 141
- const 69
 - casting away constness 210
 - member functions 254
 - object 111
 - placement in type name 80
 - qualifier 67
 - vs. #define 374
- const_cast 148, 210
- constant expressions 61, 113
- constant initializers 251
- constants
 - fixed-point decimal 25
- constructors 301
 - converting 312, 314
 - copy 315

- constructors (*continued*)
 - exception handling 362
 - exception thrown in constructor 356
 - initialization
 - explicit 302
 - nontrivial 301, 309
 - overview 299
 - trivial 301, 309
- continuation character 28, 373
- continue statement 174
- conversion
 - constructors 312
 - function 314
 - implicit conversion sequences 237
- conversion sequence
 - ellipsis 238
 - implicit 237
 - standard 237
 - user-defined 238
- conversions
 - arithmetic 103
 - array-to-pointer 108
 - Boolean 104
 - cast 143
 - complex to real 105
 - explicit keyword 314
 - floating-point 104
 - function arguments 110
 - function-to-pointer 108
 - integral 104
 - lvalue-to-rvalue 107, 111, 237
 - packed decimal 105
 - pointer 107
 - pointer to derived class 290
 - pointer to member 257
 - qualification 109
 - references 109
 - standard 103
 - user-defined 311
 - void pointer 108
- convert pragma 402
- convlit pragma 403
- copy assignment operators 317
- copy constructors 315
- covariant virtual functions 293
- CPLUSPLUS macro 381
- csect pragma 404
- cv-qualifier 67, 78
 - in parameter type specification 226
 - syntax 67

D

- data mapping 480
- data members
 - description 252
 - scope 255
 - static 261
- data types
 - aggregates 39
 - Boolean 50

- data types *(continued)*
 - built-in 39
 - character 54
 - compatible 41
 - complex 52
 - composite 41
 - compound 39
 - enumerated 61
 - fixed-point decimal 53
 - floating 51
 - incomplete 40
 - integral 49
 - scalar 39
 - user-defined 39, 55
 - void 54
- DATE macro 380
- dbx xxx
- deallocation
 - expressions 155
 - functions 210
- Debug Tool xxxix
- debugging
 - Debug Tool xxxix
- decimal
 - floating constants 24
- decimal data type operators 126
- decimal integer literals 20
- declaration 183
- declarations
 - classes 242, 246
 - description 41
 - duplicate type qualifiers 68
 - friend specifier in member list 267
 - friends 271
 - pointers to members 256
 - resolving ambiguous statements 162
 - syntax 42, 79, 184
 - unsubscripted arrays 86
- declarative region 1
- declarators
 - description 77
 - reference 87
 - restrictions 481
- decrement operator (--) 119
- default
 - clause 166, 167
 - label 167
- default constructor 301
- define pragma 405
- define preprocessor directive 373
- defined unary operator 385
- definitions
 - description 41
 - macro 373
 - member function 253
 - packed union 99
 - tentative 43
- delete operator 155
- dependent names 348
- dereferencing operator 122
- derivation 275
- derivation *(continued)*
 - array type 84
 - public, protected, private 279
- derived classes
 - catch block 359
 - construction order 307
 - pointers to 277
- designated initializer
 - aggregate types 90
 - union 92
- designator 90
 - designation 90
 - designator list 90
 - union 92
- destructors 308
 - exception handling 362
 - exception thrown in destructor 356
 - overview 299
 - pseudo 310
- digitsof operator 126
- digraph characters 35
- direct base class 284
- disability 485
- disjoint pragma 406
- division operator (/) 130
- do statement 171
- dollar sign 31
- dot operator 117
- double type specifier 51
- downcast 149
- downward compatibility xxxi
- dynamic binding 291
- dynamic_cast 149

