IBM i
7.3

Programming
IBM Rational Development Studio for i
ILE C/C++ Language Reference

SC09-7852-04

Note

Before using this information and the product it supports, read the information in “Notices” on page 413.

This edition applies to IBM® Rational® Development Studio for i (product number 5770-WDS) and to all subsequent releases and modifications until otherwise indicated in new editions. This version does not run on all reduced instruction set computer (RISC) models nor does it run on CISC models.

This document may contain references to Licensed Internal Code. Licensed Internal Code is Machine Code and is licensed to you under the terms of the IBM License Agreement for Machine Code.

© Copyright International Business Machines Corporation 1993, 2015.
US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

ILE C/C++ Language Reference..... 1

What's new for IBM i 7.3.....3

PDF file for ILE C/C++ Language Reference.....5

About ILE C/C++ Language Reference.....7
- Highlighting Conventions.....7
- Industry Standards.....8
- How to Read the Syntax Diagrams.....8
- Prerequisite and related information.....10

Scope and linkage.....11
- Scope.....11
  - Block/local scope.....12
  - Function scope.....12
  - Function prototype scope.....13
  - File/global scope.....13
  - Examples of scope in C.....13
  - Class scope (C++ only).....14
  - Namespaces of identifiers.....15
  - Name hiding (C++ only).....15
- Program linkage.....16
  - Internal linkage.....17
  - External linkage.....17
  - No linkage.....18
  - Language linkage.....18
    - Name mangling (C++ only).....19

Lexical elements.....21
- Tokens.....21
  - Keywords.....21
    - Keywords for language extensions.....22
  - Identifiers.....23
    - Characters in identifiers.....23
    - Reserved identifiers.....23
    - The __func__ predefined identifier.....24
  - Literals.....24
    - Integer literals.....25
      - Decimal integer literals.....27
      - Hexadecimal integer literals.....28
      - Octal integer literals.....28
    - Boolean literals.....28
    - Floating-point literals.....28
      - Binary floating-point literals.....28
      - Hexadecimal floating-point literals.....30
      - Decimal floating-point literals.....31
    - Packed Decimal Literals.....32
    - Character literals.....32
    - String literals.....33
      - String concatenation.....34

Pointer literals (C++11)....35
Punctuators and operators....35
Alternative tokens....36
Source program character set....36
Multibyte characters....37
Escape sequences....38
The Unicode standard (C++ only)....39
Digraph characters....39
Trigraph sequences....40
Comments....40

Data objects and declarations....43
Overview of data objects and declarations....43
Overview of data objects....43
Incomplete types....44
Compatible and composite types....45
Overview of data declarations and definitions....45
Tentative definitions....46
static_assert declaration (C++11)....47
Storage class specifiers....48
The auto storage class specifier....49
Storage duration of automatic variables....49
Linkage of automatic variables....49
The static storage class specifier....49
Linkage of static variables....50
The extern storage class specifier....50
Storage duration of external variables....51
Linkage of external variables....51
The mutable storage class specifier (C++ only)....51
The register storage class specifier....52
Storage duration of register variables....52
Linkage of register variables....52
The __thread storage class specifier....52
Type specifiers....53
Integral types....54
Boolean types....55
Floating-point types....55
Real floating-point types....55
Character types....56
The void type....57
Compatibility of arithmetic types (C only)....57
The auto type specifier (C++11)....57
The decltype(expression) type specifier (C++11)....59
The constexpr specifier (C++11)....63
User-defined types....65
Structures and unions....65
Structure and union type definition....66
Member declarations....66
Flexible array members....67
Zero-extent array members....67
Bit field members....68
Structure and union variable declarations....69
Structure and union type and variable definitions in a single statement....70
Access to structure and union members....71
Anonymous unions....71
Enumerations....72
Enumeration type definition....72
Enumeration members....74

Enumeration variable declarations....76
Enumeration type and variable definitions in a single statement....76
Compatibility of structures, unions, and enumerations (C only)....77
Compatibility across separate source files....77
typedef definitions....77
Examples of typedef definitions....78
Type qualifiers....79
The __align type qualifier....80
Examples using the __align qualifier....81
The const type qualifier....82
The restrict type qualifier(C++ only)....83
The volatile type qualifier....84
Type attributes....84
The aligned type attribute....85
The packed type attribute....85
The transparent_union type attribute (C only)....86

Declarators....89
Overview of declarators....89
Examples of declarators....90
Type names....91
Pointers....92
Pointer arithmetic....93
Type-based aliasing....93
Compatibility of pointers (C only)....94
Null pointers....95
Arrays....97
Variable length arrays....98
Compatibility of arrays....99
References (C++ only)....99
Initializers....100
Initialization and storage classes....101
Initialization of automatic variables....101
Initialization of static variables....101
Initialization of external variables....101
Initialization of register variables....102
Designated initializers for aggregate types (C only)....102
Initialization of structures and unions....104
Initialization of enumerations....106
Initialization of pointers....107
Initialization of arrays....108
Initialization of character arrays....108
Initialization of multidimensional arrays....109
Initialization of references (C++ only)....110
Reference binding....111
Direct binding....113
Variable attributes....113
The aligned variable attribute....114
The packed variable attribute....115
The mode variable attribute....116
The weak variable attribute....116

Type conversions....117
Arithmetic conversions and promotions....117
Integral conversions....118
Boolean conversions....118
Floating-point conversions....118
Integral and floating-point promotions....119

Lvalue-to-rvalue conversions ... 120
Pointer conversions ... 120
Conversion to void* ... 121
Reference conversions (C++ only) ... 122
Qualification conversions (C++ only) ... 122
Function argument conversions ... 122

Expressions and Operators ... 125
Lvalues and rvalues ... 125
Primary expressions ... 127
Names ... 127
Literals ... 128
Integer constant expressions ... 128
Identifier expressions (C++ only) ... 129
Parenthesized expressions ( ) ... 129
Scope resolution operator :: (C++ only) ... 130
Generalized constant expressions (C++11) ... 131
Function call expressions ... 132
Member expressions ... 132
Dot operator ... 132
Arrow operator -> ... 133
Unary expressions ... 133
Increment operator ++ ... 134
Decrement operator -- ... 134
Unary plus operator + ... 134
Unary minus operator - ... 135
Logical negation operator ! ... 135
Bitwise negation operator ~ ... 135
Address operator & ... 135
Indirection operator * ... 136
The typeid operator (C++ only) ... 137
The __alignof__ operator ... 138
The sizeof operator ... 138
The __typeof__ operator ... 140
Binary expressions ... 141
Assignment operators ... 141
Simple assignment operator = ... 142
Compound assignment operators ... 142
Multiplication operator * ... 143
Division operator / ... 143
Remainder operator % ... 143
Addition operator + ... 144
Subtraction operator - ... 144
Bitwise left and right shift operators << >> ... 144
Relational operators < > <= >= ... 145
Equality and inequality operators == != ... 146
Bitwise AND operator & ... 147
Bitwise exclusive OR operator ^ ... 147
Bitwise inclusive OR operator | ... 148
Logical AND operator && ... 148
Logical OR operator || ... 149
Array subscripting operator [ ] ... 149
Comma operator , ... 150
Pointer to member operators .* ->* (C++ only) ... 152
Conditional expressions ... 152
Types in conditional C expressions ... 153
Types in conditional C++ expressions ... 153
Examples of conditional expressions ... 154

Cast expressions ... 154
Cast operator () ... 154
The static_cast operator (C++ only) ... 156
The reinterpret_cast operator (C++ only) ... 158
The const_cast operator (C++ only) ... 159
The dynamic_cast operator (C++ only) ... 160
Compound literal expressions ... 162
new expressions (C++ only) ... 163
Placement syntax ... 164
Initialization of objects created with the new operator ... 165
Handling new allocation failure ... 165
delete expressions (C++ only) ... 166
throw expressions (C++ only) ... 167
Operator precedence and associativity ... 167
Reference collapsing (C++11) ... 171

Statements ... 175
Labeled statements ... 175
Expression statements ... 176
Resolution of ambiguous statements (C++ only) ... 176
Block statements ... 177
Example of blocks ... 177
Selection statements ... 177
The if statement ... 178
Examples of if statements ... 178
The switch statement ... 179
Restrictions on switch statements ... 181
Examples of switch statements ... 181
Iteration statements ... 183
The while statement ... 183
The do statement ... 184
The for statement ... 184
Examples of for statements ... 185
Jump statements ... 186
The break statement ... 186
The continue statement ... 186
Examples of continue statements ... 187
The return statement ... 188
Examples of return statements ... 188
The goto statement ... 189
Null statement ... 190

Functions ... 191
Function declarations and definitions ... 191
Function declarations ... 191
Function definitions ... 192
Explicitly defaulted functions (C++11) ... 193
Deleted functions (C++11) ... 194
Examples of function declarations ... 195
Examples of function definitions ... 195
Compatible functions (C only) ... 196
Multiple function declarations (C++ only) ... 196
Function storage class specifiers ... 197
The static storage class specifier ... 197
The extern storage class specifier ... 197
Function specifiers ... 199
The inline function specifier ... 199
Linkage of inline functions ... 200

Function return type specifiers....202
Function return values....203
Function declarators....204
Parameter declarations....204
Parameter types....205
Parameter names....206
Static array indices in function parameter declarations (C only)....206
Trailing return type (C++11)....207
Function attributes....209
The const function attribute....210
The noinline function attribute....211
The pure function attribute....211
The weak function attribute....211
The main() function....212
Function calls....213
Pass by value....214
Pass by reference....214
Allocation and deallocation functions (C++ only)....216
Default arguments in C++ functions (C++ only)....217
Restrictions on default arguments....218
Evaluation of default arguments....218
Pointers to functions....219
Constexpr functions (C++11)....220

Namespaces (C++ only)....223
Defining namespaces (C++ only)....223
Declaring namespaces (C++ only)....223
Creating a namespace alias (C++ only)....223
Creating an alias for a nested namespace....224
Extending namespaces (C++ only)....224
Namespaces and overloading (C++ only)....224
Unnamed namespaces (C++ only)....225
Namespace member definitions (C++ only)....226
Namespaces and friends (C++ only)....227
The using directive (C++ only)....227
The using declaration and namespaces....228
Explicit access (C++ only)....228
Inline namespace definitions (C++11)....229

Overloading (C++ only)....233
Overloading functions (C++ only)....233
Restrictions on overloaded functions....234
Overloading operators (C++ only)....235
Overloading unary operators (C++ only)....236
Overloading increment and decrement operators....237
Overloading binary operators (C++ only)....238
Overloading assignments (C++ only)....238
Overloading function calls (C++ only)....239
Overloading subscripting (C++ only)....240
Overloading class member access (C++ only)....241
Overload resolution (C++ only)....242
Implicit conversion sequences (C++ only)....243
Standard conversion sequences....243
User-defined conversion sequences....244
Ellipsis conversion sequences....244
Resolving addresses of overloaded functions....244

Classes (C++ only)....247

Declaring class types (C++ only)....247
Using class objects (C++ only).... 248
Classes and structures (C++ only)....249
Scope of class names (C++ only)....250
Incomplete class declarations (C++ only).... 251
Nested classes (C++ only)....251
Local classes (C++ only)....253
Local type names (C++ only)....254

Class members and friends (C++ only)....255
Class member lists (C++ only)....255
Data members (C++ only)....256
Member functions (C++ only).... 256
Inline member functions (C++ only).... 257
Constant and volatile member functions....258
Virtual member functions (C++ only)....258
Special member functions (C++ only)....258
Member scope (C++ only)....258
Pointers to members (C++ only)....259
The this pointer (C++ only).... 261
Static members (C++ only).... 263
Using the class access operators with static members (C++ only)....263
Static data members (C++ only)....264
Static member functions (C++ only).... 265
Member access (C++ only)....267
Friends (C++ only).... 269
Friend scope (C++ only)....272
Friend access (C++ only).... 274

Inheritance (C++ only)....275
Derivation (C++ only).... 276
Inherited member access (C++ only).... 279
Protected members (C++ only).... 279
Access control of base class members....280
The using declaration and class members....281
Overloading member functions from base and derived classes (C++ only).... 282
Changing the access of a class member....284
Multiple inheritance (C++ only).... 285
Virtual base classes (C++ only).... 286
Multiple access (C++ only).... 287
Ambiguous base classes (C++ only).... 287
Name hiding.... 288
Ambiguity and using declarations.... 289
Unambiguous class members.... 289
Pointer conversions....290
Overload resolution....290
Virtual functions (C++ only).... 291
Ambiguous virtual function calls (C++ only)....294
Virtual function access (C++ only).... 295
Abstract classes (C++ only).... 295

Special member functions (C++ only)....299
Overview of constructors and destructors.... 299
Constructors (C++ only)....300
Default constructors (C++ only)....300
Delegating constructors (C++11)....302
Constexpr constructors (C++11)....303
Explicit initialization with constructors (C++ only)....305

Initialization of base classes and members ... 306
Construction order of derived class objects ... 309
Destructors (C++ only) ... 311
Pseudo-destructors (C++ only) ... 312
User-defined conversions (C++ only) ... 313
Conversion constructors (C++ only) ... 314
Explicit conversion constructors (C++ only) ... 315
The explicit specifier (C++ only) ... 316
Conversion functions (C++ only) ... 317
Explicit conversion operators (C++11) ... 318
Copy constructors (C++ only) ... 319
Copy assignment operators (C++ only) ... 320

Templates (C++ only) ... 323
Template parameters (C++ only) ... 324
Type template parameters (C++ only) ... 324
Non-type template parameters (C++ only) ... 324
Template template parameters (C++ only) ... 325
Default arguments for template parameters ... 325
Naming template parameters as friends (C++11) ... 326
Template arguments (C++ only) ... 326
Template type arguments (C++ only) ... 327
Template non-type arguments (C++ only) ... 327
Template template arguments (C++ only) ... 328
Class templates (C++ only) ... 329
Class template declarations and definitions ... 332
Static data members and templates (C++ only) ... 332
Member functions of class templates (C++ only) ... 333
Friends and templates (C++ only) ... 333
Function templates (C++ only) ... 334
Template argument deduction (C++ only) ... 335
Deducing type template arguments ... 337
Deducing non-type template arguments ... 339
Overloading function templates (C++ only) ... 340
Partial ordering of function templates ... 341
Template instantiation (C++ only) ... 342
Implicit instantiation (C++ only) ... 342
Explicit instantiation (C++ only) ... 343
Template specialization (C++ only) ... 345
Explicit specialization (C++ only) ... 345
Definition and declaration of explicit specializations ... 347
Explicit specialization and scope ... 347
Class members of explicit specializations ... 347
Explicit specialization of function templates ... 348
Explicit specialization of members of class templates ... 348
Partial specialization (C++ only) ... 350
Template parameter and argument lists of partial specializations ... 351
Matching of class template partial specializations ... 351
Variadic templates (C++11) ... 352
Parameter packs (C++11) ... 352
Pack expansion (C++11) ... 354
Partial specialization (C++11) ... 360
Template argument deduction (C++11) ... 361
Name binding and dependent names (C++ only) ... 362
The typename keyword (C++ only) ... 363
The template keyword as qualifier (C++ only) ... 364

Exception handling (C++ only) ... 367

try blocks (C++ only) 367
Nested try blocks (C++ only) 368
catch blocks (C++ only) 369
Function try block handlers (C++ only) 369
Arguments of catch blocks (C++ only) 373
Matching between exceptions thrown and caught 373
Order of catching (C++ only) 373
throw expressions (C++ only) 374
Rethrowing an exception (C++ only) 375
Stack unwinding (C++ only) 376
Exception specifications (C++ only) 378
Special exception handling functions 380
The unexpected() function (C++ only) 380
The terminate() function (C++ only) 381
The set_unexpected() and set_terminate() functions 382
Example using the exception handling functions (C++ only) 382

Preprocessor directives 385
Macro definition directives 385
The #define directive 385
Object-like macros 386
Function-like macros 387
Variadic macro extensions 389
The #undef directive 390
The # operator 390
The ## operator 391
File inclusion directives 392
The #include directive 392
Using the #include Directive when Compiling Source in a Data Management File 392
Using the #include Directive When Compiling Source in an Integrated File System File 394
The #include_next directive 395
Conditional compilation directives 395
The #if and #elif directives 397
The #ifdef directive 397
The #ifndef directive 398
The #else directive 398
The #endif directive 398
Message generation directives 399
The #error directive 399
The #warning directive 400
The #line directive 400
Assertion directives 401
The null directive (#) 402
Pragma directives 402
The _Pragma preprocessing operator 403
C99 preprocessor features adopted in C++11 (C++11) 403

The ILE C language extensions 407
C99 features as extensions to C89 407
Extensions for GNU C compatibility 408
Extensions for decimal floating-point support 408

The ILE C++ language extensions 409
General IBM extensions 409
Extensions for C99 compatibility 409
Extensions for GNU C compatibility 410
Extensions for GNU C++ compatibility 410
Extensions for C++11 compatibility 411

Extensions for decimal floating-point support....412

Notices....413
Programming interface information....414
Trademarks....414
Terms and conditions....415

Index....417

ILE C/C++ Language Reference

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 ILE C/C++; however, this reference does not aim to teach programming concepts nor to promote specific programming practices.

The focus of this book is on the fundamentals and intricacies of the C and C++ languages. The availability of a particular language feature at a particular language level is controlled by compiler options. Comprehensive coverage of the possibilities offered by the compiler options is available in ILE C/C++ Compiler Reference.

The depth of coverage assumes some previous experience with C or another programming language. The intent is to present the syntax and semantics of each language implementation to help you write good programs. The compiler does not enforce certain conventions of programming style, even though they lead to well-ordered programs.

A program that conforms strictly to its language specification will have maximum portability among different environments. In theory, a program that compiles correctly with one standards-conforming compiler will compile and execute correctly under all other conforming compilers, insofar as hardware differences permit. A program that correctly exploits the extensions to the language that are provided by the language implementation can improve the efficiency of its object code.

© Copyright IBM Corp. 1993, 2015

What's new for IBM i 7.3

In addition to the existing C++11 features, new C++11 features are supported in this release of ILE C++ compiler.

Note: IBM supports selected features of C++11, known as C++0x before its ratification. IBM will continue to develop and implement the features of this standard. The implementation of the language level is based on IBM's interpretation of the standard. Until IBM's implementation of all the C++11 features is complete, including the support of a new C++11 standard library, the implementation may change from release to release. IBM makes no attempt to maintain compatibility, in source, binary, or listings and other compiler interfaces, with earlier releases of IBM's implementation of the new C++11 features.

- Defaulted and deleted functions
- Explicit conversion operators
- Generalized constant expressions
- Reference collapsing
- Right angle brackets
- Rvalue references
- Scoped enumerations
- The `nullptr` keyword
- Trailing return type
- Variadic template

You can use the LANGLVL(*EXTENDED0X) option to enable most of the C++ features and all the currently supported C++11 features.

Defaulted and deleted functions

This feature introduces two new forms of function declarations to define explicitly defaulted functions and deleted functions. For the explicitly defaulted functions, the compiler generates the default implementations, which are more efficient than manuallyprogrammed implementations. The compiler disables the deleted functions to avoid calling unwanted functions.

For more information, see “Explicitly defaulted functions (C++11)” on page 193 and “Deleted functions (C++11)” on page 194.

Explicit conversion operators

The explicit conversion operators feature supports the `explicit` function specifier being applied to the definition of a user-defined conversion function. You can use this feature to inhibit implicit conversions from being applied where they might be unintended, and thus program more robust classes with fewer ambiguity errors.

For more information, see “Explicit conversion operators (C++11)” on page 318.

Generalized constant expressions

The generalized constant expressions feature extends the expressions permitted within constant expressions. A constant expression is one that can be evaluated at compile time.

For more information, see “Generalized constant expressions (C++11)” on page 131.

Reference collapsing

With the reference collapsing feature, you can form a reference to a reference type using one of the following contexts:

- A `decltype` specifier
- A `typedef` name
- A template type parameter

© Copyright IBM Corp. 1993, 2015

For more information, see “Reference collapsing (C++11)” on page 171.