E

- EBCDIC character codes 34
- elaborated type specifier 245
- elif preprocessor directive 385
- ellipsis
 - conversion sequence 238
 - in function declaration 201
 - in function definition 201
 - in macro argument list 375
- else
 - preprocessor directive 387
 - statement 164
- enclosing class 253, 269
- endif preprocessor directive 387
- entry point
 - linkage 419
 - program 205
- enum
 - keyword 62
 - pragma 407
- enumerations 61
 - compatibility 64, 65
 - declaration 62
 - initialization 94
 - ISO support 481
 - trailing comma 62

- enumerator 62
- environment
 - pragma 409
- epilog pragma 440
- equal to operator (==) 133
- error preprocessor directive 388
- escape character \ 33
- escape sequence 33, 478
 - alarm \a 33
 - backslash \\ 33
 - backspace \b 33
 - carriage return \r 33
 - double quotation mark \" 33
 - form feed \f 33
 - horizontal tab \t 33
 - new-line \n 33
 - question mark \? 33
 - single quotation mark \' 33
 - vertical tab \v 33
- examples
 - ccnraa3 174
 - ccnraa4 175
 - ccnraa6 177
 - ccnraa7 170
 - ccnraa8 377
 - ccnraa9 377
 - ccnrab1 169
 - ccnrabc 388
 - ccnrabd 389
 - ccnx02j 10
 - ccnx02k 28
 - ccnx06a 209
 - ccnx06b 211
 - ccnx08c 467
 - ccnx10c 244
 - ccnx10d 244
 - ccnx11a 255
 - ccnx11c 258
 - ccnx11h 264
 - ccnx11i 267
 - ccnx11j 267
 - ccnx12b 228
 - ccnx13a 303
 - ccnx14a 276
 - ccnx14b 276
 - ccnx14c 277
 - ccnx14g 287
 - machine-readable xxii
 - naming of xxii
 - softcopy xxii
- exception handling 353, 430
 - argument matching 359
 - catch blocks 355
 - arguments 359
 - constructors 362
 - destructors 362
 - example, C++ 369
 - exception objects 353
 - function try blocks 353
 - handlers 353, 355
 - order of catching 359
- exception handling (*continued*)
 - rethrowing exceptions 361
 - set_terminate 369
 - set_unexpected 369
 - special functions 366
 - stack unwinding 362
 - terminate function 368
 - throw expressions 354, 361
 - try blocks 353
 - try exceptions 356
 - unexpected function 367
- exceptions
 - declaration 355
 - function try block handlers 356
 - specification 364
- exclusive OR operator, bitwise (^) 135
- explicit
 - instantiation, templates 340
 - keyword 312, 314
 - specializations, templates 341, 342
 - type conversions 143
- exponent 22
- export pragma 409
- expressions
 - allocation 151
 - assignment 128
 - binary 127
 - cast 143
 - comma 139
 - conditional 141
 - deallocation 155
 - description 111
 - integer constant 113
 - new initializer 154
 - parenthesized 115
 - pointer to member 141
 - primary 112
 - resolving ambiguous statements 162
 - statement 162
 - throw 156, 361
 - unary 118
- extension pragma 410
- extern storage class specifier 8, 11, 46, 188
 - with function pointers 214
 - with variable length arrays 86

F

- FETCHABLE preprocessor directive 420
- file inclusion 382, 383
- FILE macro 380
- file scope data declarations
 - unsubscripted arrays 86
- filetag pragma 411
- fixed-point decimal
 - constants 25
 - data type 53
- flexible array member 57
- float type specifier 51
- floating-point
 - constant 23, 24

- floating-point (*continued*)
 - conversions 104
 - literal 21
 - promotion 106
 - range 479
 - storage 479
- floating-point types 51
- for statement 171
- FORTRAN linkage 420
- free store
 - delete operator 155
 - new operator 151
- friend
 - access rules 271
 - implicit conversion of pointers 280
 - member functions 253
 - nested classes 269
 - relationships with classes when templates are involved 329
 - scope 269
 - specifier 267
 - virtual functions 294
- function attributes 203
- function designator 111
- function specifier
 - explicit 312, 314
- function templates
 - explicit specialization 344
- function try blocks 353
 - handlers 356
- function-like macro 375
- functions 183
 - allocation 210
 - arguments 183, 208
 - conversions 110
 - block 183
 - body 183
 - calling 207
 - calls 116
 - as lvalue 112
 - class templates 329
 - conversion function 314
 - deallocation 210
 - declaration 183
 - C++ 254
 - examples 185
 - multiple 187
 - parameter names 202
 - default arguments 211
 - evaluation 213
 - restrictions 212
 - definition 183
 - examples 186
 - exception handling 366
 - exception specification 364
 - friends 267
 - function call operator 183
 - function templates 330
 - function-to-pointer conversions 108
 - inline 190, 253
 - library functions 183

- functions (*continued*)
 - main 205
 - name 183
 - diagnostic 18
 - overloading 225
 - parameters 207, 208
 - pointers to 214
 - polymorphic 274
 - predefined identifier 18
 - prototype 183
 - return statements 175
 - return type 183, 198, 199
 - return value 183, 199
 - signature 200
 - specifiable attributes 203
 - specifiers 190
 - template function
 - template argument deduction 331
 - type name 80
 - virtual 254, 291, 294

G

- global register variables 48
- global variable 3, 8
 - uninitialized 89
- goto statement 177
 - restrictions 177
- greater than operator (>) 132
- greater than or equal to operator (>=) 132

H

- handlers 355
- hexadecimal
 - floating constants 23
- hexadecimal integer literals 21
- hidden names 243, 245

I

- I/O interfaces xxxvi
- identifiers 15, 113
 - case sensitivity 16
 - id-expression 78, 114
 - ISO support 477
 - labels 161
 - linkage 8
 - namespace 5
 - predefined 18
 - reserved 13, 14, 17
 - special characters 16, 31
 - truncation 17
- if
 - preprocessor directive 385
 - statement 164
- ifdef preprocessor directive 386
- ifndef preprocessor directive 386
- implementation pragma 412
- implementation-defined behavior 477
- implicit conversion 103

- implicit conversion (*continued*)
 - Boolean 104
 - floating-point 104
 - integral 104
 - lvalue 111
 - packed decimal 105
 - pointer to derived class 277, 280
 - pointers to base class 278
 - types 103
- implicit conversions
 - complex to real 105
- implicit instantiation
 - templates 339
- include preprocessor directive 382
- include_next preprocessor directive 383
- inclusive OR operator, bitwise (|) 136
- incomplete type 84
 - as structure member 56, 57
 - class declaration 246
- incomplete types 40
- increment operator (++) 118
- indentation of code 373
- indirect base class 274, 285
- indirection operator (*) 122
- info pragma 413
- information hiding 1, 2, 251, 278
- inheritance
 - multiple 274, 284
 - overview 273
- initialization
 - aggregate types 92
 - auto object 89
 - base classes 304
 - class members 304
 - extern object 89
 - references 109
 - register object 90
 - static data members 263
 - static object 89
 - union member 94
- initializer lists 88, 304
- initializers 88
 - aggregate types 90, 92
 - enumerations 94
 - unions 94
- inline
 - assembly statements 178
 - function specifier 190
 - functions 190, 253
 - pragma 414
- input and output xxxvi
- input record 445
- integer
 - constant expressions 61, 113
 - data types 49
 - ISO support 478
 - literals 19
 - promotion 106
- integral
 - conversions 104
- interaction with other IBM products xxxviii

K

- keyboard 485
- keywords 13
 - __cdecl 194
 - _Export 100, 197
 - description 17
 - exception handling 353
 - language extension 14
 - template 319, 349, 350
 - underscore characters 14