Right angle brackets
In the C++ language, two consecutive closing angle brackets (>) must be separated with a white space, because they are otherwise parsed as the bitwise right-shift operator (>>). The right angle bracket feature removes the white space requirement for consecutive right angle brackets, thus making programming more convenient.

For more information, see “Class templates (C++ only)” on page 329.

Rvalue references
With the rvalue references feature, you can overload functions based on the value categories of arguments and similarly have lvalueness detected by template argument deduction. You can also have an rvalue bound to an rvalue reference and modify the rvalue through the reference. This enables a programming technique with which you can reuse the resources of expiring objects and therefore improve the performance of your libraries, especially if you use generic code with class types, for example, template data structures. Additionally, the value category can be considered when writing a forwarding function.

For more information, see “References (C++ only)” on page 99.

Scoped enumerations

With the scoped enumeration feature, you can get the following benefits:

- The ability to declare a scoped enumeration type, whose enumerators are declared in the scope of the enumeration.
- The ability to declare an enumeration without providing the enumerators. The declaration of an enumeration without providing the enumerators is referred to as forward declaration.
- The ability to specify explicitly the underlying type of an enumeration.
- Improved type safety with no conversions from the value of an enumerator (or an object of an enumeration type) to an integer.

For more information, see “Enumerations” on page 72.

The nullptr keyword
This feature introduces `nullptr` as a null pointer constant. The `nullptr` constant can be distinguished from integer 0 for overloaded functions. The constants of 0 and NULL are treated as of the integer type for overloaded functions, whereas `nullptr` can be implicitly converted to only the pointer type, pointer-to-member type, and bool type.

For more information, see “Null pointers ” on page 95.

Trailing return type

The trailing return type feature is useful when declaring the following types of templates and functions:

- Function templates or member functions of class templates with return types that depend on the types of the function arguments
- Functions or member functions of classes with complicated return types
- Perfect forwarding functions

For more information, see “Trailing return type (C++11)” on page 207.

Variadic templates
With the variadic templates feature, you can define class or function templates that have any number (including zero) of parameters.

For more information, see “Variadic templates (C++11)” on page 352.

PDF file for ILE C/C++ Language Reference

You can view and print a PDF file of this information.

To view or download the PDF version of this document, select ILE C/C++ Language Reference.

Saving PDF files

To save a PDF on your workstation for viewing or printing:

1. Right-click the PDF link in your browser.
2. Click the option that saves the PDF locally.
3. Navigate to the directory in which you want to save the PDF.
4. Click **Save**.

Downloading Adobe Reader

You need Adobe Reader installed on your system to view or print these PDFs. You can download a free copy from the Adobe Web site (www.adobe.com/products/acrobat/readstep.html).

© Copyright IBM Corp. 1993, 2015

About ILE C/C++ Language Reference

The *C/C++ Language Reference* describes the syntax, semantics, and IBM implementation of the C and C++ programming languages. Syntax and semantics constitute a complete specification of a programming language, but complete implementations can differ because of extensions. The IBM implementations of Standard C and Standard C++ attest to the organic nature of programming languages, reflecting pragmatic considerations and advances in programming techniques, which are factors that influence growth and change. The language extensions to C and C++ also reflect the changing needs of modern programming environments.

The aims of this reference are to provide a description of the C and C++ languages outside of any comprehensive historical context, and to promote a programming style that emphasizes portability. The expression *Standard C* is a generic term for the current formal definition of the C language, preprocessor, and runtime library. The expression is ambiguous because subsequent formal definitions of the language have appeared while implementations of its predecessors are still in use. This reference describes a C language consistent with the C89 language level. To avoid further ambiguity and confusion with K&R C, this reference uses *ISO C* to mean Standard C, avoiding the term *Standard C*, and the term *K&R C* to refer to the C language plus the generally accepted extensions produced by Brian Kernighan and Dennis Ritchie (K&R C) that were in use prior to ISO C. The expression *Standard C++* refers to the C++ language Standard (ISO/IEC 14882:2003).

The C and C++ languages described in this reference are based on the following standards:

- Information Technology - Programming languages - C, ISO/IEC 9899:1990, also known as C89
- Information Technology - Programming languages - C, ISO/IEC 9899:1999, also known as C99
- Information Technology - Programming languages - C++, ISO/IEC 14882:1998, also known as C++98
- Information Technology - Programming languages - C++, ISO/IEC 14882:2003(E), also known as Standard C++
- Information Technology - Programming languages - Extension for the programming language C to support decimal floating-point arithmetic, ISO/IEC WDTR 24732. This draft technical report has been submitted to the C standards committee, and is available at http://www.open-std.org/JTC1/SC22/WG14/www/docs/n1176.pdf.

In addition to the standardized language levels, ILE C++ supports a subset of C++0x features.

Note: C++0x is a new version of the C++ programming language standard. IBM continues to develop and implement the features of the new standard. The implementation of the language level is based on IBM's interpretation of the standard. Until IBM's implementation of all the features of the C++0x standard is complete, including the support of a new C++ standard library, the implementation may change from release to release. IBM makes no attempt to maintain compatibility, in source, binary, or listings and other compiler interfaces, with earlier releases of IBM's implementation of the new features of the C++0x standard and therefore they should not be relied on as a stable programming interface.

C++0x has been ratified and published as ISO/IEC 14882:2011. All references to C++0x in this document are equivalent to the ISO/IEC 14882:2011 standard. Corresponding information, including programming interfaces, will be updated in a future release.

IBM i The C++ language described in this reference is consistent with Standard C++ and documents the features supported by the IBM C++ compiler.

Highlighting Conventions

Highlight	Usage
Bold	Identifies commands, keywords, files, directories, and other items whose names are predefined by the system.

© Copyright IBM Corp. 1993, 2015

Highlight	Usage
Italics	Identify parameters whose actual names or values are to be supplied by the programmer. *Italics* are also used for the first mention of new terms.
`Example`	Identifies examples of specific data values, examples of text similar to what you might see displayed, examples of portions of program code, messages from the system, or information that you should actually type.

Examples are intended to be instructional and do not attempt to minimize runtime, conserve storage, or check for errors. The examples do not demonstrate all of the possible uses of language constructs. Some examples are only code fragments and will not compile without additional code.

This document uses icons to delineate text as follows:

Table 1. Qualifying Elements

Qualifier/Icon	Meaning
C only C	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 C++	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 IBM i	The text describes a feature that is an IBM compiler extension to the standard language specifications.
C++0x only C++0x	The text describes a feature that is introduced into standard C++ as part of C++0x.

Industry Standards

The following standards are supported:

- The C language is consistent with the International Standard for Information Systems-Programming Language C (ISO/IEC 9899-1990).
- The C language is also consistent with the International Standard for Information Systems-Programming Language C (ISO/IEC 9899–1999 (E)).
- The C++ language is consistent with the International Standard for Information Systems-Programming Language C++ (ISO/IEC 14882:1998).
- The C++ language is also consistent with the International Standard for Information Systems-Programming Language C++ (ISO/IEC 14882:2003(E)).

How to Read the Syntax 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.

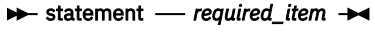
  The ►—— symbol indicates that a command, directive, or statement is continued from the previous line.

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

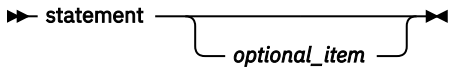
  Diagrams of syntactical units other than complete commands, directives, or statements start with the ►—— symbol and end with the ——► symbol.

Note: In the following diagrams, `statement` represents a C or C++ command, directive, or statement.

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

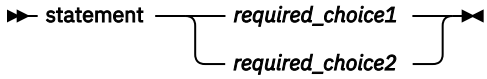


- Optional items appear below the main path.

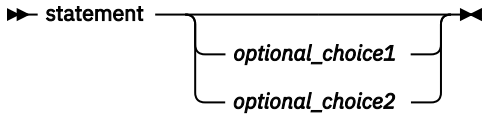


- If you can choose from two or more items, they appear vertically, in a stack.

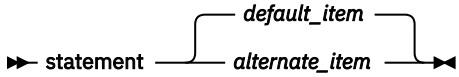
  If you *must* choose one of the items, one item of the stack appears on the main path.



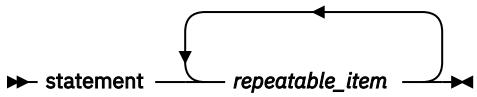
  If choosing one of the items is optional, the entire stack appears below the main path.



  The item that is the default appears above the main path.



- An arrow returning to the left above the main line indicates an item that can be repeated.



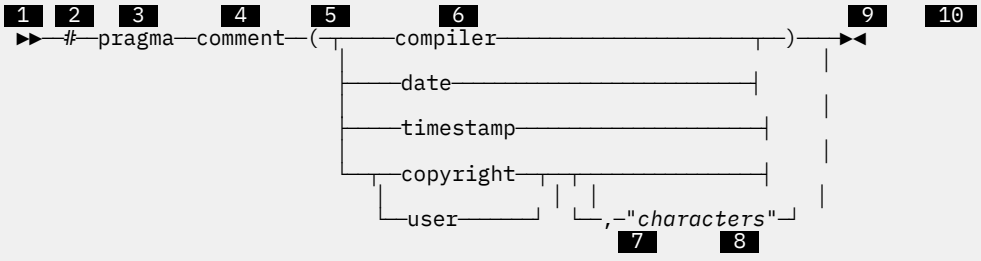
  A repeat arrow above a stack indicates that you can make more than one choice from the stacked items, or repeat a single choice.

- Keywords appear in nonitalic letters and should be entered exactly as shown (for example, `extern`).

  Variables appear in italicized lowercase letters (for example, *identifier*). 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.

The following syntax diagram example shows the syntax for the `#pragma comment` directive. See "Pragma directives" on page 402 for information on the `#pragma` directive.



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")
```

Prerequisite and related information

Use the IBM i Information Center as your starting point for IBM i technical information.

You can access the information center as follows:

- From the following Web site:

```
http://www.ibm.com/systems/i/infocenter/
```

The IBM i Information Center contains new and updated system information, such as software and hardware installation, Linux®, WebSphere®, Java™, high availability, database, logical partitions, CL commands, and system application programming interfaces (APIs). In addition, it provides advisors and finders to assist in planning, troubleshooting, and configuring your system hardware and software.

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” on page 11. The different types of linkages are discussed in “Program linkage” on page 16.

Related information

- “Storage class specifiers” on page 48
- “Namespaces (C++ only)” on page 223

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.

© Copyright IBM Corp. 1993, 2015

Table 2. 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

- “Namespaces (C++ only)” on page 223

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 177

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 175

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 191

File/global scope

C *Beginning of 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.

C *End of C only.*

C++ *Beginning of 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.

C++ *End of C++ only.*

Related information

- “Namespaces (C++ only)” on page 223
- “Internal linkage” on page 17
- “The extern storage class specifier” on page 50

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[])
{

    printf("%d\n", i);              /* Prints 1 */

    {
        int i = 2, j = 3;           /* i and j defined at block scope */
                                   /* global definition of i is hidden */
        printf("%d\n%d\n", i, j);   /* Prints 2, 3 */

        {
            int i = 0;  /* i is redefined in a nested block */
                        /* previous definitions of i are hidden */
            printf("%d\n%d\n", i, j); /* Prints 0, 3 */
        }

        printf("%d\n", i);          /* Prints 2 */

    }

    printf("%d\n", i);              /* Prints 1 */

    return 0;

}
```

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

- “Classes (C++ only)” on page 247
- “Scope of class names (C++ only)” on page 250
- “Member scope (C++ only)” on page 258
- “Friend scope (C++ only)” on page 272
- “Access control of base class members” on page 280
- “Scope resolution operator :: (C++ only)” on page 130

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

- “Classes (C++ only)” on page 247
- “Member functions (C++ only)” on page 256
- “Member scope (C++ only)” on page 258
- “Namespaces (C++ only)” on page 223

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” on page 17 : identifiers can only be seen within a translation unit.
- “External linkage” on page 17 : identifiers can be seen (and referred to) in other translation units.
- “No linkage” on page 18: 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++ *Beginning of C++ only.*

You can have linkage between translation units written in different programming languages, which is called *language linkage*. Language linkage enables the close relationship among all ILE languages by allowing code in one ILE language to link with code written in another ILE language. In C++, all identifiers have a language linkage, which by default is C++. Language linkage must be consistent across translation units. Non-C or non-C++ language linkage implies that an identifier has external linkage. For IBM i specific usage information, see "ILE Calling Conventions" in *ILE C/C++ Programmer's Guide.*

C++ *End of C++ only.*

Related information

- “The static storage class specifier” on page 49
- “The extern storage class specifier” on page 50

- “Function storage class specifiers” on page 197
- “Type qualifiers” on page 79
- “Anonymous unions” on page 71

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` or C++11 `constexpr` 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 *Beginning of 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.

C *End of C only.*

C++ *Beginning of 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.

C++ *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.

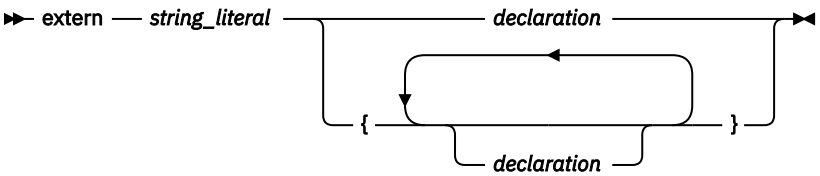
Language linkage

IBM i C On an IBM i system, language linkage is available for C through the use of `#pragma argument`. See "ILE C/C++ Pragmas" in *ILE C/C++ Compiler Reference* and "ILE Calling Conventions" in *ILE C/C++ Programmer's Guide* for more information.

C++ 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

IBM i See "Working with Multi-Language Applications" in the *ILE C/C++ Programmer's Guide* for additional language linkages supported by ILE C++.

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');
}
```

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.

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" on page 21
- "Source program character set" on page 36
- "Comments" on page 40

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" on page 21
- "Identifiers" on page 23
- "Literals" on page 24
- "Punctuators and operators" on page 35

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 AUT0 is not.

Table 3. C and C++ keywords

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

C++11 *Beginning of C++11 only.*

In C++11, the keyword `auto` is no longer used as a storage class specifier. Instead, it is used as a type specifier, which can deduce the type of an `auto` variable from the type of its initializer expression.

The keyword `extern` was previously used as a storage specifier or as part of a linkage specification. The C++11 standard adds a third usage to use this keyword to specify explicit instantiation declarations.

C++11 *End of C++11 only.*

C++ *Beginning of C++ only.*

The C++ language also reserves the following keywords:

© Copyright IBM Corp. 1993, 2015

Table 4. C++ keywords

asm	export	operator	this
bool	decltype	private	throw
catch	false	protected	true
class	friend	public	try
const_cast	inline	reinterpret_cast	typeid
constexpr	mutable	static_assert	typename
delete	namespace	static_cast	using
dynamic_cast	new	template	virtual
explicit	nullptr		wchar_t

C++11 `constexpr`, `nullptr`, `decltype` and `static_assert` are four keywords in the C++11 standard.

C++ *End of C++ only.*

Keywords for language extensions

IBM i *Beginning of IBM Extension.*

In addition to standard language keywords, ILE C/C++ reserves the following keywords for use in language extensions and for future use:

Table 5. Keywords for C and C++ language extensions

__alignof	_Decimal32[1]	__restrict	decimal
__alignof__	_Decimal64[1]	__restrict__	_Decimal
__attribute__	_Decimal128[1]	__signed__	__align
__attribute	__extension__	__signed	_Packed
__const__	__label__	__static_assert[2]	__ptr128
	__inline__	__volatile__	__ptr64
		__thread[1]	bool (C only)
		typeof	
		__typeof__	

Note:

1. These keywords are recognized only when **LANGLVL(*EXTENDED)** or **LANGLVL(*EXTENDED0X)** is specified.

2. C++0x `__static_assert` is a keyword for C language extension for compatibility with the C++0x standard.

Table 6. Keywords for C and C++ language extensions related to C99

restrict (C++ only)
_Pragma
_Bool (C 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.

Related information

- **LANGLVL(*EXTENDED)** and **LANGLVL(*EXTENDED0X)** in the *ILE C/C++ Compiler Reference*

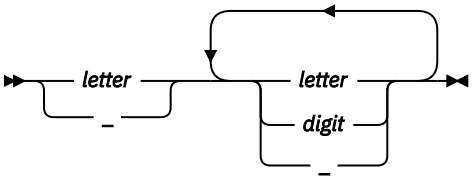
IBM i *End of IBM Extension.*

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 129
- “The Unicode standard (C++ only)” on page 39
- “Keywords” on page 21

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.

The universal character names for letters and digits outside of the basic source character set are allowed in C++ .

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

Related information

- **`LANGLVL(*EXTENDED)`** in the *ILE C/C++ Compiler Reference*

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 runtime. 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++ *Beginning of 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>`.

C++ *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
```

Related information

- “Function declarations and definitions” on page 191

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” on page 25
- “Boolean literals” on page 28
- “Floating-point literals” on page 28
- “Packed Decimal Literals” on page 32
- “Character literals” on page 32
- “String literals” on page 33

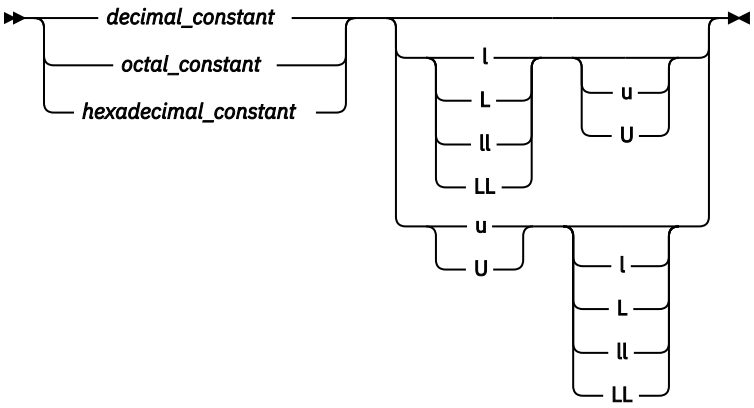
Integer literals

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

- “Decimal integer literals” on page 27
- “Hexadecimal integer literals” on page 28
- “Octal integer literals” on page 28

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

Integer literal syntax



The `long long` features

There are two `long long` features:

- the C99 `long long` feature
- the non-C99 `long long` feature

IBM i Both of the two features have the corresponding extension parts:

- the C99 `long long` feature with the associated IBM extensions
- the non-C99 IBM `long long` extension

Types of integer literals outside of C99

In the non-C99 modes, you can enable the non-C99 IBM `long long` extension.

The following table lists the integer literals and shows the possible data types when the C99 `long long` feature is not enabled.

Table 7. Types of integer literals outside of C99

Representation	Suffix	Promotion order					
		int	unsigned int	long int	unsigned long int	IBM i long long int	IBM i unsigned long long int
decimal	None	+		+	+		
octal, hex	None	+	+	+	+		
All	u or U		+		+		
decimal	l or L			+	+		
octal, hex	l or L			+	+		
All	Both u or U and l or L				+		
decimal	ll or LL					+	+
octal, hex	ll or LL					+	+
All	Both u or U and ll or LL						+

Note: When none of the `long long` features are enabled, types of integer literals include all the types in this table except the last two columns.

C Types of integer literals in C99

In the C99 modes, the C99 `long long` feature is enabled automatically.

After you enable the C99 `long long` feature, the compiler has all the functionality of the non-C99 IBM `long long` extension. Aside from literals that are out of range, the only difference is the specific typing rules for decimal integer literals that do not have a suffix containing u or U. Literals that are out of range under the non-C99 IBM `long long` extension might have implied type `long long int` or `unsigned long long int` under the C99 `long long` feature with the associated IBM extensions.

The following example demonstrates the different behaviors of the compiler when you use these two `long long` modes:

```
#include <stdio.h>

int main(){
    if(0>3999999999-4000000000){
        printf("C99 long long");
    }
    else{
        printf("non-C99 IBM long long extension");
    }
    return 0;
}
```

In this example, the values 3999999999 and 4000000000 are too large to fit into the a 32-bit `long int` type, but they can fit into either the `unsigned long` or the `long long int` type. If you enable the C99 `long long` feature, the two values have the `long long int` type, so the difference of 3999999999 and 4000000000 is negative. Otherwise, if you enable the non-C99 IBM `long long` extension, the two values have the `unsigned long` type, so the difference is positive.