L

- label
 - implicit declaration 3
 - in switch statement 166
 - statement 161
- langlvl pragma 417
- Language Environment xxx
- language extension 14
- language level 410, 417, 430
- leaves pragma 418
- left-shift operator (<<) 132
- less than operator (<) 132
- less than or equal to operator (<=) 132
- limits
 - floating-point 479
 - integer 478
- LINE macro 380
- line preprocessor directive 389
- linkage 1, 7
 - auto storage class specifier 44
 - COBOL 420
 - const cv-qualifier 69
 - extern storage class specifier 11, 46
 - external 8
 - FORTRAN 420
 - in function definition 188
 - inline member functions 254
 - internal 7, 44, 188
 - language 9, 419
 - multiple function declarations 187
 - none 9
 - PL/I 420
 - pragma 419
 - register storage class specifier 48
 - specifications 9
 - static storage class specifier 45
 - with function pointers 214
- linking xxxii
- linking to non-C++ programs 9
- literals 19
 - Boolean 21
 - character 26
 - compound 151
 - floating-point 21
 - integer 19
 - decimal 20
 - hexadecimal 21
 - octal 21

- literals (*continued*)
 - string 27
- local
 - classes 248
 - type names 249
- logical operators
 - ! (logical negation) 120
 - || (logical OR) 137
 - && (logical AND) 136
- long double type specifier 51
- long long type specifier 49
- long type specifier 49
- LONGNAME compiler option 16
- longname pragma 422
- LookAt message retrieval tool xxiii
- loop optimization 449
 - qunroll compiler option 449
- lvalues 67, 111, 113
 - casting 143
 - conversions 107, 111, 237

M

- macro
 - definition 373
 - typeof operator 126
 - function-like 375
 - invocation 375
 - object-like 374
 - variable argument 375
- macros 455
 - related to compiler options 459
 - related to language features 464
 - related to the compiler 457
 - related to the platform 458
- main function 205
 - arguments 205
 - example 205
- margins pragma 425
- member functions
 - const and volatile 254
 - definition 253
 - friend 253
 - special 254
 - static 263
 - this pointer 257, 295
- member lists 242, 251
- members
 - access 265
 - access control 283
 - class member access operators 117
 - data 252
 - pointers to 141, 256
 - protected 278
 - scope 255
 - static 247, 260
 - virtual functions 254
- memory
 - data mapping 480
- message retrieval tool, LookAt xxiii
- modifiable lvalue 111, 128

- modulo operator (%) 131
- multibyte character 32
 - concatenation 29
 - ISO support 477
 - overview 477
- multicharacter literal 26
- multidimensional arrays 85
- multiple
 - access 286
 - inheritance 274, 284
- multiplication operator (*) 130
- mutable storage class specifier 47

N

- name binding 348
- name hiding 6, 116
 - accessible base class 290
 - ambiguities 288
- name mangling
 - function 195
 - pragma 426, 427
 - scheme 429
- names
 - conflicts 5
 - hidden 116, 243, 245
 - local type 249
 - long name support 17
 - mangling 10
 - resolution 2, 280, 289
- namespace
 - class names 245
 - context 6
 - of identifiers 5
- namespaces 217
 - alias 217, 218
 - declaring 217
 - defining 217
 - explicit access 223
 - extending 218
 - friends 221
 - member definitions 221
 - namespace scope object
 - exception thrown in constructor 356
 - overloading 219
 - unnamed 219
 - user-defined 3
 - using declaration 222
 - using directive 222
- narrow character literal 26
- nested classes
 - friend scope 269
 - scope 246
- nesting level limits 482
- new operator
 - default arguments 212
 - description 151
 - initializer expression 154
 - placement syntax 153
 - set_new_handler function 154
- noinline pragma 414

- NOLONGNAME compiler option 16
- nolongname pragma 422
- nomargins pragma 425
- nosequence pragma 445
- not equal to operator (!=) 133
- Notices 487
- null
 - character \0 28
 - pointer 95
 - pointer constants 108
 - preprocessor directive 390
 - statement 178
- number sign (#)
 - preprocessor directive character 373
 - preprocessor operator 378

O

- object_model pragma 429
- object-like macro 374
- objects 111
 - class
 - declarations 242
 - description 39
 - lifetime 1
 - namespace scope
 - exception thrown in constructor 356
 - restrict-qualified pointer 73
 - static
 - exception thrown in destructor 356
- octal integer literals 21
- one's complement operator (~) 120
- operator functions 227
- operator_new pragma 430
- operators 29
 - (subtraction) 131
 - (decrement) 119
 - > (arrow) 117
 - >* (pointer to member) 141
 - , (comma) 139
 - :: (scope resolution) 116
 - ! (logical negation) 120
 - != (not equal to) 133
 - ? : (conditional) 141
 - / (division) 130
 - . (dot) 117
 - * (pointer to member) 141
 - () (function call) 116, 183
 - * (indirection) 122
 - * (multiplication) 130
 - (unary minus) 120
 - [] (array subscripting) 138
 - % (remainder) 131
 - > (greater than) 132
 - >> (right- shift) 132
 - >= (greater than or equal to) 132
 - < (less than) 132
 - << (left- shift) 132
 - <= (less than or equal to) 132
 - | (bitwise inclusive OR) 136
 - || (logical OR) 137

- operators (*continued*)
 - & (address) 121
 - & (bitwise AND) 135
 - && (logical AND) 136
 - + (addition) 131
 - ++ (increment) 118
 - = (simple assignment) 128
 - == (equal to) 133
 - ^ (bitwise exclusive OR) 135
 - alternative representations 30
 - assignment 128
 - copy assignment 317
 - associativity 156
 - binary 127
 - bitwise negation operator (~) 120
 - compound assignment 129
 - const_cast 148
 - defined 385
 - delete 155
 - digitsof 126
 - dynamic_cast 149
 - equality 133
 - new 151
 - overloading 227, 253
 - binary 231
 - unary 229
 - pointer to member 141, 257
 - precedence 156
 - examples 159
 - type names 80
 - precisionof 126
 - preprocessor
 - # 378
 - ## 379
 - pragma 391
 - reinterpret_cast 146
 - relational 132
 - scope resolution 276, 287, 294
 - sizeof 123
 - static_cast 145
 - typeid 122
 - typeof 125
 - unary 118
 - unary plus operator (+) 120
- optimization
 - controlling, using option_override pragma 431
 - inlining 414
- option_override pragma 431
- OR operator, logical (||) 137
- OS linkage 420
- overload resolution 236, 290
 - resolving addresses of overloaded functions 238
- overloading
 - description 225
 - function templates 337
 - functions 225, 281
 - restrictions 226
 - operators 227, 241
 - assignment 232
 - binary 231
 - class member access 235

- overloading (*continued*)
 - operators (*continued*)
 - decrement 230
 - function call 233
 - increment 230
 - subscripting 234
 - unary 229
- overriding virtual functions 295
 - covariant virtual function 293