When both the C99 and non-C99 `long long` features are disabled, integer literals that have one of the following suffixes cause a severe compiler error:

- `ll` or `LL`
- Both `u` or `U` and `ll` or `LL`

IBM i If a value cannot fit into the `long long int` type, the compiler might use the `unsigned long long int` type for the literal. In this case, the compiler generates a message to indicate that the value is too large.

The following table lists the integer literals and shows the possible data types when the C99 `long long` feature is enabled.

Table 8. Types of integer literals in C99

Representation	Suffix	Promotion order					
		int	unsigned int	long int	unsigned long int	IBM i long long int	IBM i unsigned long long int
decimal	None	+		+		+	+
octal, hex	None	+	+	+	+	+	+
All	u or U		+		+		+
decimal	l or L			+		+	+
octal, hex	l or L			+	+	+	+
All	Both u or U and l or L				+		+
decimal	ll or LL					+	+
octal, hex	ll or LL					+	+
All	Both u or U and ll or LL						+

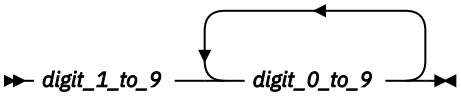
Related information

- “Integral types” on page 54
- “Integral conversions” on page 118
- **LANGLVL** in the ILE C/C++ Compiler Reference

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:

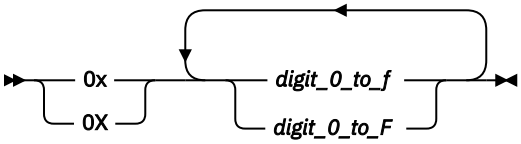
```
485976
-433132211
```

```
+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



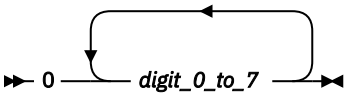
The following are examples of hexadecimal integer literals:

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

Related information

- “Boolean types” on page 55
- “Boolean conversions” on page 118

Floating-point literals

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

- “Binary floating-point literals” on page 28
- “Hexadecimal floating-point literals” on page 30
- IBM i “Decimal floating-point literals” on page 31

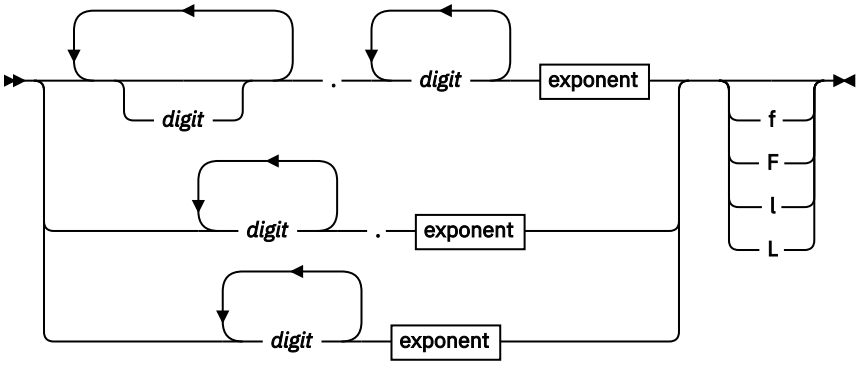
Binary floating-point literals

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

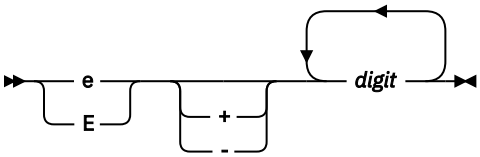
- An integral part
- A decimal point
- A fractional part
- An exponent part
- An optional suffix

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	0.00000000004
1e+5	100000
7.321E-3	0.007321
3.2E+4	32000
0.5e-6	0.0000005
0.45	0.45
6.e10	60000000000

Related information

- “Floating-point types” on page 55
- “Floating-point conversions ” on page 118

- “Unary expressions” on page 133

Hexadecimal floating-point literals

Real hexadecimal floating-point constants, which are a C99 feature, consist of the following:

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

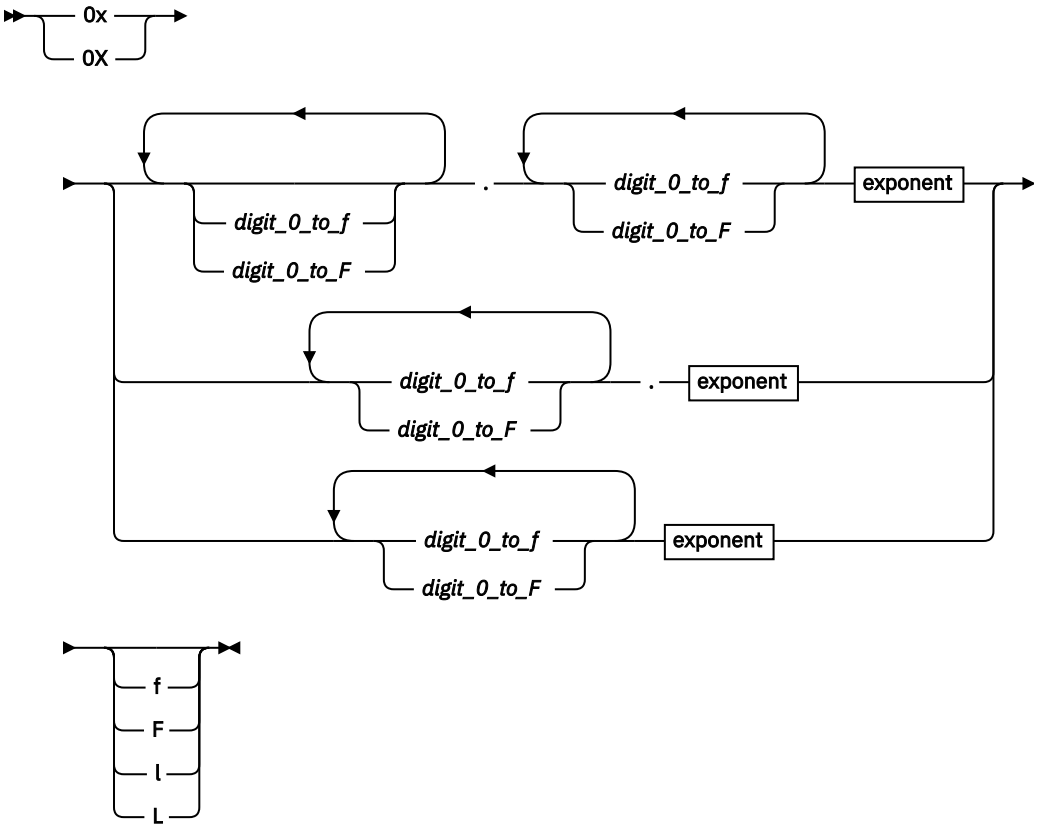
The significant part represents a rational number and is composed of the following:

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

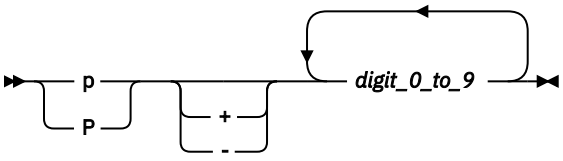
The optional fraction part is a period followed by a sequence of hexadecimal digits.

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

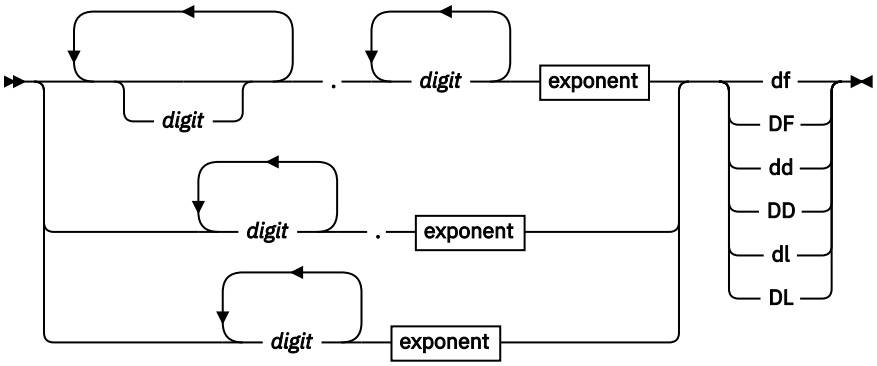
IBM i *Beginning of IBM Extension.*

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

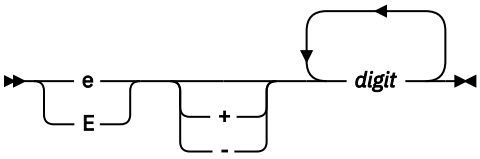
- An integral part
- A decimal point
- A fractional part
- An exponent part
- An optional suffix

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 `dl` 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;
```

When using decimal floating-point data, more accurate results will occur if decimal floating-point literals are used. As previously stated, an unsuffixed floating-point constant has type double. Consider this initialization:

```
_Decimal64 rate = 0.1;
```

The constant 0.1 has type double, which cannot accurately represent the decimal value 0.1. The variable rate will get a value slightly different from 0.1. The definition should be coded as follows:

```
_Decimal64 rate = 0.1dd;
```

When the decimal floating-point literals are converted or when the constant decimal floating-point expressions are resolved, the default rounding mode used will be "round to the nearest, ties to even."

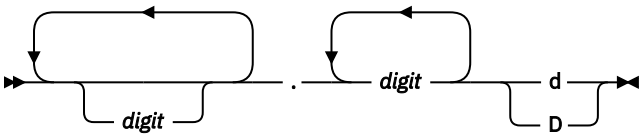
Note: Decimal floating-point literal suffixes are recognized only when the **LANGLVL(*EXTENDED)** is specified.

IBM i *End of IBM Extension.*

Packed Decimal Literals

IBM i C A *packed decimal literal* is a kind of floating-point literal that provides the ability to accurately represent large numeric quantities. It can consist of an integral part, a decimal point, a fractional part, and the mandatory suffix D. A packed decimal literal can have up to sixty-three significant digits, including integral and fractional parts.

A packed decimal literal is of the form:



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.

Related information

- See the *ILE C/C++ Programmer's Guide* for more information.

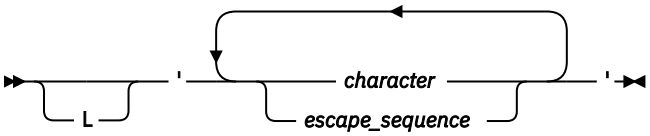
Character literals

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 38 for a list of other characters that are represented by escape characters.)

IBM i The value of a narrow or wide character literal containing a single character is the numeric representation of the character in the character set used at runtime. The lowest four bytes represent the value of an integer character literal that contains more than one character. The lowest two bytes of the lowest multibyte character represent the value of a wide character literal. For the locale type

LOCALETYPE(*LOCALEUTF), the lowest four bytes of the lowest multibyte character represent the value of the wide character literal.

Outside of the basic source character set, the universal character names for letters and digits are allowed in C++ .

The following are examples of character literals:

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

Related information

- “Source program character set” on page 36
- “The Unicode standard (C++ only)” on page 39
- “Character types ” on page 56

String literals

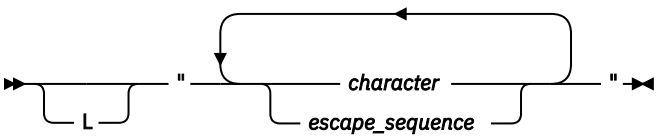
A *string literal* contains a sequence of characters or escape sequences enclosed in double quotation mark symbols. A string literal with the prefix L is a *wide string literal*. A string literal without the prefix L is an *ordinary* or *narrow string literal*.

C The type of a narrow string literal is array of `char`. The type of a wide string literal is array of `wchar_t`.

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++.

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:

```
#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. ILE 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. By default, the compiler considers strings to be writeable. You can use the **#pragma strings** directive or the PFROPT compiler option to change the default storage for string literals.

Related information

- “Character types ” on page 56
- “Source program character set” on page 36
- “The Unicode standard (C++ only)” on page 39
- **PFROPT** compiler option in the *ILE C/C++ Compiler Reference*
- **#pragma strings** in the *ILE C/C++ Compiler Reference*

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 .

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.

Note: C In C99, narrow strings can be concatenated with wide string literals. C++0x In C++0x, the changes to string literal concatenation in the C99 preprocessor are adopted to provide a common preprocessor interface for C and C++ compilers. Narrow strings can be concatenated with wide string literals in C++0x. For more information, see “C99 preprocessor features adopted in C++11 (C++11)” on page 403.

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"  */
```

Pointer literals (C++11)

The only pointer literal is the `nullptr` keyword that is a `prvalue` of type `std::nullptr_t`. A `prvalue` of this type is a null pointer constant that can be converted to any pointer type, pointer-to-member type, or bool type.

Related information

- “Null pointers ” on page 95
- “Pointer conversions” on page 120

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 9. C and C++ punctuators

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

C++ *Beginning of C++ only.*

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

Table 10. C++ punctuators

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

C++ *End of C++ only.*

Related information

- “Expressions and operators” on page 125

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
{	<%
}	%>
[	<:
]	:>
#	%:
##	%:%:

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 39

Source program character set

The following lists the basic source character sets that are available at both compile time and runtime:

- 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 or the equivalent vertical bar (|) character in EBCDIC.
- 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.

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

Related information

- "Characters in identifiers" on page 23

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.

Multibyte characters can appear in any of the following contexts:

- String literals and character constants. To declare a multibyte literal, use an ordinary character representation. For example:

```
char *a = "multibyte string";
char b = "multibyte-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.
- You cannot mix wide characters and multibyte characters in macro definitions. For example, a macro expansion that concatenates a wide string and a multibyte string is not permitted.
- Assignment between wide characters and multibyte characters is not permitted.
- Concatenating wide character strings and multibyte character strings is not permitted.

Related information

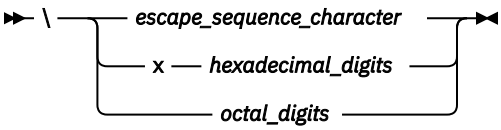
- "Character literals" on page 32

- “The Unicode standard (C++ only)” on page 39
- “Character types ” on page 56

Escape sequences

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

Escape character syntax



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

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

The escape sequences and the characters they represent are:

Escape sequence	Character represented
\a	Alert (bell, alarm)
\b	Backspace
\f	Form feed (new page)
\n	New-line
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab
\'	Single quotation mark
\"	Double quotation mark
\?	Question mark
\\	Backslash

The value of an escape sequence represents the member of the character set used at runtime. Escape sequences are translated during preprocessing. For example, 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 (C++ only)

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.

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**
where N *is a hexadecimal digit*	
\UNNNNNNNN	NNNNNNNN
\uNNNN	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.

The digraph characters are:

Digraph character	**Token**	**Description**
%: or %%	#	number sign
<:	[	left bracket
:>	]	right bracket
<%	{	left brace
%>	}	right brace
%:%: or %%%%	##	preprocessor macro concatenation operator

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

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

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

Related information

- **__DIGRAPHS__** in the *ILE C/C++ Compiler Reference*
- **OPTION(*DIGRAPH)** in the *ILE C/C++ Compiler Reference*

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;
```

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.

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 within a comment.

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** in the *ILE C/C++ Compiler Reference*
- “Multibyte characters” on page 37

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” on page 43
- “static_assert declaration (C++11)” on page 47
- “Storage class specifiers” on page 48
- “Type specifiers” on page 53
- “User-defined types” on page 65
- “Type qualifiers” on page 79
- “Type attributes” on page 84

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 “Declarators” on page 89.

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*.

C++ 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.

© Copyright IBM Corp. 1993, 2015

Table 11. C/C++ data types						
Data object	**Basic**	**Compound**	**Built-in**	**User-defined**	**Scalar**	**Aggregate**
integer types	+		+		+	
floating-point types	+		+		+	
character types			+		+	
Booleans	+		+		+	
void type	+[1]		+		+	
pointers		+	+		+	
arrays		+	+			+
structures		+		+		+
unions		+		+		
enumerations		+		+	see note[2]	
C++ classes		+		+		+
`typedef` types		+		+		

Note:

1. The `void` type is really an incomplete type, as discussed in "Incomplete types" on page 44. Nevertheless, Standard C++ defines it as a fundamental type.
2. C The C standard does not classify enumerations as either scalar or aggregate. C++ Standard C++ classifies enumerations as scalars.

Related information

- "Classes (C++ only)" on page 247

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
- C++ Pointers to class types that are declared but not defined
- C++ Classes that are declared but not defined

The following examples illustrate incomplete types:

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

Related information

- "The void type" on page 57
- "Incomplete class declarations (C++ only)" on page 251

Compatible and composite types

C *Beginning of 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 53 discusses compatibility for basic and user-defined types in C.

C *End of C only.*

C++ *Beginning of 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.

C++ *End of C++ only.*

Related information

- "Compatibility of arrays " on page 99
- "Compatibility of pointers (C only)" on page 94
- "Compatible functions (C only)" on page 196

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
`extern double pi;`	`double pi = 3.14159265;`

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

Note: C 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" on page 48, which specify storage duration and linkage
- "Type specifiers" on page 53, which specify data types
- "Type qualifiers" on page 79, which specify the mutability of data values
- "Overview of declarators" on page 89, which introduce and include identifiers
- "Initializers" on page 100, which initialize storage with initial values

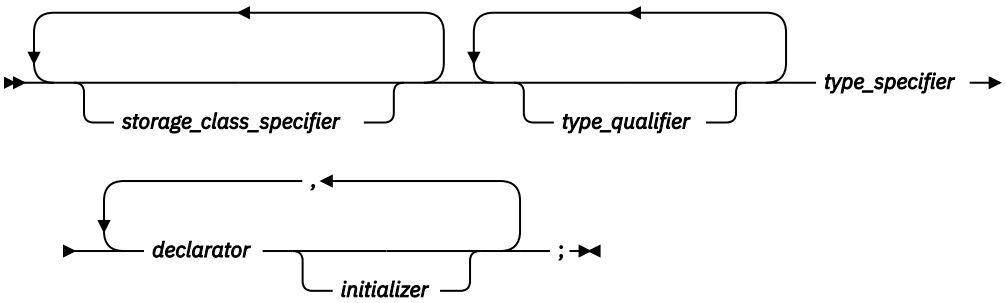
IBM i *Beginning of IBM Extension.*

In addition, ILE C/C++ allows you to use *attributes* to modify the properties of data objects. *Type* attributes, which can be used to modify the definition of user-defined types, are described in "Type attributes" on page 84. Variable attributes, which can be used to modify the declaration of variables, are described in "Variable attributes" on page 113.