P

- pack pragma 435
- packed
 - assignments and comparisons 129
 - structure member 57
 - structures 65
 - unions 65, 99
- packed decimal
 - conversions 105
- Packed qualifier 99
- page pragma 438
- pagesize pragma 439
- parenthesized expressions 79, 115
- pass by reference 87, 209
- pass by value 208
- PDF documents xxii
- PL/I linkage 420
- placement syntax 153
- pointer to member
 - conversions 257
 - declarations 256
 - operators 141, 257
- pointers
 - arrays 480
 - conversions 107, 108, 290
 - cv-qualified 81
 - dereferencing 82
 - description 80
 - generic 108
 - null 95
 - pointer arithmetic 81
 - restrict-qualified 73
 - this 257
 - to functions 214
 - to members 141, 256
 - type-qualified 81
 - void* 107
- polymorphism
 - polymorphic classes 241, 292
 - polymorphic functions 274
- portability issues 477
- POSIX 475
- postfix
 - ++ and -- 118, 119
- pound sign (#)
 - preprocessor directive character 373
 - preprocessor operator 378
- pragma operator 391
- pragmas
 - _Pragma 391
- pragmas (*continued*)
 - checkout 399
 - comment 400
 - convert 402
 - convlit 403
 - csect 404
 - define 405
 - disjoint 406
 - enum 407
 - environment 409
 - epilog 440
 - export 409
 - extension 410
 - filetag 411
 - implementation 412
 - info 413
 - inline 414
 - IPA considerations 394
 - langlvl 417
 - leaves 418
 - linkage 419
 - longname 422
 - margins 425
 - namemangling 426
 - namemanglingrule 427
 - noinline 414
 - nolongname 422
 - nomargins 425
 - nosequence 445
 - object_model 429
 - operator_new 430
 - option_override 431
 - options 433
 - pack 435
 - page 438
 - pagesize 439
 - preprocessor directive 390
 - priority 439
 - prolog 440
 - reachable 441
 - report 442
 - runopts 443
 - sequence 445
 - skip 446
 - standard 391
 - subtitle 447
 - target 448
 - title 449
 - variable 451
 - wsizeof 452
 - XOPTS 454
- precedence of operators 156
- precisionof operator 126
- predefined identifier 18
- predefined macros
 - CPLUSPLUS 381
 - DATE 380
 - FILE 380
 - LINE 380
 - STDC 380
 - STDC_HOSTED 381

- predefined macros (*continued*)
 - STDC_VERSION 381
 - TARGET_LIB 456
 - TIME 381
- prefix
 - ++ and -- 118, 119
 - decimal floating constants 24
 - hexadecimal floating constants 23
 - hexadecimal integer literals 21
 - octal integer literals 21
- prelinking xxxii
- preprocessor directives 373
 - conditional compilation 384
 - ISO support 481
 - preprocessing overview 373
 - special character 373
- preprocessor operator
 - _Pragma 391
 - # 378
 - ## 379
- primary expressions 112
- priority pragma 439
- program management binder xxxiii
- prolog pragma 440
- promotions
 - integral and floating-point 106
- pseudo-destructors 310
- punctuators 29
 - alternative representations 30
- pure specifier 252, 254, 294, 296
- pure virtual functions 296

Q

- qualification conversions 109
- qualified name 116, 247
- qualifiers
 - __callback 69
 - __far 70
 - __ptr32 72
 - _Packed 99
 - const 67
 - in parameter type specification 226
 - restrict 73
 - volatile 67, 74

R

- record
 - margins 425
 - sequence numbers 445
- reentrant variables 451
- references
 - as return types 199
 - binding 98
 - conversions 109
 - declarator 121
 - description 87
 - initialization 98
- register storage class specifier 47
- register variables 48

- registers
 - ISO support 480
- reinterpret_cast 146
- release changes xxv
- remainder operator (%) 131
- report
 - pragma 442
- restrict 73
 - in parameter type specification 226
- return statement 175, 199
- return type
 - reference as 199
 - size_t 123
- right-shift operator (>>) 132
- RTTI support 122
- run-time library file types xxxvii
- runopts pragma 443
- runtime options 443
- rvalues 111

S

- scalar types 39, 80
- scope 1
 - class 4
 - class names 245
 - description 1
 - enclosing and nested 2
 - friends 269
 - function 3
 - function prototype 3
 - global 3
 - global namespace 3
 - identifiers 5
 - local (block) 2
 - local classes 248
 - macro names 378
 - member 255
 - nested classes 246
- scope resolution operator
 - ambiguous base classes 287
 - description 116
 - inheritance 276
 - virtual functions 294
- sequence point 140
- sequence pragma 445
- set_new_handler function 154
- set_terminate function 369
- set_unexpected function 367, 369
- shift operators << and >> 132
- shift states 477
- short type specifier 49
- shortcut keys 485
- side effect 74
- signed type specifiers
 - char 54
 - int 49
 - long 49
 - long long 49
- size_t 123
- sizeof operator 123