IBM i *End of IBM Extension.*

All declarations have the form:

Data declaration syntax



Related information

- "Function declarations and definitions" on page 191

Tentative definitions

C *Beginning of 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 */
```

C *End of C only.*

C++ *Beginning of 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.

C++ *End of C++ only.*

static_assert declaration (C++11)

Note: IBM supports selected features of C++11, known as C++0x before its ratification. IBM will continue to develop and implement the features of this new standard. The implementation of the language level is based on IBM's interpretation of the standard. Until IBM's implementation of all the C++11 features are complete, including the support of a new C++11 standard library, the implementation may change from release to release. IBM makes no attempt to maintain compatibility, in source, binary, or listings and other compiler interfaces, with earlier releases of IBM's implementation of the new C++11 features and therefore they should not be relied on as a stable programming interface.

Static assertions can be declared to detect and diagnose common usage errors at compile time. A *static_assert declaration* declaration takes the following form:

A *static_assert declaration* takes the following form:

static_assert declaration syntax

The *constant-expression* must be a constant expression that can be contextually converted to `bool`. If the value of the expression converted in such a way is false, the compiler issues a severe error containing the *string literal* with the source location of the *static_assert declaration*. Otherwise, the *static_assert declaration* has no effect.

The *static_assert declaration* can appear anywhere that a using-declaration can, including namespace scope, block scope, and class member declaration lists.

The *static_assert declaration* does not declare a new type or object, and does not imply any size or time cost at run time.

The addition of static assertions to the C++ language has the following benefits:

- Libraries can detect common usage errors at compile time.
- Implementations of the C++ Standard Library can detect and diagnose common usage errors, improving usability.

You can use a static_assert declaration to check important program invariants at compile time.

Examples: static_assert declaration

static_assert in namespace scope:

```
    static_assert(sizeof(void *) == 8, "DTAMDL(*LLP64) is not allowed for this module.");
```

static_assert in class scope, with templates:

```
    #include <type_traits>
    #include <string>
    template<typename T>
    struct X {
        static_assert(std::tr1::is_pod<T>::value, "POD required to instantiate class template
X.");
        // ...
    };
    int main() {
        X<std::string> x;
```

```
    return 0;
}
```

static_assert in block scope, with templates:

```
template <typename T, int N>
void f() {
    static_assert (N >=0, "length of array a is negative.");
    T a[N];
    // ...
}
int main() {
    f<int, -1>();
    return 0;
}
```

An erroneous **static_assert** with an invalid constant expression:

```
static_assert(1 / 0, "never shows up!");
```

When this is compiled, instead of showing the string literal in the static_assert declaration, the compiler issues an error message indicating that the divisor must not be zero.

Related information

- "Extensions for C++11 compatibility" on page 411

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 runtime 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
- C++ mutable
- register
- IBM i __thread

C++11 *Beginning of C++11 only.*

In C++11, the keyword `auto` is no longer used as a storage class specifier. Instead, it is used as a type specifier. The compiler deduces the type of an `auto` variable from the type of its initializer expression. For more information, see "The auto type specifier (C++11)" on page 57.

The keyword `extern` was previously used as a storage specifier or as part of a linkage specification. The C++11 standard adds a third usage to use this keyword to specify explicit instantiation declarations. For more information, see "Explicit instantiation (C++ only)" on page 343.

C++11 *End of C++11 only.*

Related information

- “Function storage class specifiers” on page 197
- “Initializers” on page 100

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.

Note: C++11 In C++11, the keyword `auto` is no longer used as a storage class specifier. Instead, it is used as a type specifier. The compiler deduces the type of an auto variable from the type of its initializer expression. For more information, see “The auto type specifier (C++11)” on page 57.

Related information

- “Initialization and storage classes” on page 101
- “Block statements” on page 177
- “The goto statement” on page 189

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.

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.

C 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

Related information

- “The static storage class specifier” on page 197
- “Static members (C++ only)” on page 263

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 101
- “Internal linkage” on page 17
- “Namespaces (C++ only)” on page 223

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.

Note: C++0x The keyword `extern` was previously used as a storage specifier or as part of a linkage specification. The C++0x standard adds a third usage to use this keyword to specify explicit instantiation declarations. For more information, see "Explicit instantiation (C++ only)" on page 343.

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 17
- "Initialization and storage classes" on page 101
- "The extern storage class specifier" on page 197
- "Class scope (C++ only)" on page 14
- "Namespaces (C++ only)" on page 223

The mutable storage class specifier (C++ only)

C++ 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 79
- "References (C++ only)" on page 99

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:

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

C++ *Beginning of 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
```

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 101
- “Block/local scope” on page 12
- “References (C++ only)” on page 99

The __thread storage class specifier

IBM i *Beginning of IBM Extension.*

The `__thread` storage class marks a static variable as having *thread-local* storage duration. This means that in a multi-threaded application a unique instance of the variable is created for each thread that uses it and destroyed when the thread terminates. The `__thread` storage class specifier can provide a convenient way of assuring thread-safety; declaring an object as per-thread allows multiple threads

to access the object without the concern of race conditions while avoiding the need for low-level programming of thread synchronization or significant program restructuring.

Note: In order for the `__thread` keyword to be recognized, you must compile with the **LANGLVL(*EXTENDED)** option. See **LANGLVL** in the *ILE C/C++ Compiler Reference* for details.

The specifier can be applied to any of the following:

- global variables
- file-scoped static variables
- function-scoped static variables
- C++ static data members of a class

It cannot be applied to block-scoped automatic variables or non-static data members.

The specifier can appear on its own, or preceded immediately by either the `static` or `extern` specifier, as in the following examples:

```
__thread int i;
extern __thread struct state s;
static __thread char *p;
```

Variables marked with the `__thread` specifier can be initialized or uninitialized. C++ `__thread` variables must be initialized with a constant expression and must not have a static constructor.

Applying the address-of operator (&) to a thread-local variable returns the runtime address of the current thread's instance of the variable. That thread can pass this address to any other thread but when the first thread terminates any pointers to its thread-local variables become invalid.

Related information

- **LANGLVL** in the *ILE C/C++ Compiler Reference*

IBM i *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” on page 54
    - “Boolean types” on page 55
    - “Floating-point types” on page 55
    - “Character types ” on page 56
  - “The void type” on page 57
- “User-defined types” on page 65

C++ A type is a literal type if it satisfies one of the following conditions:

- It is a scalar type.
- It is a reference type.
- It is an array of literal type.
- C++11 It is a class type with all the following properties:
  - The class has a trivial destructor.

- Each constructor call and full expression in the initializers for nonstatic data members (if any) is a constant expression.
- The class is an aggregate type or has at least one `constexpr` constructor or constructor template that is not a copy or move constructor.
- All nonstatic data members and base classes of the class are of literal types.

C++11 In the C++11 standard, the following type specifiers are introduced:

- The auto type specifer
- The decltype(*expression*) type specifier

Related information

- "Function return type specifiers" on page 202

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

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++ *Beginning of 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.

C++ *End of C++ only.*

Related information

- "Integer literals" on page 25
- "Integral conversions" on page 118
- "Arithmetic conversions and promotions" on page 117
- "Overloading (C++ only)" on page 233

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.

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`.

C *Beginning of C only.*

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

C *End of C only.*

C++ *Beginning of 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`.

C++ *End of C++ only.*

Related information

- “Boolean literals” on page 28
- “Boolean conversions” on page 118

Floating-point types

Floating-point type specifiers fall into the following categories:

- “Real floating-point types” on page 55

Real floating-point types

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

- float
- double
- long double

IBM i *Beginning of IBM Extension.*

Decimal floating-point types consist of the following:

- _Decimal32
- _Decimal64
- _Decimal128

IBM i *End of IBM Extension.*

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

Table 12. Magnitude ranges of real floating-point types	
Type	**Range**
float	approximately 1.2^{-38} to 3.4^{38}
double, long double	approximately 2.2^{-308} to 1.8^{308}
_Decimal32	0.000001^{-95} to 9.999999^{96}
_Decimal64	0.000000000000001^{-383} to 9.99999999999999^{384}
_Decimal128	$0.00000000000000000000000000000001^{-6143}$ to $9.999999999999999999999999999999999^{6144}$

If a floating-point constant is too large or too small, the result is undefined by the language.

The declarator for a simple floating-point declaration is an identifier. Initialize a simple floating-point variable with a float constant or with a variable or expression that evaluates to an integer or floating-point number.

IBM i *Beginning of IBM Extension.*

You can use decimal floating-point types with any of the operators that are supported for binary floating-point types. You can also perform implicit or explicit conversions between decimal floating-point types and all other integral types or generic floating-point types. However, these are the restrictions on the use of decimal floating-point types with other arithmetic types:

- You cannot mix decimal floating-point types with generic floating-point types in arithmetic expressions, unless you use explicit conversions.
- Implicit conversion between decimal floating-point types and real binary floating-point types is only allowed via assignment, with the simple assignment operator =. Implicit conversion is performed in simple assignments, which also include function argument assignments and function return values.

IBM i *End of IBM Extension.*

Related information

- “Floating-point literals” on page 28
- “Floating-point conversions ” on page 118
- “Arithmetic conversions and promotions” on page 117

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 **DFTCHAR(*SIGNED|*UNSIGNED)** option or the **#pragma chars** directive. See **DFTCHAR(*SIGNED|*UNSIGNED)** in the *ILE C/C++ Compiler Reference* for more information.

Related information

- "Character literals" on page 32
- "String literals" on page 33
- "Arithmetic conversions and promotions" on page 117
- **DFTCHAR(*SIGNED|*UNSIGNED)** in the *ILE C/C++ Compiler Reference*

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 92
- "Comma operator ," on page 150
- "Conditional expressions" on page 152
- "Function declarations and definitions" on page 191

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.

The auto type specifier (C++11)

Note:

IBM supports selected features of C++11, known as C++0x before its ratification. IBM will continue to develop and implement the features of this standard. The implementation of the language level is based on IBM's interpretation of the standard. Until IBM's implementation of all the C++11 features are complete, including the support of a new C++ standard library, the implementation may change from release to release. IBM makes no attempt to maintain compatibility, in source, binary, or listings and other compiler interfaces, with earlier releases of IBM's implementation of the new features of the C++11 features and therefore they should not be relied on as a stable programming interface.

C++11 introduces the keyword `auto` as a new type specifier. `auto` acts as a placeholder for a type to be deduced from the initializer expression of a variable. With auto type deduction enabled, you no longer need to specify a type while declaring a variable. Instead, the compiler deduces the type of an auto variable from the type of its initializer expression.

The following examples demonstrate the usage of auto type deduction.

```
auto x = 1; //x : int
float* p;
auto x = p; //x : float*
auto* y = p; //y : float*
double f();
auto x = f(); //x : double
const auto& y = f(); //y : const double&
class R;
R* h();
auto* x = h(); //x : R*
auto y = h(); //y : R*
int& g();
auto x = g(); //x : int
const auto& y = g(); //y : const int&
auto* z = g(); //error, g() does not return a pointer type
```

By delegating the task of type deduction to the compiler, auto type deduction increases programming convenience, and potentially eliminates typing errors made by programmers. Auto type deduction also reduces the size and improves the readability of programs.

The following two examples demonstrate the benefits of enabling auto type deduction. The first example does not enable auto type deduction.

```
for (vector<T>::iterator i = vec.begin(); i < vec.end(); i++)
{
int* a = new int(1);
//...
}
```

With auto type deduction enabled, the first example can be simplified as follows:

```
for (auto i = vec.begin(); i < vec.end(); i++)
{
auto a = new auto(1);
//...
}
```

The following rules and constraints apply to the use of auto as a type specifier in auto type deduction.

- Auto type deduction cannot deduce array types.

  ```
  int x[5];
  auto y[5] = x;   //error, x decays to a pointer,
                   //which does not match the array type
  ```

- Auto type deduction cannot deduce *cv-qualifier* or reference type from the Initializer.

  ```
  int f();
  auto& x = f();   //error, cannot bind a non-const reference
                   //to a temporary variable
  int& g();
  auto y = g();    //y is of type int
  auto& z = g();   //z is of type int&
  ```

- Auto type deduction supports multi-variable auto declarations. If the list of declarators contains more than one declarator, the type of each declarator can be deduced independently. If the deduced type is not the same in each deduction, the program is ill-formed.

  ```
  auto x=3, y=1.2, *z=new auto(1); //error y: deduced as double,
                                   //but was previously deduced as int
  ```

- The name of the object that is declared can not be used in its initializer expression.

  ```
  auto x = x++; //error
  ```

- auto can not be used in function parameters.

```
int func(auto x = 3) //error
{
//...
}
```

Note: In C++11, the keyword `auto` is no longer used as a storage class specifier.

Related information

- “Storage class specifiers” on page 48
- “The auto storage class specifier” on page 49
- “Type qualifiers” on page 79
- “Extensions for C++11 compatibility” on page 411

The decltype(expression) type specifier (C++11)

Note: IBM supports selected features of C++11, known as C++0x before its ratification. IBM will continue to develop and implement the features of the this standard. The implementation of the language level is based on IBM's interpretation of the standard. Until IBM's implementation of all the C++11 features are complete, including the support of a new C++ standard library, the implementation may change from release to release. IBM makes no attempt to maintain compatibility, in source, binary, or listings and other compiler interfaces, with earlier releases of IBM's implementation of the new C++11 features and therefore they should not be relied on as a stable programming interface.

The *decltype(expression)* specifier is a type specifier introduced in C++11. With this type specifier, you can get a type that is based on the resultant type of a possibly type-dependent expression.

decltype(*expression*) takes *expression* as an operand. When you define a variable by using decltype(*expression*), it can be thought of as being replaced by the compiler with the type or the derived type of *expression*. Consider the following example:

```
int i;
static const decltype(i) j = 4;
```

In this example, `decltype(i)` is equivalent to the type name `int`.

General rules for using decltype

When you use decltype(*expression*) to get a type, the following rules are applicable:

- If *expression* is an unparenthesized id-expression or class member, decltype(*expression*) is the type of the entity named by expression. If there is no such entity, or if *expression* names a set of overloaded functions, the program is ill formed.
- Otherwise, if *expression* is a function call or an invocation of an overloaded operator (parentheses around *expression* are ignored), decltype(*expression*) is the return type of the statically chosen function.
- Otherwise, if *expression* is an lvalue, decltype(*expression*) is T&, where T is the type of *expression*.
- Otherwise, decltype(*expression*) is the type of *expression*.

The following example illustrates how these rules are used:

```
const int* bar(){
    return new int[0];
}
struct A{
    double x;
};
template <class T> T tFoo(const T& t){
    return t;
}
bool func(){
    return false;
}
struct Foo{
    template <typename T, typename U>
```

```
    static decltype((*(T*)0) * (*(U*)0)) foo(const U& arg1, const T& arg2){
    return arg1 * arg2;
    }
};
template <typename T, typename U> struct Bar{
    typedef decltype((*(T*)0) + (*(U*)0)) btype;
    static btype bar(T t, U u);
};
int main(){
    int i = 4;
    const int j = 6;
    const int& k = i;
    int a[5];
    int *p;
    decltype(i) var1;         // int
    decltype(1) var2;         // int
    decltype(2+3) var3;     // int(+ operator returns an rvalue)
    decltype(i=1) var4 = i; // int&, because assignment to int
                            // returns an lvalue
    decltype((i)) var5 = i; // int&
    decltype(j) var6 = 1; // const int
    decltype(k) var7 = j; // const int&
    decltype("decltype") var10 = "decltype"; // const char(&)[9]
    decltype(a) var8; // int[5]
    decltype(a[3]) var9 = i; // int&([] returns an lvalue)
    decltype(*p) var11 = i; // int&(*operator returns an lvalue)
    decltype(tFoo(A())) var12; // A
    decltype(func()) var13; // bool
    decltype((func())) var14; // bool, parentheses around f() are ignored
    decltype(func) var15; // bool()
    decltype(&func) var16; // bool(*)()
    decltype(&A::x) var17; // double A::*
    decltype(Foo::foo(3.0, 4u)) var18; // double
    decltype(Bar<float, short>::bar(1,3)) var19; // float
    return 0;
}
```

In this example, the comment after each decltype statement explains the type of the defined variable.

The following example illustrates an incorrect usage of decltype(expression):

```
int func(){
    return 0;
}
int func(int a){
    return 0;
}
int main(){
    int i = 4;
    // Incorrect usage. func names an overload function
    decltype(func) var1;
    // Correct usage. The overload operation is not ambiguous
    decltype(func(i)) var2;
    return 0;
}
```

In this example, the compiler issues an error message because it does not know which func function to match.

Rules for using decltype with structure member variables

When you use decltype(*expression*) to get a type, and *expression* is an unparenthesized member variable of an *object expression* (with a `.` operator) or a *pointer expression* (with a -> operator), the following rules apply:

- If the object expression or the pointer expression is specified with a constant or volatile qualifier, the type qualifier does not contribute to the result of decltype(*expression*).
- The lvalueness or rvalueness of the object expression or the pointer expression does not affect whether decltype(*expression*) is a reference type or not.

Example:

```
struct Foo{
    int x;
};
int main(){
```

```
    struct Foo f;
    const struct Foo g = {0};
    volatile struct Foo* h = &f;
    struct Foo func();
    decltype(g.x) var1;         // int
    decltype(h->x) var2;         // int
    decltype(func().x) var3;     // int
    return 0;
}
```

In this example, the constant qualifier of the object expression g is not desired in the result of `decltype(g.x)`. Similarly, the volatile qualifier of the pointer expression h is not desired in the result of `decltype(h->x)`. The object expression g and the pointer expression h are lvalues, and the object expression `func()` is a rvalue, but they do not affect whether the decltype results of their unparenthesized member variables are reference types or not.

If *expression* declared in decltype(*expression*) is a parenthesized structure member variable, the constant or volatile type qualifier of the parent object expression or pointer expression of *expression* contributes to the result of decltype(*expression*). Similarly, the lvalueness or rvalueness of the object expression or the pointer expression affects the result of decltype(*expression*).

Example:

```
struct Foo{
    int x;
};
int main(){
    int i = 1;
    struct Foo f;
    const struct Foo g = {0};
    volatile struct Foo* h = &f;
    struct Foo func();
    decltype((g.x)) var1 = i;             // const int&
    decltype((h->x)) var2 = i;          // volatile int&
    decltype((func().x)) var3 = 1;      // int
    return 0;
}
```

In this example, the result of `decltype((g.x))` inherits the constant qualifier of the object expression g. Similarly, the result of `decltype((h->x))` inherits the volatile qualifier of the pointer expression h. The object expression g and the pointer expression h are lvalues, so `decltype((g.x))` and `decltype((h->x))` are reference types. The object expression `func()` is a rvalue, so `decltype((func().x))` is a nonreference type.

If you use the built-in operators `.*` or `->*` within a decltype(*expression*), the constant or volatile type qualifier of the parent object expression or pointer expression of *expression* contributes to the result of decltype(*expression*), regardless of whether *expression* is a parenthesized or an unparenthesized structure member variable. Similarly, the lvalueness or rvalueness of the object expression or the pointer expression affects the result of decltype(*expression*).

Example:

```
class Foo{
    int x;
};
int main(){
    int i = 0;
    Foo f;
    const Foo & g = f;
    volatile Foo* h = &f;
    const Foo func();
    decltype(f.*&Foo::x) var1 = i; // int&, f is an lvalue
    decltype(g.*&Foo::x) var2 = i; // const int&, g is an lvalue
    decltype(h->*&Foo::x) var3 = i; // volatile int&, h is an lvalue
    decltype((h->*&Foo::x)) var4 = i; // volatile int&, h is an lvalue
    decltype(func().*&Foo::x) var5 = 1; // const int, func() is an rvalue
    decltype((func().*&Foo::x)) var6 = 1; // const int, func() is an rvalue
    return 0;
}
```

Side effects and decltype

If you use decltype(*expression*) to get a type, additional operations in the decltype parenthetical context can be performed, but they do not have side effects outside of the decltype context.

Consider the following example:

```
int i = 5;
static const decltype(i++) j = 4; // i is still 5
```

The variable `i` is not increased by 1 outside of the decltype context.

There are exceptions to this rule. In the following example, because the expression given to decltype must be valid, the compiler has to perform a template instantiation:

```
template <int N>
struct Foo{
static const int n=N;
};
int i;
decltype(Foo<101>::n,i) var = i; // int&
```

In this example, Foo template instantiation occurs, even though `var` is only determined by the type of the variable `i`.

Redundant qualifiers and specifiers with decltype

Because decltype(*expression*) is considered syntactically to be a type specifier, the following redundant qualifiers or specifiers are ignored:

- constant qualifiers
- volatile qualifiers
- & specifiers

The following example demonstrates this case:

```
int main(){
    int i = 5;
    int& j = i;
    const int k = 1;
    volatile int m = 1;
    // int&, the redundant & specifier is ignored
    decltype(j)& var1 = i;
    // const int, the redundant const qualifier is ignored
    const decltype(k) var2 = 1;
    // volatile int, the redundant volatile qualifer is ignored
    volatile decltype(m) var3;
    return 0;
}
```

Note: The functionality of ignoring the redundant & specifiers in decltype(*expression*) is not supported in the current C++11 standard, but it is implemented in this compiler release.