- sizeof operator (*continued*)
 - with variable length arrays 86
- skip pragma 446
- source
 - program
 - margins 425
- space character 373
- special characters 31
- special member functions 254
- specifiers
 - access control 279
 - inline 190
 - pure 254
 - storage class 43
- splice preprocessor directive ## 379
- stack unwinding 362
- Standard C 477
- Standard C++ 477
- standard type conversions 103
- statements 161
 - block 163
 - break 173
 - continue 174
 - do 171
 - expressions 162
 - for 171
 - goto 177
 - if 164
 - labels 161
 - null 178
 - resolving ambiguities 162
 - restriction 481
 - return 175, 199
 - selection 164, 166
 - switch 166
 - while 170
- static
 - binding 291
 - data members 261
 - in array declaration 45, 202
 - initialization of data members 263
 - member functions 263
 - members 247, 260
 - storage class specifier 44, 188
 - linkage 45
 - with variable length arrays 86
- static storage class specifier 8
- static_cast 145
- STDC macro 380
- STDC_HOSTED macro 381
- STDC_VERSION macro 381
- storage class specifiers 43
 - auto 44
 - extern 46, 188
 - mutable 47
 - register 47
 - static 44, 188
- storage duration 1
 - auto storage class specifier 44
 - extern storage class specifier 46
 - register storage class specifier 48

- storage duration (*continued*)
 - static 44, 188
 - exception thrown in destructor 356
- storage of variables 480
- string
 - conversion 478
 - literal 27
 - terminator 28
- stringize preprocessor directive # 378
- struct type specifier 56
- structures 55, 244
 - as base class 280
 - as class type 241, 242
 - compatibility 64, 65
 - flexible array member 56, 57
 - identifier (tag) 56
 - initialization 92
 - ISO support 481
 - members 56
 - alignment 56
 - incomplete types 57
 - layout in memory 56, 92
 - packed 57
 - padding 56
 - zero-extent array 56
 - namespaces within 6
 - packed 56
 - unnamed members 92
- subscript declarator
 - in arrays 85
- subscripting operator 84, 138
 - in type name 80
- subtitle pragma 447
- subtraction operator (-) 131
- suffix
 - decimal floating constants 24
 - floating-point literals 21
 - hexadecimal floating constants 23
 - integer literal constants 19
- switch statement 166
- System Programming C facility xxxviii

T

- tags
 - enumeration 61, 62
 - structure 56
 - union 56
- target pragma 448
- TARGET_LIB macro 456
- technical support xxiv
- template arguments 322
 - deduction 331
 - deduction, non-type 336
 - deduction, type 334
 - non-type 323
 - template 325
 - type 322
- template keyword 350