IBM i *Beginning of IBM Extension.*

`__ptr64` and `__ptr128`are pointer attribute specifiers which are used to specify the size of pointer type. If the *expression* declared in decltype(*expression*) is pointer type, its pointer attribute specifier is not ignored in the result of decltype(*expression*). If any pointer attribute is specified to decltype(*expression*), it results in duplicated pointer attributes on the same declaration and is diagnosed as an error.

Example:

```
int * ptr;
decltype(ptr) ptr1;        //ptr1 has the type "int *"
decltype(ptr) __ptr64 ptr2;    //error, __ptr64 is unexpected
int * __ptr64 ptr3;
decltype(ptr3) ptr4;        //ptr4 has the type "int * __ptr64"
decltype(ptr3) __ptr128 ptr5;    //error, __ptr128 is unexpected
```

IBM i *End of IBM Extension.*

Template dependent names and decltype

Without using the decltype feature, when you pass parameters from one function to another function, you might not know the exact types of the results that are passed back. The decltype feature provides a mechanism to generalize the return types easily. The following program shows a generic function that performs the multiplication operation on some operands:

```
struct Math{
    template <typename T>
    static T mult(const T& arg1, const T& arg2){
        return arg1 * arg2;
    }
};
```

If `arg1` and `arg2` are not the same type, the compiler cannot deduce the return type from the arguments. You can use the decltype feature to solve this problem, as shown in the following example:

```
struct Foo{
    template<typename T, typename U>
    static decltype((*(T*)0)*(*(U*)0)) mult(const T& arg1, const U& arg2)
    {
        return arg1 * arg2;
    }
};
```

In this example, the return type of the function is the type of the multiplication result of the two template-dependent function parameters.

The typeof operator and decltype

IBM i The decltype feature is similar to the existing *typeof* feature. One difference between these two features is that `decltype` only accepts an expression as its operand, while `typeof` can also accept a type name. Consider the following example:

```
__typeof__(int) var1;     // okay
decltype(int) var2;     // error
```

In this example, `int` is a type name, so it is invalid as the operand of decltype.

Note: `__typeof__` is an alternate spelling of typeof.

Related information

- “Keywords” on page 21
- “Name binding and dependent names (C++ only)” on page 362
- “Extensions for C++11 compatibility” on page 411
- Lvalues and rvalues
- References (C++ only)

The constexpr specifier (C++11)

Note: IBM supports selected features of C++11, known as C++0x before its ratification. IBM will continue to develop and implement the features of this standard. The implementation of the language level is based on IBM's interpretation of the standard. Until IBM's implementation of all the C++11 features is complete, including the support of a new C++11 standard library, the implementation may change from release to release. IBM makes no attempt to maintain compatibility, in source, binary, or listings and other compiler interfaces, with earlier releases of IBM's implementation of the new C++11 features.

The C++11 standard introduces a new keyword `constexpr` as a declaration specifier. You can apply the `constexpr` specifier only to the following contexts:

- The definition of a variable
- The declaration of a function or function template
- The declaration of a static data member

For example:

```
constexpr int i = 1;       // OK, definition
constexpr extern int j;    // Error, not a definition
constexpr int f1();        // OK, function declaration, but must be defined before use
```

If you declare a function that is not a constructor with a `constexpr` specifier, that function is a `constexpr` function. Similarly, if you declare a constructor with a `constexpr` specifier, that constructor is a `constexpr` constructor. Both `constexpr` functions and `constexpr` constructors are implicitly inline. For example:

```
struct S {
  constexpr S(int i) : mem(i) { }   // OK, declaration of a constexpr constructor
  private:
    int mem;
};
constexpr S s(55);      // OK, invocation of a constexpr constructor
```

If any declaration of a function or function template is specified with `constexpr`, all its declarations must contain the `constexpr` specifier. For example:

```
constexpr int f1();         // OK, function declaration
int f1() {                  // Error, the constexpr specifier is missing
  return 55;
}
```

Function parameters cannot be declared with the `constexpr` specifier. The following example demonstrates this:

```
constexpt int f4(constexpr int);      //Error
```

A `constexpr` specifier used in an object declaration declares the object as `const`. Such an object must be of a literal type and initialized. If it is initialized by a constructor call, that call must be a constant expression. Otherwise, if a `constexpr` specifier is used in a reference declaration, every full expression that appears in its initializer must be a constant expression. Each implicit conversion used in converting the initializer expressions and each constructor call used for the initialization must be valid in a constant expression. For example:

```
constexpr int var;         // Error, var is not initialized
constexpr int var1 = 1;  // OK

void func() {
  var1 = 5;   //Error, var1 is const
}

struct L {
  constexpr L() : mem(55) { }
  constexpr L(double d) : mem((int)d) { }
  L(int i) : mem(i) { }
  operator int() { return mem; }
private:
  int mem;
};

// Error, initializer involves a non-constexpr constructor.
constexpr L var2(55);

double var3 = 55;

// Error, initializer involves a constexpr constructor with non-constant argument
constexpr L var4(var3);

// Error, involves conversion that uses a non-constexpr conversion function
constexpr int var5 = L();
```

A `constexpr` specifier for a nonstatic member function that is not a constructor declares that member function to be `const`. The class of that `constexpr` member function must be a literal type. In the following example, the class `NL` is a non-literal type because it has a user-provided destructor.

```
struct NL {
  constexpr int f(){       //error, enclosing class is not a literal type
     return 55;
  }
  ~NL() { }
};
```

A call to a `constexpr` function produces the same result as a call to an equivalent non-`constexpr` function, except that a call to a `constexpr` function can appear in a constant expression.

The `main` function cannot be declared with the `constexpr` specifier.

Related information

- “Literals” on page 24
- “Constexpr functions (C++11)” on page 220
- “Constexpr constructors (C++11)” on page 303
- “Generalized constant expressions (C++11)” on page 131

User-defined types

The following are user-defined types:

- “Structures and unions” on page 65
- “Enumerations” on page 72
- “typedef definitions” on page 77
- C++ Classes
- C++ Elaborated type specifiers

C++ classes are discussed in “Classes (C++ only)” on page 247. Elaborated type specifiers are discussed in “Scope of class names (C++ only)” on page 250.

Related information

- “Type attributes” on page 84

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.

C++ 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 66 and “Structure and union variable declarations” on page 69; 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 70.

Structures and unions are subject to alignment considerations. For a complete discussion of alignment, see "Aligning data" in the *ILE C/C++ Programmer's Guide*.

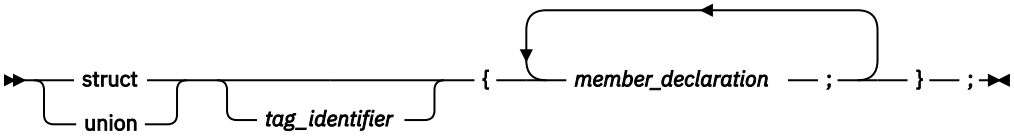
Related information

- “Classes and structures (C++ only)” on page 249

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 70. 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.

Related information

- "The aligned type attribute" on page 85
- "The packed type attribute" on page 85

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. 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 " on page 67.

IBM i As an extension to Standard C and C++ , ILE C/C++ also allows zero-extent arrays as members of structures and unions, as described in "Zero-extent array members" on page 67.

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

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.

IBM i *Beginning of IBM Extension.*

ILE C/C++ extends Standard C and C++ to ease the restrictions on flexible arrays and allow the following:

- Flexible array members can be declared in any part of a structure, not just as the last member.

  C++ The type of any member that follows the flexible array member must be compatible with the type of the flexible array member.

  C The type of any member following the flexible array member is not required to be compatible with the type of the flexible array member; however, a warning is issued in this case.
- Structures containing flexible array members can be members of other structures.
- Flexible array members can be statically initialized.

In the following example:

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

struct Foo foo1 = { 55, {6, 8, 10} };
struct Foo foo2 = { 55, {15, 6, 14, 90} };
```

`foo1` creates an array b of 3 elements, which are initialized to 6, 8, and 10; while `foo2` creates an array of 4 elements, which are initialized to 15, 6, 14, and 90.

Flexible array members can only be initialized if they are contained in the outermost part of nested structures. Members of inner structures cannot be initialized.

IBM i *End of IBM Extension.*

Related information

- “Variable length arrays” on page 98

Zero-extent array members

IBM i *Beginning of IBM Extension.*

A zero-extent array is an array with no dimensions. Like a flexible array member, a zero-extent array can be used to access a variable-length object.

A zero-extent array must be explicitly declared with zero as its dimension:

```
array_identifier[0]
```

Like a flexible array member, a zero-extent array can be declared in any part of a structure, not just as the last member.

The type of any member following the zero-extent array is not required to be compatible with the type of the zero-extent array; however, a warning is issued in this case.

Unlike a flexible array member, a structure containing a zero-extent array can be a member of another array. Also, the `sizeof` operator can be applied to a zero-extent array; the value returned is 0.

A zero-extent array can only be statically initialized with an empty set. For example:

```
struct foo {
   int a;
   char b[0];
} bar = { 100, { } };
```

Otherwise, it must be initialized as a dyamically-allocated array.

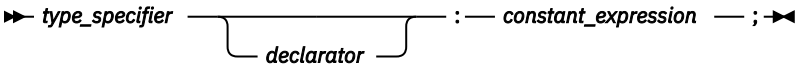
Zero-extent array members can only be initialized if they are contained in the outermost part of nested structures. Members of inner structures cannot be initialized.

IBM i *End of IBM Extension.*

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



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 *Beginning of 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`.

C *End of C only.*

C++ *Beginning of C++ only.*

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

C++ *End of C++ only.*

The maximum bit-field length is 64 bits. To increase portability, do not use bit fields greater than 32 bits in size.

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.

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

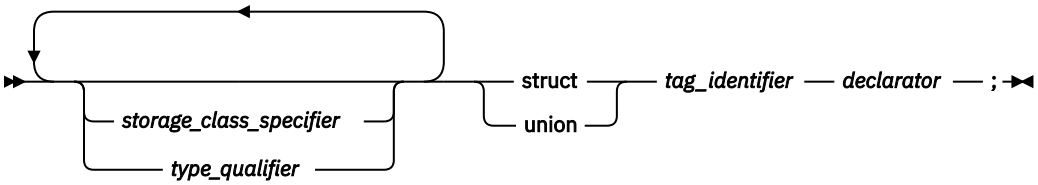
Related information

- "Alignment of bit fields" in the *ILE C/C++ Programmer's Guide*

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 114
- "The __align type qualifier" on page 80
- "The packed variable attribute" on page 115
- "Initialization of structures and unions" on page 104
- "Compatibility of structures, unions, and enumerations (C only)" on page 77
- "Dot operator ." on page 132
- "Arrow operator ->" on page 133

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 104
- “Storage class specifiers” on page 48
- “Type qualifiers” on page 79

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 132
- “Arrow operator ->” on page 133

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.

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
}
```

C++ 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 49
- "Member functions (C++ only)" on page 256

Enumerations

An *enumeration* is a data type consisting of a set of named values that represent integral constants, known as *enumeration constants*. An enumeration is 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 72 and "Enumeration variable declarations" on page 76; 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 76.

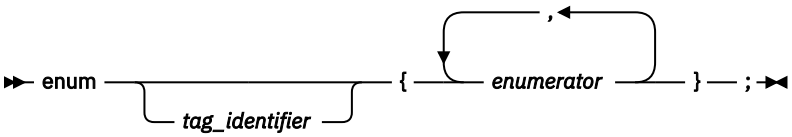
Related information

- "Arithmetic conversions and promotions" on page 117

Enumeration type definition

An enumeration type definition contains the `enum`, C++11 `enum class`, or `enum struct` keyword followed by an optional identifier (the enumeration tag), C++11 an optional underlying type specifier, and a brace-enclosed list of enumerators. C++11 For scoped enumerations, the identifier is mandatory. A comma separates each enumerator in the enumerator list. C99 allows a trailing comma between the last enumerator and the closing brace. C++ also supports this feature, for compatibility with C99.

Enumeration definition syntax



C++11

```
Unscoped enumeration definition syntax

>>-enum--+-+----------------+--+------------------+--------------------+-->
         | '-tag_identifier-'  '-:--type_specifier-'                    |
         '-nested_name_specifier--::--tag_identifier--:--type_specifier-'

      .-,----------.
      V            |
>--{----enumerator-+--}--;-----------------------------------><

Scoped enumeration definition syntax

>>-+-enum class--+--+--------------------------+--tag_identifier-->
   '-enum struct-'  '-nested_name_specifier--::-'

                            .-,----------.
                            V            |
```

```
>--+--------------------+--{----enumerator-+--}--;-------------><
   '-:--type_specifier-'
```

Enumeration forward declaration syntax

```
>>-+-enum--tag_identifier--:--type_specifier----------------+--->
   '-+-enum class--+--tag_identifier--+-------------------+-'
     '-enum struct-'                  '-:--type_specifier-'

>--;-------------------------------------------------------------><
```

The keyword `enum` declares an unscoped enumeration type, whose enumerators are unscoped enumerators. The keywords `enum class` or `enum struct` declare a scoped enumeration type, whose enumerators are scoped enumerators. The keywords `enum class` and `enum struct` are semantically equivalent.

For the unscoped enumeration, each enumerator is declared both in the scope of the enumeration specifier, and the scope that immediately contains the enumeration specifier.

For the scoped enumeration, each enumerator is declared in the scope of the enumeration specifier. In addition, the scoped enumeration does not allow the value of an enumerator (or an object of an enumeration type) to be converted to an integer by integral promotion, but explicit cast to `int`, `char`, `float`, or any other scalar type is allowed. For example:

```
enum class Colour { orange, green, purple };
enum class Fruit { apple, pear, grape, orange };

void functionx(int) {}
void functionx(Colour) {}
void functionx(Fruit) {}

int main()
{
  functionx(green);            // error, green is not introduced into
                               // the global scope
  functionx(Colour::green);    // calls functionx(Colour)
  functionx(Fruit::apple);     // calls functionx(Fruit)
  int i = Colour::orange;      // error, no conversion from Colour to int
  int j = (int)Colour::green;  // valid, explicit cast to integer is allowed
}
```

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 76. 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.

C++11 The *type_specifier* specifies the underlying type of an enumeration explicitly. Each enumeration has an underlying type. The *type_specifier* can be any integer type. Note that the enumeration type is not an integer type. By specifying the underlying type of an enumeration, you can ensure program correctness and use the enumeration reliably in structs. If *type_specifier* is not specified, the underlying type of the enumeration is determined as follows:

- If the enumeration is scoped, the underlying type has the fixed type `int`.
- If the enumeration is unscoped, the underlying type is an integral type that can represent all the enumerator values defined in the enumeration.

For example:

```
// the underlying type is "unsigned int"
enum Colour : unsigned int { orange = 0, green = 1, purple = 0x02U };

// the underlying type is "unsigned long"
enum class E : unsigned long { E1 = 0, E2 = 1, Emax = 0xFFFFFFF0U };

// if the type_specifier is not specified, the underlying type
// is "int" for the scoped enumeration
enum class A { A1, A2, A3 = 100, A4 };

// the underlying type is an integer type enough to represent the given
```

```
// enumerator values, eg. [signed/unsigned] long long
enum A { A1, A2, A3 = 10000000000000000000, A4 };
```

The *nested_name_specifier* refers to the class or namespace where the enumeration was previously declared. For example:

```
class C{

//Declaration of a scoped enumeration A.
enum class A;

};

// C is the nested_name_specifier that refers to the class where
// the enumeration A was declared.
enum class C::A { a, b };
```

You can forward declare an enumeration without providing a list of enumerators. The forward declaration of an enumeration can reduce compile time and dependencies, because it can physically decouple the implementation specifics of the enumeration and its usage.

The forward declaration of an enumeration is a redeclaration of an enumeration in the current scope or a declaration of a new enumeration. The redeclaration of the same enumeration must match the previous declaration. The rules for the redeclaration of enumerations are as follows:

- A scoped enumeration cannot be later redeclared as unscoped or with a different underlying type.
- An unscoped enumeration cannot be later redeclared as scoped and each redeclaration must include a *type_specifier* specifying the same underlying type.

For example:

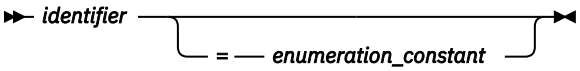
```
enum E1 : unsigned long;        // valid, forward declaration of an unscoped
                                // enumeration with the underlying type
                                // "unsigned long"

enum class E2 : char;           // valid, forward declaration of a scoped
                                // enumeration with the underlying type
                                // "char"

enum class E3;                  // valid, forward declaration of a scoped
                                // enumeration with the implied underlying
                                // type "int"

enum E4;                        // error, you must specify the underlying type
                                // when you forward declare an unscoped
                                // enumeration

enum E1 : unsigned long;        // valid, the redeclaration of E1 matches the
                                // previous declaration of E1

enum class E2 : short;          // error, the previously declared enumeration
                                // E2 had the underlying type "char", and it
                                // cannot be redeclared with a different
                                // underlying type

enum E3 : int;                  // error, the previously declared enumeration
                                // E3 was a scoped enumeration, and it cannot
                                // be redeclared as an unscoped enumeration

enum class E3 : int { a, b, c };// valid, definition of E3
```

C++ C++ supports a trailing comma in the enumerator list.

Enumeration members

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

Enumeration member declaration syntax



C 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`).

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

C++11 *Beginning of C++11 only.*

The underlying type of the following enumerations is fixed. The enumerator values obtained by following the previous rules must be representable by the underlying type. Otherwise the enumeration is ill-formed.

- Scoped enumerations
- Unscoped enumerations with explicit underlying type specified

The underlying type of an unscoped enumeration, with no explicit underlying type, is not fixed, and is determined by enumerator values as follows:

- The underlying type is an implementation-defined integral type that is not larger than `int` and can represent all values of the enumeration.
- If the values of the enumeration are not representable by an `int` or `unsigned int`, the underlying type is an implementation-defined integral type large enough to represent all enumerator values.
- If no type can represent all enumerator values, the enumeration is ill-formed.

C++11 *End of C++11 only.*

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;
}
```

C++11 With scoped enumerations, you can avoid such an error, because the enumerators are declared only in the scope of the enumeration, not the scope containing the enumeration.

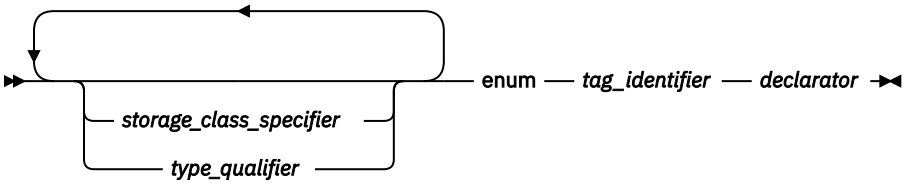
Related information

- “Integral types” on page 54

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.

C++ The keyword enum is optional in enumeration variable declarations.

Related information

- “Initialization of enumerations” on page 106
- “Compatibility of structures, unions, and enumerations (C only)” on page 77

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++ *Beginning of 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;
```

C++ *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 cannot 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.

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 full discussion of the **ENUM** compiler option and related pragmas, see **ENUM** and **#pragma enum** in the *ILE C/C++ Compiler Reference*.

Related information

- “Arithmetic conversions and promotions” on page 117
- “Classes (C++ only)” on page 247
- “Structure and union type definition” on page 66
- “Incomplete types” on page 44

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 runtime 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 91
- "Type specifiers" on page 53
- "Structures and unions" on page 65
- "Classes (C++ only)" on page 247
- "Friends (C++ only)" on page 269

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++ *Beginning of 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.

C++ *End of C++ only.*

C++0x *Beginning of C++0x only.*

Declaring typedef names as friends

In the C++0x standard, the extended friend declarations feature is introduced, with which you can declare typedef names as friends. For more information, see “Extended friend declarations” on page 270.

C++0x *End of C++0x 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

ILE C/C++ recognizes the following type qualifiers:

- IBM i `__align`
- `const`
- `restrict`
- `volatile`

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
`const int * ptr1;`	Defines a pointer to a constant integer: the value pointed to cannot be changed.
`int * const ptr2;`	Defines a constant pointer to an integer: the integer can be changed, but `ptr2` cannot point to anything else.
`const int * const ptr3;`	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, union, or class variable, they also apply to the members of the structure, class or union.

Related information

- “Pointers” on page 92
- “Constant and volatile member functions” on page 258

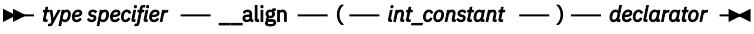
The __align type qualifier

IBM i *Beginning of IBM Extension.*

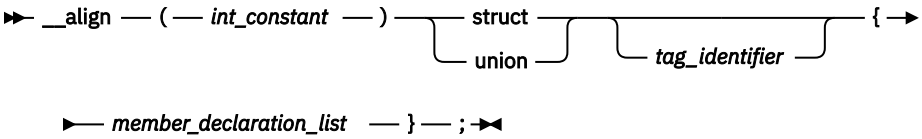
The `__align` qualifier is a language extension that allows you to specify an explicit alignment for an aggregate or a static (or global) variable. The specified byte boundary affects the alignment of an aggregate as a whole, not that of its members. The `__align` qualifier can be applied to an aggregate definition nested within another aggregate definition, but not to individual elements of an aggregate. The alignment specification is ignored for parameters and automatic variables.

A declaration takes one of the following forms:

__align qualifier syntax for simple variables



__align qualifier syntax for structures or unions



where *int_constant* is a positive integer value indicating the byte-alignment boundary. The legal values are 1, 2, 4, 8, or any other positive power of two.

The following restrictions and limitations apply:

- The `__align` qualifier cannot be used where the size of the variable alignment is smaller than the size of the type alignment.
- Not all alignments may be representable in an object file.
- The `__align` qualifier cannot be applied to the following:
  - Individual elements within an aggregate definition.
  - Individual elements of an array.
  - Variables of incomplete type.
  - Aggregates declared but not defined.
  - Other types of declarations or definitions, such as a typedef, a function, or an enumeration.

IBM i *End of IBM Extension.*

Examples using the __align qualifier

IBM i *Beginning of IBM Extension.*

Applying **__align** to static or global variables:

```
int __align(1024) varA;        /* varA is aligned on a 1024-byte boundary  */
main()                         /* and padded with 1020 bytes               */
{...}
```

```
static int __align(512) varB;  /* varB is aligned on a 512-byte boundary   */
                               /* and padded with 508 bytes                */
```

```
int __align(128) functionB( ); /* An error                                 */
```

```
typedef int __align(128) T;    /* An error                                 */
```

```
__align enum C {a, b, c};      /* An error                                 */
```

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

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

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

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

```
__align(128) struct S {int i;};     /* sizeof(struct S) == 128             */

struct S sarray[10];                /* sarray is aligned on 128-byte boundary
                                       with sizeof(sarray) == 1280         */

struct S __align(64) svar;          /* error - alignment of variable is
                                         smaller than alignment of type    */

struct S2 {struct S s1; int a;} s2; /* s2 is aligned on 128-byte boundary
                                         with sizeof(s2) == 256            */
```

Applying **__align** to an array:

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

Applying **__align** where the size of the variable alignment differs from the size of the type alignment:

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

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

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

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

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

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

Related information

- “The aligned variable attribute” on page 114
- “The __alignof__ operator” on page 138
- "Aligning data" in the *ILE C/C++ Programmer's Guide*

IBM i *End of IBM Extension.*

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 *Beginning of 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 *End of C only.*

C++ *Beginning of 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.

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 385
- “The this pointer (C++ only)” on page 261

The restrict type qualifier(C++ only)

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__`) 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.

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.

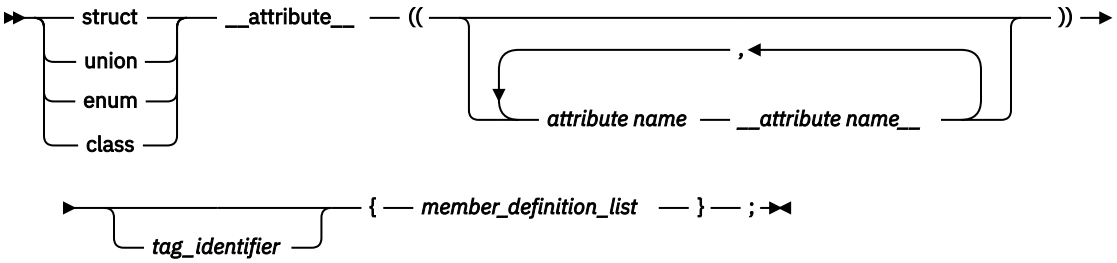
Type attributes

IBM i *Beginning of IBM Extension.*

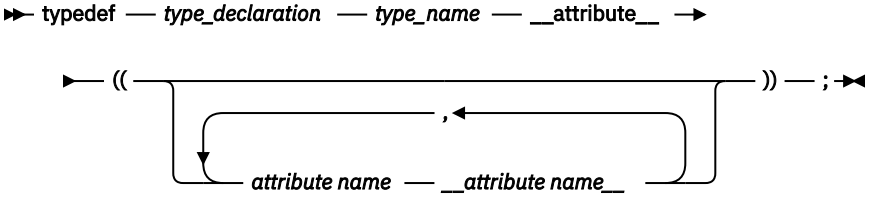
Type attributes are language extensions. These language features allow you to use named attributes to specify special properties of data objects. *Type* attributes apply to the definitions of user-defined types, such as structures, unions, enumerations, classes, and `typedef` definitions. Any variables 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



The *attribute name* can be specified with or without double underscore characters leading and trailing; however, using the double underscore reduces the likelihood of a name conflict with a macro of the

same name. For unsupported attribute names, the compiler issues diagnostics and ignores the attribute specification. Multiple attribute names can be specified in the same attribute specification.

The following type attributes are supported:

- "The aligned type attribute" on page 85
- "The packed type attribute" on page 85
- "The transparent_union type attribute (C only)" on page 86

Related information

- "Variable attributes" on page 113
- "Function attributes" on page 209

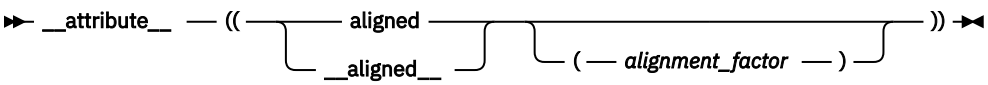
IBM i *End of IBM Extension.*

The aligned type attribute

IBM i *Beginning of IBM Extension.*

The `aligned` type attribute allows you to override the default alignment mode to specify a minimum alignment value, expressed as a number of bytes, for a structure, class, union, enumeration, or other user-defined type created in a `typedef` declaration. The `aligned` attribute is typically used to increase the alignment of any variables declared of the type to which the attribute applies.

aligned type attribute syntax



The *alignment_factor* is the number of bytes, specified as a constant expression that evaluates to a positive power of 2. You can specify a value up to a maximum 1048576 bytes. If you omit the alignment factor (and its enclosing parentheses), the compiler automatically uses 16 bytes. 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.

The alignment value that you specify will be applied to all instances of the type. Also, the alignment value applies to the variable as a whole; if the variable is an aggregate, the alignment value applies to the aggregate as a whole, not to the individual members of the aggregate.

In all of the following examples, the `aligned` attribute is applied to the structure type A. Because a is declared as a variable of type A, it will also receive the alignment specification, as will any other instances declared of type A.

```
struct __attribute__((__aligned__(8))) A {};
struct __attribute__((__aligned__(8))) A {} a;
typedef struct __attribute__((__aligned__(8))) A {} a;
```

Related information

- "The __align type qualifier" on page 80
- "The aligned variable attribute" on page 114
- "The __alignof__ operator" on page 138
- "Aligning data" in the *ILE C/C++ Programmer's Guide*

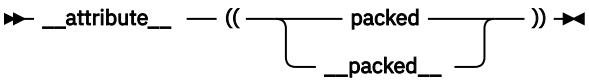
IBM i *End of IBM Extension.*

The packed type attribute

IBM i *Beginning of IBM Extension.*

The packed type attribute specifies that the minimum alignment should be used for the members of a structure, class, union, or enumeration type. For structure, class, or union types, the alignment is one byte for a member and one bit for a bit field member. For enumeration types, the alignment is the smallest size that will accomodate the range of values in the enumeration. All members of all instances of that type will use the minimum alignment.

packed type attribute syntax



Unlike the aligned type attribute, the packed type attribute is not allowed in a typedef declaration.

Related information

- “The __align type qualifier” on page 80
- “The packed variable attribute” on page 115
- “The __alignof__ operator” on page 138
- "Aligning data" in the *ILE C/C++ Programmer's Guide*

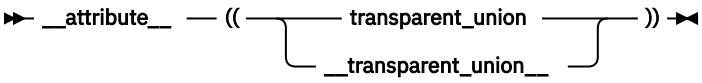
IBM i *End of IBM Extension.*

The transparent_union type attribute (C only)

IBM i *Beginning of IBM Extension.*

The transparent_union attribute applied to a union definition or a union typedef definition indicates the union can be used as a *transparent union*. Whenever a transparent union is the type of a function parameter and that function is called, the transparent union can accept an argument of any type that matches that of one of its members without an explicit cast. Arguments to this function parameter are passed to the transparent union, using the calling convention of the first member of the union type. Because of this, all members of the union must have the same machine representation. Transparent unions are useful in library functions that use multiple interfaces to resolve issues of compatibility.

transparent_union type attribute syntax



The union must be a complete union type. The transparent_union type attribute can be applied to anonymous unions with tag names.

When the transparent_union type attribute is applied to the outer union of a nested union, the size of the inner union (that is, its largest member) is used to determine if it has the same machine representation as the other members of the outer union. For example,

```
union __attribute__((__transparent_union__)) u_t {
   union u2_t {
      char a;
      short b;
      char c;
      char d;
   };
   int a;
};
```

the attribute is ignored because the first member of union u_t, which is itself a union, has a machine representation of 2 bytes, whereas the other member of union u_t is of type int, which has a machine representation of 4 bytes.

The same rationale applies to members of a union that are structures. When a member of a union to which type attribute transparent_union has been applied is a struct, the machine representation of the entire struct is considered, rather than members.

All members of the union must have the same machine representation as the first member of the union. This means that all members must be representable by the same amount of memory as the first member of the union. The machine representation of the first member represents the maximum memory size for any remaining union members. For instance, if the first member of a union to which type attribute `transparent_union` has been applied is of type `int`, then all following members must be representable by at most 4 bytes. Members that are representable by 1, 2, or 4 bytes are considered valid for this transparent union.

Floating-point types (`float`, `double`, `float _Complex`, or `double _Complex`) types can be members of a transparent union, but they cannot be the first member. The restriction that all members of the transparent union have the same machine representation as the first member still applies.

IBM i *End of IBM Extension.*

Declarators

This section continues the discussion of data declarations and includes the following topics:

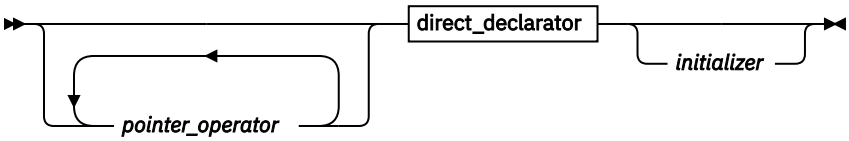
- "Overview of declarators" on page 89
- "Type names" on page 91
- "Pointers" on page 92
- "Arrays" on page 97
- "References (C++ only)" on page 99
- "Initializers" on page 100
- "Variable attributes" on page 113

Overview of declarators

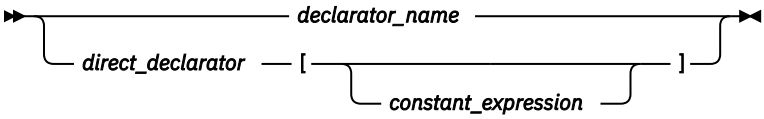
A *declarator* designates a data object or function. A declarator can also include an initialization. Declarators appear in most data definitions and declarations and in some type definitions.

For data declarations, a declarator has the form:

Declarator syntax

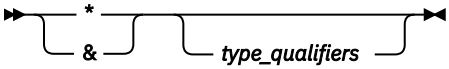


Direct declarator



C *Beginning of C only.*

Pointer operator (C only)

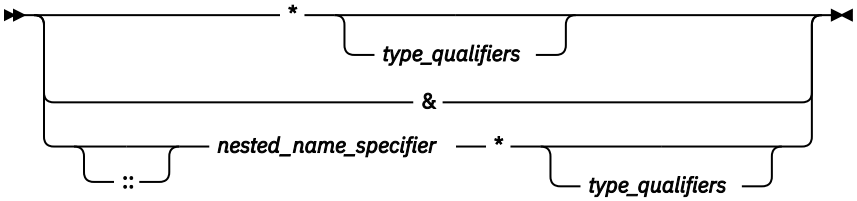


Declarator name (C only)

C *End of C only.*

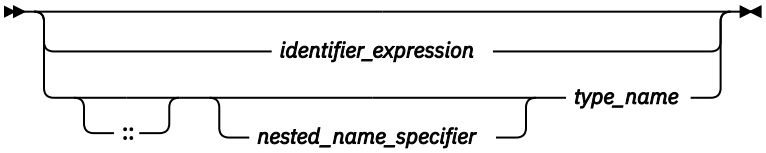
C++ *Beginning of C++ only.*

Pointer operator (C++ only)



Declarator name (C++ only)

© Copyright IBM Corp. 1993, 2015



C++ *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 100.

For the details of function declarators, see "Function declarators" on page 204. C++11 For the details of trailing return types, see "Trailing return type (C++11)" on page 207.

The following are known as *derived declarator* types, and are therefore discussed in this section:

- "Pointers" on page 92
- "Arrays" on page 97
- "References (C++ only)" on page 99

IBM i In addition, for compatibility with GNU C and C++, ILE 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 113.

Related information

- "Type qualifiers" on page 79

Examples of declarators

The following table indicates the declarators within the declarations:

Declaration	Declarator	Description
`int owner;`	`owner`	`owner` is an integer data object.
`int *node;`	`*node`	`node` is a pointer to an integer data object.
`int names[126];`	`names[126]`	`names` is an array of 126 integer elements.
`volatile int min;`	`min`	`min` is a volatile integer.
`int * volatile volume;`	`* volatile volume`	`volume` is a volatile pointer to an integer.
`volatile int * next;`	`*next`	`next` is a pointer to a volatile integer.
`volatile int * sequence[5];`	`*sequence[5]`	`sequence` is an array of five pointers to volatile integer data objects.
`extern const volatile int clock;`	`clock`	`clock` is a constant and volatile integer with static storage duration and external linkage.

Related information

- "Type qualifiers" on page 79
- "Array subscripting operator [ ]" on page 149
- "Scope resolution operator :: (C++ only)" on page 130
- "Function declarators" on page 204

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 13. Type names

Syntax	Description
`int *[5]`	array of 5 pointers to `int`
`int (*)[5]`	pointer to an array of 5 integers
`int (*)[*]`	pointer to an variable length array of an unspecified number of integers
`int *()`	function with no parameter specification returning a pointer to `int`
`int (*)(void)`	function with no parameters returning an `int`
`int (*const [])(unsigned int, ...)`	array of an unspecified number of constant pointers to functions returning an `int`. Each function takes one parameter of type `unsigned int` 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 167

- “Examples of expressions and precedence” on page 170
- “The typename keyword (C++ only)” on page 363
- “Parenthesized expressions ( )” on page 129

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.

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-qualifed pointer.

The following table provides examples of pointer declarations.

Table 14. Pointer declarations

Declaration	Description
`long *pcoat;`	`pcoat` is a pointer to an object having type `long`
`extern short * const pvolt;`	`pvolt` is a constant pointer to an object having type `short`
`extern int volatile *pnut;`	`pnut` is a pointer to an `int` object having the `volatile` qualifier
`float * volatile psoup;`	`psoup` is a `volatile` pointer to an object having type `float`
`enum bird *pfowl;`	`pfowl` is a pointer to an enumeration object of type `bird`
`char (*pvish)(void);`	`pvish` is a pointer to a function that takes no parameters and returns a `char`
C++11 `nullptr_t pnull;`	`pnull` is a null pointer that does not point to any valid object or function.

Related information

- “Type qualifiers” on page 79
- “Initialization of pointers” on page 107
- “Compatibility of pointers (C only)” on page 94
- “Pointer conversions” on page 120
- “Address operator &” on page 135
- “Indirection operator *” on page 136
- “Pointers to functions” on page 219

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 134
- “Arrays” on page 97
- “Decrement operator --” on page 134
- “Expressions and operators” on page 125

Type-based aliasing

The compiler follows the type-based aliasing rule in the C and C++ standards when the **ALIAS(*ANSI)** 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

[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

practice of casting a pointer to an incompatible type and then dereferencing it violates this rule. (Note that `char` pointers are an exception to this rule.)

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 158
- **`ALIAS(*ANSI)`** in the *ILE C/C++ Compiler Reference*

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);
```

- a character type

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 */
```

Null pointers

A null pointer has a reserved value that is called a null pointer constant for indicating that the pointer does not point to any valid object or function. You can use null pointers in the following cases:

- Initialize pointers.
- Represent conditions such as the end of a list of unknown length.
- Indicate errors in returning a pointer from a function.

A null pointer constant is an integer constant expression that evaluates to zero. For example, a null pointer constant can be 0, `0L`, or such an expression that can be cast to type `(void *)0`. C++11 C++11 defines a new null pointer constant `nullptr` that can only be converted to any pointer type, pointer-to-member type, or bool type.

You can specify any of the following values for a null pointer constant:

- 0
- NULL
- C++11 nullptr

Note: NULL is a macro. It must be defined before use.

Null pointer constants

0

You can use an integer constant expression with the value 0 or an expression that is cast to`(void *)0` as a null pointer constant.

NULL

The macro NULL and value 0 are equivalent as null pointer constants, but NULL is cleaner because it represents the purpose of using the constant for a pointer.

C++11 *Beginning of C++11 only.*

nullptr

`nullptr` is an explicit null pointer constant. In C++, initializing null pointers with 0 or NULL have the following problems:

- It is impossible to distinguish between a null pointer and integer 0 for overloaded functions. For example, given two overloaded functions `f(int)` and`f(char*)`, the call `f(0)` resolves to `f(int)` because `0` is converted to the integral type instead of the pointer type. See Example1.
- A null pointer constant does not have a type-safe name. The macro NULL cannot be distinguished from the `0` constant for overloaded functions and error detection.

To solve the problems of null pointer constants, C++11 introduced a new keyword `nullptr`. The `nullptr` constant can be distinguished from integer 0 for overloaded functions. See Example 2.

A null pointer constant with the `nullptr` value has the following characteristics:

- It can be converted to any pointer or pointer-to-member type.
- It cannot be implicitly converted to any other type, except for the bool type.
- It cannot be used in an arithmetic expression.
- It can be compared with the integer 0.

- It can be used in relational expressions to compare with pointers or data of the `std::nullptr_t` type.

See Example 3 for information about how to use `nullptr` as an initializer for all pointer types and in comparison expressions.

Note: It is still acceptable to assign the value 0 or NULL to a pointer.

The `nullptr` keyword designates a constant rvalue of type `decltype(nullptr)`. The typedef expression is `typedef decltype(nullptr) nullptr_t`, of which `nullptr_t` is a typedef for `decltype(nullptr)` that is defined in <cstddef>. A non-type template parameter and argument can have the `std::nullptr_t` type. If a non-type template parameter is of type `std::nullptr_t`, the corresponding argument must be of type `std::nullptr_t`. See Example 4. If a non-type template parameter is of one of the following types, the type of the corresponding non-type template argument can be `std::nullptr_t`. The null pointer conversion occurs for the last three types:

- `std::nullptr_t`
- pointer
- pointer-to-member
- bool

When you use `nullptr` in exception handling, pay attention to the `throw` and `catch` arguments. A handler is a match for an exception object of type E if the handler is of type `cv T` or `const T&`. T is a pointer or pointer-to-member type and E is of type `std::nullptr_t`. See Example 5.