- templates
 - arguments
 - non-type 323
 - type 322
 - class
 - declaration and definition 327
 - distinction from *template class* 326
 - explicit specialization 344
 - member functions 329
 - static data members 328
 - declaration 319
 - dependent names 348
 - explicit specializations 341, 343
 - class members 343
 - declaration 341
 - definition and declaration 342
 - function templates 344
 - function
 - argument deduction 336
 - overloading 337
 - partial ordering 337
 - function templates 330
 - type template argument deduction 334
 - instantiation 319, 338, 341
 - explicit 340
 - implicit 339
 - name binding 348
 - parameters 320
 - default arguments 321
 - non-type 320
 - template 321
 - type 320
 - partial specialization 346
 - matching 348
 - parameter and argument lists 347
 - point of definition 349
 - point of instantiation 349
 - pragma define 405
 - pragma implementation 412
 - relationship between classes and their friends 329
 - scope 343
 - specialization 319, 338, 341
- temporary objects 359
- tentative definition 43
- terminate function 353, 354, 359, 362, 366, 368
 - set_terminate 369
- this pointer 257, 295
- throw expressions 156, 353, 361
 - argument matching 359
 - rethrowing exceptions 361
 - within nested try blocks 354
- TIME macro 381
- title pragma 449
- tokens 13, 373
 - alternative representations for operators and punctuators 30
- translation limits 482
- translation unit 1
- trigraph sequences 35
- truncation
 - integer division 130
- try blocks 353
 - nested 354
- try keyword 353
- type attributes 74
- type name 79
 - local 249
 - qualified 116, 247
 - typename keyword 349
 - typeof operator 125
- type qualifiers
 - __callback 69
 - __far 70
 - __ptr32 72
 - _Packed 99
 - const 67, 69
 - const and volatile 78
 - duplicate 68
 - in function parameters 226
 - restrict 67, 73
 - volatile 67
- type specifier
 - class types 242
 - elaborated 245
- type specifiers 49
 - _Bool 50
 - bool 50
 - char 54
 - complex 52
 - double 51
 - enumeration 61
 - fixed-point decimal 53
 - float 51
 - int 49
 - long 49
 - long double 51
 - long long 49
 - short 49
 - unsigned 49
 - void 54
 - wchar_t 49, 54
- typedef specifier 65
 - class declaration 249
 - local type names 249
 - pointers to members 257
 - qualified type name 247
 - with variable length arrays 86
- typeid operator 122
- typename keyword 349
- typeof operator 125
- types
 - class 242
 - conversions 143
 - type-based aliasing 82
 - variably modified 85

U

- unary expressions 118
- unary operators 118
 - minus (-) 120
 - plus (+) 120

- undef preprocessor directive 378
- underscore character 14, 17
 - in identifiers 17
- unexpected function 353, 366, 367, 368
 - set_unexpected 369
- Unicode 34
- unions 55
 - as class type 241, 242
 - compatibility 64, 65
 - designated initializer 90
 - initialization 94
 - ISO support 481
 - packing
 - using `_Packed` qualifier 99
 - specifier 56
 - unnamed members 92
- universal character name 16, 27, 34
- UNIX System Services xxxiv
- UNIX System Services C functions xxxvi
- unnamed namespaces 219
- unsigned type specifiers
 - char 54
 - int 49
 - long 49
 - long long 49
 - short 49
- unsubscripted arrays
 - description 86, 202
- user-defined conversions 311
- user-defined data types 39, 55
- using declarations 222, 280, 289
 - changing member access 283
 - overloading member functions 281
- using directive 222
- USL xxix

V

- variable
 - in specified registers 48
- variable attributes 100
- variable length array 40, 86, 177
 - as function parameter 86, 208, 236
 - sizeof 114
 - type name 80
- variable pragma 451
- variables
 - storage of 480
- variably modified types 85, 86, 167
- virtual
 - base classes 275, 285, 290
- virtual functions 254, 291
 - access 296
 - ambiguous calls 294
 - overriding 295
 - pure specifier 296
- visibility 1, 6
 - block 2
 - class members 266
- void 54
 - in function definition 198, 201

- void (*continued*)
 - pointer 107, 108
- volatile
 - member functions 254
 - qualifier 67, 74

W

- wchar_t type specifier 26, 49, 54
- while statement 170
- white space 13, 36, 373, 378
- wide characters
 - ISO support 477
 - literals 26
- wide string literal 29
- wsizetof pragma 452

X

- XL C compiler-specific features xxviii
- XL C/C++ compiler utilities xxix
- XL C++ compiler-specific features xxix
- XOPTS pragma 454

Z

- z/OS UNIX System Services xxxiv
- z/OS UNIX System Services C functions xxxvi
- z/OS XL C compiler-specific features xxviii
- z/OS XL C/C++ compiler utilities xxix
- z/OS XL C++ compiler-specific features xxix



Program Number: 5694-A01

SC09-4815-07