Examples

Example 1. This example illustrates inappropriate use of the NULL constant for overloaded functions:

```
#include <stdio.h>

void func(int* i){
   printf("func(int*)\n");
}

void func(int i){
   printf("func(int)\n");
}

int main(){
   func(NULL);
}
```

Suppose you want the main function to call `func(int* i)`. As a result, the `main` function calls `func(int i)` instead of `func(int* i)` because the constant NULL is equal to the integer 0. Constant 0 is implicitly converted to `(void*)0`, only when `func(int i)` does not exist.

Example 2. This example illustrates how `nullptr` is used in overloading functions:

```
void f( char* );
void f( int );
f( nullptr );     // calls f( char* )
f( 0 );           // calls f( int )
```

Example 3. The following expressions illustrate the correct and incorrect use of the `nullptr` constant:

```
char* p = nullptr;     // p has the null pointer value
char* p1 = 0;          // p has the null pointer value
int a = nullptr;       // error
int a2 = 0;            // a2 is zero of integral type
if( p == 0 );          // evaluates to true
if( p == nullptr );    // evaluates to true
if( p );               // evaluates to false
if( a2 == 0 );         // evaluates to true
if( a2 == nullptr );   // error, no conversion
if( nullptr );         // OK, conversion to the bool type
if( nullptr == 0 );    // OK, comparison with 0
nullptr = 0;           // error, nullptr is not an lvalue
nullptr + 2;           // error
```

Example 4. This example illustrates that a non-type template parameter or argument can have the `std::nullptr_t` type:

```
typedef decltype(nullptr) nullptr_t;
template <nullptr_t> void fun1(); // non-type template parameter

fun1<nullptr>();      // non-type template arguments
fun1<0>();            // error. The corresponding argument must be of type
                      // std::nullptr_t if the parameter is of type std::nullptr_t.

template <int* p> void fun2();
fun2<nullptr>();      //Correct

template<typename T> void h( T t );
h( 0 );                    // deduces T = int
h( nullptr );              // deduces T = nullptr_t
h( (float*) nullptr );     // deduces T = float*
```

Example 5. This example illustrates how to use `nullptr` in exception handling:

```
int main() {
try {
   throw nullptr;
} catch(int* p) {  // match the pointer.
   return p == 0;
}
}
```

C++11 *End of C++11 only.*

Arrays

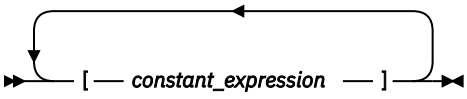
An *array* is a collection of objects of the same data type, allocated contiguously in memory. Individual objects in an array, called *elements*, are accessed by their position in the array. The subscripting operator ([]) provides the mechanics for creating an index to array elements. This form of access 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. C++ 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.

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]
```

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.

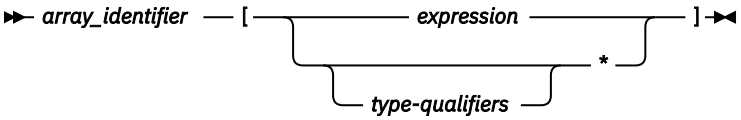
Related information

- “Array subscripting operator [ ]” on page 149
- “Initialization of arrays” on page 108

Variable length arrays

A variable length array, which is a C99 feature, is an array of automatic storage duration whose length is determined at runtime.

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. A variable length array cannot be initialized.

Note: In C++ applications, storage allocated for use by variable length arrays is not released until the function they reside in completes execution.

A variable length array can be the operand of a `sizeof` expression. In this case, the operand is evaluated at runtime, 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.

The C++ extension does not include support for references to a variable length array type; neither may a function parameter be a reference to a variable length array type.

Related information

- “Flexible array members ” on page 67

Compatibility of arrays

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 17

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.

Reference types include both lvalue reference and C++11 rvalue reference types. A lvalue reference type is defined by placing the reference modifier & after the type specifier. C++11 An rvalue reference type is defined by placing the reference modifier && or and 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.

The following types of references are invalid:

- References to `NULL`
- References to `void`
- References to invalid objects or functions
- References to bit fields
- References to references except with C++11 reference collapsing. See "Reference collapsing (C++11)" on page 171 for more information.

References to `NULL` are not allowed.

Related information

- "Initialization of references (C++ only)" on page 110
- "Pointers" on page 92
- "Reference conversions (C++ only)" on page 122
- "Address operator &" on page 135
- "Pass by reference" on page 214

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.

C++ You can initialize variables at namespace scope with nonconstant expressions.

C You cannnot initialize variables at global scope with nonconstant expressions.

"Initialization and storage classes" on page 101 discusses the rules for initialization according to the storage class of variables.

"Designated initializers for aggregate types (C only)" on page 102 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 104
- "Initialization of pointers" on page 107
- "Initialization of arrays" on page 108
- "Initialization of references (C++ only)" on page 110

Related information

- "Using class objects (C++ only)" on page 248

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.

Note: C++0x In C++0x, the keyword `auto` is no longer used as a storage class specifier. Instead, it is used as a type specifier. The compiler deduces the type of an `auto` variable from the type of its initializer expression. For more information, see "The auto type specifier (C++11)" on page 57.

Related information

- "The auto storage class specifier" on page 49

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 49

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 50

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 52

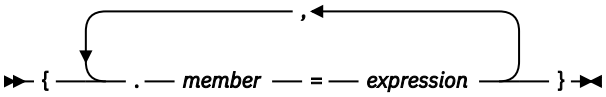
Designated initializers for aggregate types (C only)

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*.

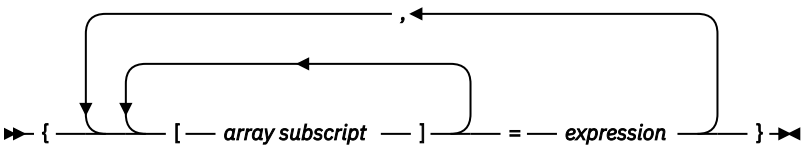
Designated initializers allow for the following flexibility:

- 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 `klm` are initialized with designated initializers:

```
struct xyz {
        int a;
        int b;
```

```
            int c;
            } klm = { .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
grid[0] [0]	8	grid[1] [2]	0
grid[0] [1]	6	grid[1] [3]	0
grid[0] [2]	4	grid[2] [0]	9
grid[0] [3]	1	grid[2] [1]	3
grid[1] [0]	0	grid[2] [2]	1
grid[1] [1]	0	grid[2] [3]	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” on page 104

- “Initialization of arrays” on page 108

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 264 for more information.

You can 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 102.

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 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 };
```

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 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" };
```

Related information

- “Structure and union variable declarations” on page 69
- “Explicit initialization with constructors (C++ only)” on page 305
- “Assignment operators” on page 141

Initialization of enumerations

The initializer for an enumeration variable contains the = symbol followed by an expression *enumeration_constant*.

C++ 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;
```

C++11 *Beginning of C++11 only.*

The following rules apply to both the scoped and unscoped enumerations.

- An enumeration cannot be initialized using an integer or enumeration constant from a different enumeration, without an explicit cast.
- An uninitialized enumeration variable has undefined value.

The following statement declares an unscoped enumeration `color`.

```
enum color { white, yellow, green, red, brown };
```

The following statement declares a scoped enumeration `letter` and references the scoped enumerators directly inside the scope of the enumeration. The initial values of A, B, C, and D are 0, 1, 1, and 2.

```
enum class letter { A, B, C = B, D = C + 1 };
```

The following statement defines a variable `let1` and initializes `let1` to the value of A. The integer value associated with A is 0.

```
letter let1 = letter :: A;
```

To reference scoped enumerators outside of the enumeration's scope, you must qualify the enumerators with the name of the enumeration. For example, the following statement is invalid.

```
letter let2 = A;     //invalid
```

The keyword `enum` in the following statement is optional and can be omitted.

```
enum letter let3 = letter :: B;
```

The `white` enumerator is visible in the following statement, because `color` is an unscoped enumeration.

```
color color1 = white;     // valid
```

Unscoped enumerations can also be qualified with their enumeration scope, for example:

```
color color2 = color :: yellow;    // valid
```

You cannot initialize an enumeration with an enumeration constant from a different enumeration or an integer without an explicit cast. For example, the following two statements are invalid.

```
letter let4 = color :: white;   // invalid
letter let5 = 1;                // invalid
```

You can use explicit cast to initialize an enumeration with an enumeration constant from a different enumeration or an integer. For example, the following two statements are valid.

```
letter let6 = (letter) color :: white;     // valid

letter let7 = (letter) 2;                  // valid
```

C++11 *End of C++11 only.*

Related information

- "Enumeration variable declarations" on page 76

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;` C++11 Or with the `nullptr` keyword. Such a pointer is a null pointer. It does not point to any object.

The following examples define pointers with null pointer values:

```
char *a = 0;
char *b = NULL;
```

C++11 *Beginning of C++11 only.*

```
char *ch = nullptr;
```

C++11 *End of C++11 only.*

Related information

- “Pointers” on page 92

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 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`.

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
`name1[0]`	J	`name2[0]`	J	`name3[0]`	J
`name1[1]`	a	`name2[1]`	a	`name3[1]`	a
`name1[2]`	n	`name2[2]`	n	`name3[2]`	n

Element	Value	Element	Value	Element	Value
		name2[3]	\0	name3[3]	\0

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

```
static char name3[3] = "Jan";
```

C++ 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
grid[0][0]	8	grid[1][2]	1
grid[0][1]	6	grid[1][3]	1
grid[0][2]	4	grid[2][0]	0
grid[0][3]	1	grid[2][1]	0
grid[1][0]	9	grid[2][2]	0
grid[1][1]	3	grid[2][3]	0

- C *Beginning of 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
`grid[0][0]`	0	`grid[1][2]`	0
`grid[0][1]`	0	`grid[1][3]`	0
`grid[0][2]`	0	`grid[2][0]`	8
`grid[0][3]`	0	`grid[2][1]`	6
`grid[1][0]`	0	`grid[2][2]`	4
`grid[1][1]`	0	`grid[2][3]`	1

C *End of C only.*

Related information

- “Arrays” on page 97

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

Related information

- “References (C++ only)” on page 99
- “Pass by reference” on page 214

Reference binding

Suppose T and U are two types. If ignoring top-level cv-qualifiers, T is of the same type as U or is a base class of U, T and U are reference-related.

Example 1

```
typedef int t1;
typedef const int t2;
```

In this example, `t1` and `t2` are reference-related.

If T and U are reference-related, and T is at least as cv-qualified as U, T is reference-compatible with U. In Example 1, `t1` is not reference-compatible with `t2`, but `t2` is reference-compatible with `t1`.

If an lvalue reference `r` to type T is to be initialized by an expression e of type U, and T is reference-compatible with U, the reference `r` can be bound directly to e or a base class subobject of e unless T is an inaccessible or ambiguous base class of U.

Example 2

```
int a = 1;
const int& ra = a;

struct A {};
struct B: A {} b;

A& rb = b;
```

In this example, the `const int` type is reference-compatible with the `int` type, so `ra` can be bound directly to `a`. Structure A is reference-related to structure B, so `rb` can be bound directly to `b`.

If an lvalue reference `r` to type T is to be initialized by an expression e of type U, `r` can be bound to the lvalue result of the conversion of e or a base class of e if the following conditions are satisfied. In this case, the conversion function is chosen by overload resolution.

- U is a class type.
- T is not reference-related to U.
- e can be converted to an lvalue of type S, and T is reference-compatible with S.

Example 3

```
struct A {
   operator int&();
};

const int& x= A();
```

In this example, structure A is a class type, and the `const int` type is not reference-related to structure A. However, A can be converted to an lvalue of type `int`, and `const int` is reference-compatible with `int`, so reference x of type `const int` can be bound to the conversion result of `A()`.

By default, the compiler cannot bind a non-const or volatile lvalue reference to an rvalue.

Example 4

```
int& a = 2; // error
const int& b = 1; // ok
```

In this example, the variable a is a non-const lvalue reference. The compiler cannot bind a to the temporary initialized with the rvalue expression 2, and issues an error message. The variable b is a non-volatile const lvalue reference, which can be initialized with the temporary initialized with the rvalue expression 1.

C++11 *Beginning of C++11 only.*

Suppose an expression e of type U belongs to one of the following value categories:

- An xvalue
- A class prvalue
- An array prvalue
- A function lvalue

If an rvalue reference or a non-volatile const lvalue reference `r` to type T is to be initialized by the expression e, and T is reference-compatible with U, reference `r` can be initialized by expression e and bound directly toe or a base class subobject of e unless T is an inaccessible or ambiguous base class of U.

Example 5

```
int& func1();
int& (&&rf1)()=func1;

int&& func2();
int&& rf2 = func2();

struct A{
   int arr[5];
};
int(&&ar_ref)[5] = A().arr;
A&& a_ref = A();
```

In this example, `rf1`, `rf2`, `ar_ref`, and `a_ref` are all rvalue references. `rf1` is bound to the function lvalue `func1`, `rf2` is bound to the xvalue result of the call `func2()`, `ar_ref` is bound to the array prvalue`A().arr`, and `a_ref` is bound to the class prvalue `A()`.

Suppose `r` is an rvalue reference or non-volatile const lvalue reference to type T, and `r` is to be initialized by an expression e of type U. `r` can be bound to the conversion result of e or a base class of e if the following conditions are satisfied. In this case, the conversion function is chosen by overload resolution.

- U is a class type.
- T is not reference-related to U.
- e can be converted to an xvalue, class prvalue, or function lvalue type of S, and T is reference-compatible with S.

Example 6

```
int i;
struct A {
   operator int&&() {
     return static_cast<int&&>(i);
   }
 };

const int& x = A();

int main() {
   assert(&x == &i);
}
```

In this example, structure A is a class type, and the `const int` type is not reference-related to structure A. However, A can be converted to an xvalue of type `int`, and `const int` is reference-compatible with `int`, so reference x of `const int` can be initialized with `A()` and bound to variable `i`.

An rvalue reference can be initialized with an lvalue in the following contexts:

- A function lvalue
- A temporary converted from an lvalue
- An rvalue result of a conversion function for an lvalue object that is of a class type

Example 7

```
int i = 1;
int&& a = 2; // ok
int&& b = i; // error
double&& c = i; // ok
```

In this example, the rvalue reference a can be bound to the temporary initialized with the rvalue expression 2, but the rvalue reference b cannot be bound to the lvalue expression `i`. You can bind the rvalue reference cto the temporary value `1.0` that is converted from the variable `i`.

C++11 *End of C++11 only.*

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.

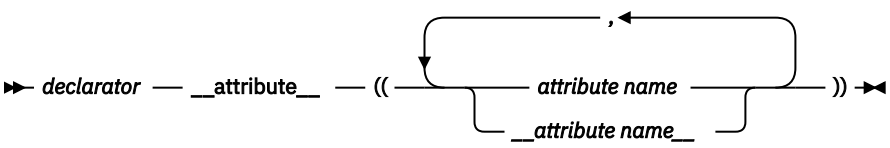
Variable attributes

IBM i *Beginning of IBM Extension.*

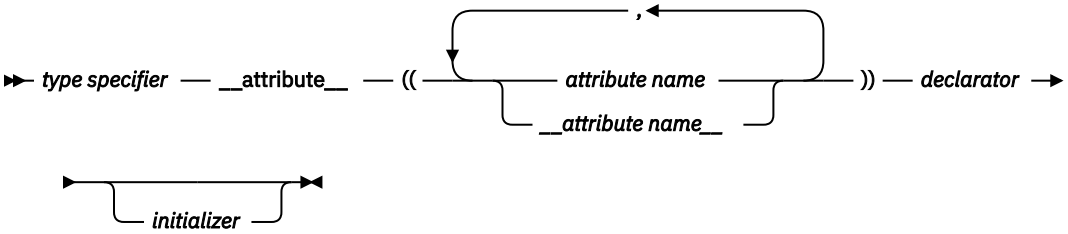
Variable attributes are language extensions provided to facilitate the compilation of programs developed with the GNU C/C++ compilers. These language features allow you to use named attributes to specify special properties of data objects. *Variable* attributes apply to the declarations of simple variables, aggregates, and member variables of aggregates.

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 IBM i 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;          /* 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 following variable attributes are supported:

- "The aligned variable attribute" on page 114
- "The packed variable attribute" on page 115
- "The mode variable attribute" on page 116
- "The weak variable attribute" on page 116

Related information

- "Type attributes" on page 84
- "Function attributes" on page 209

IBM i *End of IBM Extension.*

The aligned variable attribute

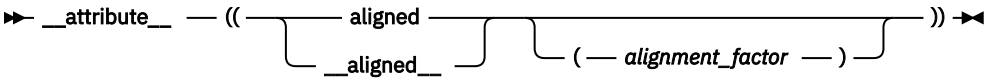
IBM i *Beginning of IBM Extension.*

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



The *alignment_factor* is the number of bytes, specified as a constant expression that evaluates to a positive power of 2. You can specify a value up to a maximum of 1048576 bytes. If you omit the alignment factor (and its enclosing parentheses) the compiler automatically uses 16 bytes. 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;
```

Related information

- “The __align type qualifier” on page 80
- "Aligning data" in the *ILE C/C++ Programmer's Guide*
- “The __alignof__ operator” on page 138
- “The aligned type attribute” on page 85

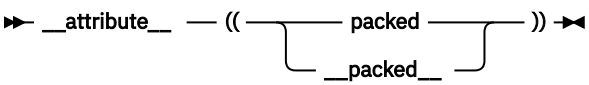
IBM i *End of IBM Extension.*

The packed variable attribute

IBM i *Beginning of IBM Extension.*

The variable attribute `packed` allows you to override the default alignment mode, to reduce the alignment for all members of an aggregate, or selected members of an aggregate to the smallest possible alignment: one byte for a member and one bit for a bit field member.

packed variable attribute syntax



Related information

- “The __align type qualifier” on page 80

- "Aligning data" in the *ILE C/C++ Programmer's Guide*
- “The __alignof__ operator” on page 138
- “The packed type attribute” on page 85

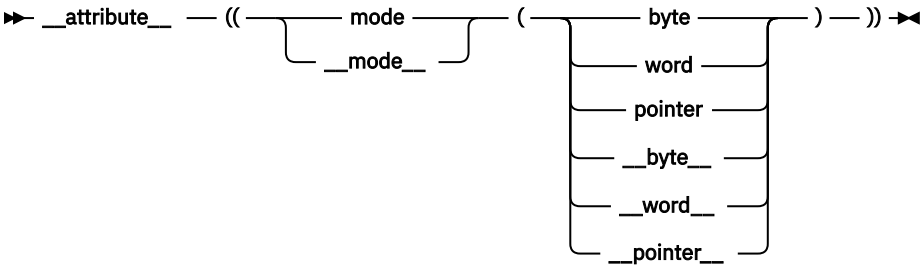
IBM i *End of IBM Extension.*

The mode variable attribute

IBM i *Beginning of IBM Extension.*

The variable attribute `mode` allows you to override the type specifier in a variable declaration, to specify the size of a particular integral type.

mode variable attribute syntax



The valid argument for the mode is any of the of the following type specifiers that indicates a specific width:

- `byte` means a 1-byte integer type
- `word` means a 4-byte integer type
- `pointer` means an 8-byte integer type

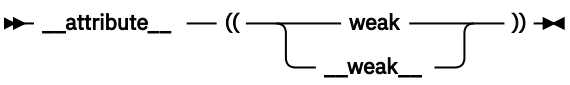
IBM i *End of IBM Extension.*

The weak variable attribute

IBM i *Beginning of IBM Extension.*

The `weak` variable attribute causes the symbol resulting from the variable declaration to appear in the object file as a weak symbol, rather than a global one. The language feature provides the programmer writing library functions with a way to allow variable definitions in user code to override the library declaration without causing duplicate name errors.

weak variable attribute syntax



Related information

- **#pragma weak** in the *ILE C/C++ Compiler Reference*
- “The weak function attribute” on page 211

IBM i *End of IBM Extension.*

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.
- 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.

C The implicit conversion result is an rvalue.

C++ The implicit conversion result belongs to one of the following value categories depending on different converted expressions types:

- An lvalue if the type is an lvalue reference type or an C++11 rvalue reference to a function type
- C++11 An xvalue if the type is an rvalue reference to an object type
- A C++11 (prvalue) rvalue in other cases

You can perform *explicit* type conversions using a *cast* expression, as described in "Cast expressions" on page 154. 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" on page 117
- "Lvalue-to-rvalue conversions" on page 120
- "Pointer conversions" on page 120
- "Reference conversions (C++ only)" on page 122
- "Qualification conversions (C++ only)" on page 122
- "Function argument conversions" on page 122

Related information

- "User-defined conversions (C++ only)" on page 313
- "Conversion constructors (C++ only)" on page 314
- "Conversion functions (C++ only)" on page 317
- "The switch statement" on page 179
- "The if statement" on page 178
- "The return statement" on page 188

Arithmetic conversions and promotions

The following sections discuss the rules for the standard conversions for arithmetic types:

- "Integral conversions" on page 118
- "Floating-point conversions " on page 118
- "Boolean conversions" on page 118

© Copyright IBM Corp. 1993, 2015

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 119.

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`.

C++11 A null pointer with the `nullptr` value is converted to `false`.

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.

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. “Expressions and operators” on page 125 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 15. Conversion rankings for floating-point types

Operand type
`long double`
`double`
`float`

Table 16. Conversion rankings for decimal floating-point types

Operand type
IBM i _Decimal128
IBM i _Decimal64
IBM i _Decimal32

Table 17. Conversion rankings for integer types

Operand type
`unsigned long long` or `unsigned long long int`
`long long` or `long long int`
`unsigned long int`
`long int`[1]

Table 17. Conversion rankings for integer types (continued)
Operand type
`unsigned int`[1]
`int` and enumerated types
`short int`
`char`, `signed char` and `unsigned char`
Boolean

Note:

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 54
- "Boolean types" on page 55
- "Floating-point types" on page 55
- "Character types " on page 56
- "Enumerations" on page 72
- "Binary expressions" on page 141

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
C++ 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 125

Pointer conversions

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

C *Beginning of 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 18. Legal assignment conversions for C pointers

Left operand type	Permitted right operand types
pointer to (object) T	• the constant `0` • a pointer to a type compatible with T • a pointer to `void` (`void*`)
pointer to (function) F	• the constant `0` • a 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*`.

C *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 92
- “Integer constant expressions” on page 128
- “Arrays” on page 97
- “Pointers to functions” on page 219
- “Pointers to members (C++ only)” on page 259
- “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 *Beginning of C only.*

The allowable assignment conversions involving `void*` as the left operand are shown in the following table.

Table 19. Legal assignment conversions in C for void*	
Left operand type	**Permitted right operand types**
(void*)	• The constant 0. • A pointer to an object. The object may be of incomplete type. • (void*)

C *End of C only.*

C++ *Beginning of 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.

C++ *End of C++ only.*

Related information

- "The void type" on page 57

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 99
- "Initialization of references (C++ only)" on page 110
- "Function calls" on page 213
- "Function return values" on page 203

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 79

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.
- C++ Non-static class member functions are converted to pointers to members.

Related information

- “Integral and floating-point promotions” on page 119
- “The transparent_union type attribute (C only)” on page 86
- “Function call expressions” on page 132
- “Function calls” on page 213

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" on page 125
- "Primary expressions" on page 127
- "Function call expressions" on page 132
- "Member expressions" on page 132
- "Unary expressions" on page 133
- "Binary expressions" on page 141
- "Conditional expressions" on page 152
- "Compound literal expressions" on page 162
- "Cast expressions" on page 154
- "new expressions (C++ only)" on page 163
- "delete expressions (C++ only)" on page 166
- "throw expressions (C++ only)" on page 167

"Operator precedence and associativity" on page 167 provides tables listing the precedence of all the operators described in the various sections listed above.

C++ 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 235.

Lvalues and rvalues

Expressions can be categorized into one of the following value categories:

Lvalue

An expression can appear on the left side of an assignment expression if the expression is not `const` qualified.

C++11 **Xvalue**

An rvalue reference that is to expire.

C++11 **(Prvalue) rvalue**

C++11 A non-xvalue expression that appears only on the right side of an assignment expression.

Rvalues include both xvalues and prvalues. Lvalues and xvalues can be referred as glvalues.

Notes:

- Class C++11 (prvalue) rvalues can be cv-qualified, but non-class C++11 (prvalue) rvalues cannot be cv-qualified.
- Lvalues C++11 and xvalues can be of incomplete types, but C++11 (prvalue) rvalues must be of complete types or `void` types.

© Copyright IBM Corp. 1993, 2015

An *object* is a region of storage that can be examined and stored into. An *lvalue* or C++11 *xvalue*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, a C++11 xvalue, a C++11 (prvalue) 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()`

C++11 *Beginning of C++11 only.*

The following expressions are xvalues:

- The result of calling a function whose return type is of an rvalue reference type
- A cast to an rvalue reference
- A nonstatic data member of a non-reference type accessed through an xvalue expression
- A pointer to member access expression in which the first operand is an xvalue expression and the second operand is of a pointer to member type

See the following example:

```
int a;
int&& b= static_cast<int&&>(a);

struct str{
   int c;
};

int&& f(){
   int&& var =1;
   return var;
}

str&& g();
int&& rc = g().c;
```

In this example, The initializer for rvalue reference b is an xvalue because it is a result of a cast to an rvalue reference. A call to the function `f()` produces an xvalue because the return type of this function is of the `int&&` type. The initializer for rvalue reference `rc` is an xvalue because it is an expression that accesses a nonstatic non-reference data member c through an xvalue expression.

C++11 *End of C++11 only.*

Related information

- “Arrays” on page 97
- “Lvalue-to-rvalue conversions” on page 120

Primary expressions

Primary expressions fall into the following general categories:

- “Names” on page 127 (identifiers)
- “Literals” on page 128 (constants)
- “Integer constant expressions” on page 128
- “Identifier expressions (C++ only)” on page 129
- “Parenthesized expressions ( )” on page 129
- C++ The `this` pointer (described in “The this pointer (C++ only)” on page 261)
- 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 20. 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

Table 20. Primary expressions: Names (continued)		
Name declared as	**Evaluates to**	**Is an lvalue?**
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 `sizeof` operator.	C no C++ 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 `sizeof` operator, or as the function in a function call expression.	C no C++ 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 24
- "The this pointer (C++ only)" on page 261

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 initialized with compile-time constant expressions or C++11 `constexpr` expressions
- `static const` data members initialized with compile-time constant expressions or C++11 `constexpr` expressions
- 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 runtime, 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.)

Note: C++11 The C++11 standard generalizes the concept of constant expressions. For more information, see "Generalized constant expressions (C++11)" on page 131

Related information

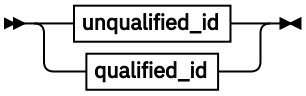
- "The sizeof operator" on page 138

Identifier expressions (C++ only)

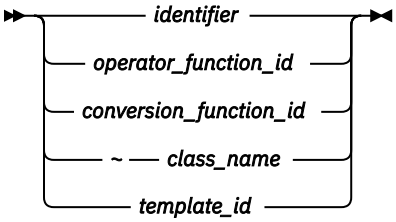
An identifier expression, or *id-expression*, is a restricted form of primary expression. Syntactically, an *id-expression* requires a higher level of complexity than a simple identifier to provide a name for all of the language elements of C++.

An *id-expression* can be either a qualified or unqualified identifier. It can also appear after the dot and arrow operators.

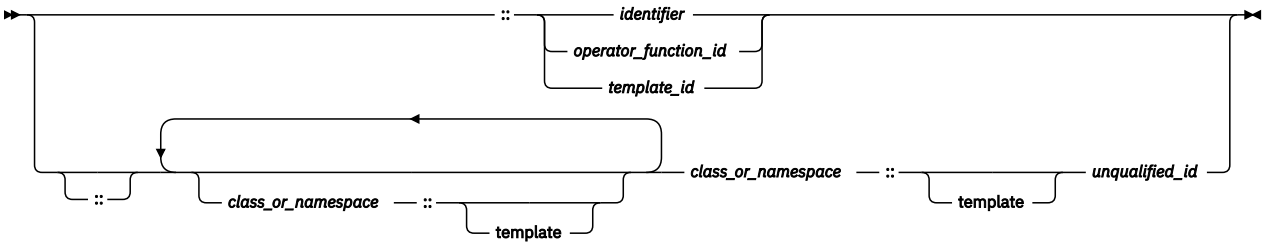
Identifier expression syntax



unqualified_id



qualified_id

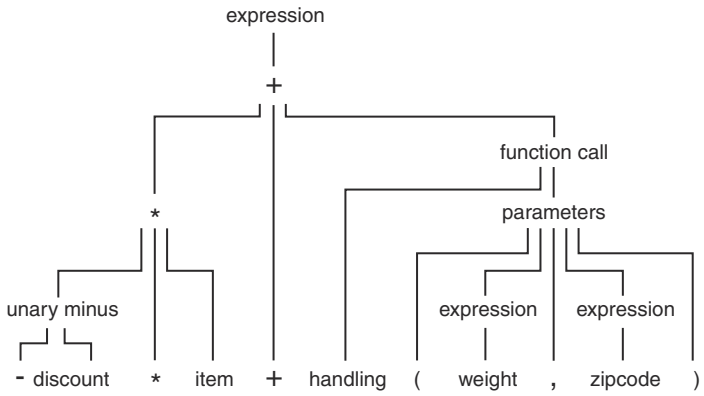


Related information

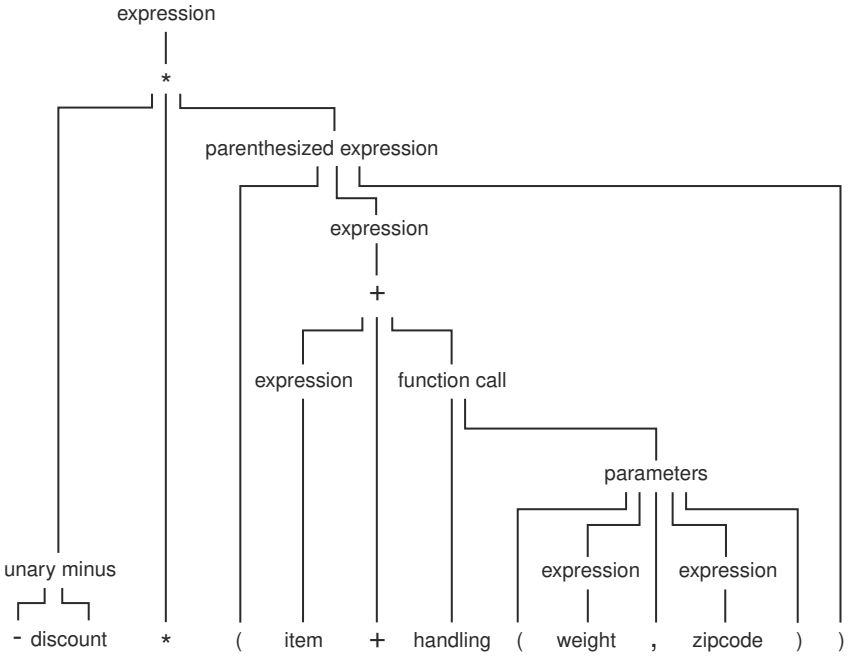
- "Identifiers" on page 23
- "Declarators" on page 89

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 167

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 250
- “Namespaces (C++ only)” on page 223

Generalized constant expressions (C++11)

The C++11 standard generalizes the concept of constant expressions and introduces a new keyword `constexpr` as a declaration specifier. A constant expression is an expression that can be evaluated at compile time by the compiler. The major benefits of this feature are as follows:

- Improves type safety and portability of code that requires compile-time evaluation
- Improves support for systems programming, library building, and generic programming
- Improves the usability of Standard Library components. Library functions that can be evaluated at compile time can be used in contexts that require constant expressions.

An object declaration with the `constexpr` specifier declares that object to be constant. The `constexpr` specifier can be applied only to the following contexts:

- The definition of an object
- The declaration of a function or function template
- The declaration of a static data member of a literal type

If you declare a function that is not a constructor with a `constexpr` specifier, then that function is a `constexpr` function. Similarly, if you declare a constructor with a `constexpr` specifier, then that constructor is a `constexpr` constructor.

With this feature, constant expressions can include calls to template and non-template `constexpr` functions, `constexpr` objects of class literal types, and references bound to `const` objects that are initialized with constant expressions.

Related information

- “The constexpr specifier (C++11)” on page 63
- “Constexpr functions (C++11)” on page 220
- “Constexpr constructors (C++11)” on page 303

- “Extensions for C++11 compatibility” on page 411

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`.

C A function call expression is always an rvalue.

C++ A function call belongs to one of the following value categories depending on the result type of the function:

- An lvalue if the result type is an lvalue reference type or C++11 an rvalue reference to a function type
- C++11 An xvalue if the result type is an rvalue reference to an object type
- A C++11 (prvalue) rvalue in other cases

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 122
- “Function calls” on page 213

Member expressions

Member expressions indicate members of classes, structures, or unions. The member operators are:

- “Dot operator .” on page 132
- “Arrow operator ->” on page 133

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 71
- “Pseudo-destructors (C++ only)” on page 312

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 92
- “Access to structure and union members” on page 71
- “Class members and friends (C++ only)” on page 255
- “Pseudo-destructors (C++ only)” on page 312

Unary expressions

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

The supported unary operators are:

- “Increment operator ++” on page 134
- “Decrement operator --” on page 134
- “Unary plus operator + ” on page 134
- “Unary minus operator - ” on page 135
- “Logical negation operator !” on page 135
- “Bitwise negation operator ~” on page 135
- “Address operator &” on page 135
- “Indirection operator *” on page 136
- C++ typeid
- IBM i alignof
- sizeof
- IBM i __typeof__

All unary operators have the same precedence and have right-to-left associativity, as shown in Table 24 on page 168.

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 93
- “Lvalues and rvalues” on page 125
- “Arithmetic conversions and promotions” on page 117

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.

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.

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 55

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
```

Related information

- "Trigraph sequences" on page 40

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++ *Beginning of 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.

C++ *End of C++ only.*

Related information

- “Indirection operator *” on page 136
- “Pointers” on page 92
- “References (C++ only)” on page 99

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.

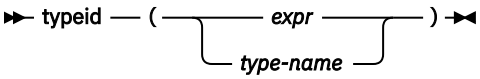
Related information

- “Arrays” on page 97
- “Pointers” on page 92

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



The `typeid` operator requires RunTime Type Identification (RTTI) to be generated, which must be explicitly specified at compile time through a compiler option.

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 runtime. 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 91
- "The __typeof__ operator" on page 140

The __alignof__ operator

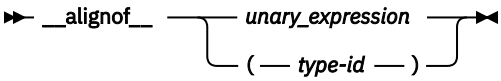
IBM i *Beginning of IBM Extension.*

The `__alignof__` operator is a language extension to C99 and Standard C++ that returns the number of bytes used in the alignment of its operand. The operand can be an expression or a parenthesized type identifier. If the operand is an expression representing an lvalue, the number returned by `__alignof__` represents the alignment that the lvalue is known to have. The type of the expression is determined at compile time, but the expression itself is not evaluated. If the operand is a type, the number represents the alignment usually required for the type on the target platform.

The `__alignof__` operator may not be applied to the following:

- An lvalue representing a bit field
- A function type
- An undefined structure or class
- An incomplete type (such as `void`)

__alignof__ operator syntax



If *type-id* is a reference or a referenced type, the result is the alignment of the referenced type. If *type-id* is an array, the result is the alignment of the array element type. If *type-id* is a fundamental type, the result is implementation-defined.

Related information

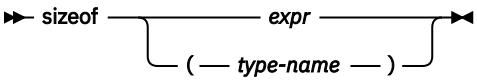
- "The aligned variable attribute" on page 114

IBM i *End of IBM Extension.*

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



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.

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.
C++ 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.
C++ 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.

C++11 *Beginning of C++11 only.*

`sizeof...` is a unary expression operator introduced by the variadic template feature. This operator accepts an expression that names a parameter pack as its operand. It then expands the parameter pack and returns the number of arguments provided for the parameter pack. Consider the following example:

```
template<typename...T> void foo(T...args){
    int v = sizeof...(args);
}
```

In this example, the variable v is assigned to the number of the arguments provided for the parameter pack `args`.

Notes:

- The operand of the `sizeof...` operator must be an expression that names a parameter pack.
- The operand of the `sizeof` operator cannot be an expression that names a parameter pack or a pack expansion.

For more information, see “Variadic templates (C++11)” on page 352

C++11 *End of C++11 only.*

Related information

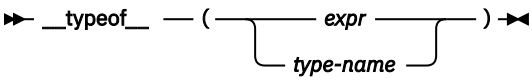
- "Type names" on page 91
- "Integer constant expressions" on page 128
- "Arrays" on page 97
- "References (C++ only)" on page 99

The __typeof__ operator

IBM i *Beginning of 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.

__typeof__ operator syntax



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; }
```

Related information

- "Type names" on page 91
- "typedef definitions" on page 77
- LANGLVL(*EXTENDED) in the *ILE C/C++ Compiler Reference*

IBM i *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 141
- "Multiplication operator *" on page 143
- "Division operator/" on page 143
- "Remainder operator %" on page 143
- "Addition operator +" on page 144
- "Subtraction operator -" on page 144
- "Bitwise left and right shift operators << >>" on page 144
- "Relational operators < > <= >=" on page 145
- "Equality and inequality operators == !=" on page 146
- "Bitwise AND operator &" on page 147
- "Bitwise exclusive OR operator ^" on page 147
- "Bitwise inclusive OR operator |" on page 148
- "Logical AND operator &&" on page 148
- "Logical OR operator ||" on page 149
- "Array subscripting operator [ ]" on page 149
- "Comma operator ," on page 150
- C++ "Pointer to member operators .* ->* (C++ only)" on page 152

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 25 on page 169.

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 125
- "Arithmetic conversions and promotions" on page 117

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 =" on page 142
- "Compound assignment operators" on page 142

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.

C The result of an assignment expression is not an lvalue. C++ 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 125
- “Pointers” on page 92
- “Type qualifiers” on page 79

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.

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
+= or -=	Arithmetic	Arithmetic
+= or -=	Pointer	Integral type
*=, /=, and %=	Arithmetic	Arithmetic
<<=, >>=, &=, ^=, and |=	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
+=	`index += 2`	`index = index + 2`
-=	`*(pointer++) -= 1`	`*pointer = *(pointer++) - 1`
*=	`bonus *= increase`	`bonus = bonus * increase`

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 |= ON	flag = flag | ON

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.

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.

The sign of the remainder is the same as the sign of the quotient.

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 93
- “Pointer conversions” on page 120

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. Behavior is undefined if the pointers do not refer to objects in the same array.

IBM i The type of the result of pointer difference arithmetic will be `ptrdiff_t` (defined in `stddef.h`) if the TERASPACE(*NO) compiler option is specified. If the TERASPACE(*YES) compiler option is specified, the result of the pointer difference arithmetic will be of type signed long long.

Related information

- “Pointer arithmetic” on page 93
- “Pointer conversions” on page 120

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:

```
0000000111110110
```

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.
<=	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.

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`.

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 142

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	0000000001011100
bit pattern of b	0000000000101110
bit pattern of a & b	0000000000001100

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 ¬ 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	0000000001011100
bit pattern of b	0000000000101110
bit pattern of a ^ b	0000000001110010

Related information

- "Trigraph sequences" on page 40

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 a	0000000001011100
bit pattern of b	0000000000101110
bit pattern of a | b	0000000001111110

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 40

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. Array bounds are not checked for built-in array types.

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
```

C++ 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 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];
}
```

Related information

- “Pointers” on page 92
- “Integral types” on page 54
- “Lvalues and rvalues” on page 125
- “Arrays” on page 97
- “Overloading subscripting (C++ only)” on page 240
- “Pointer arithmetic” on page 93

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 *Beginning of C only.*

The result of a comma expression is not an lvalue.

C *End of C only.*

C++ *Beginning of 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
```

C++ *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**
`for (i=0; i<2; ++i, f() );`	A `for` statement in which `i` is incremented and `f()` is called at each iteration.
`if ( f(), ++i, i>1 ) { /* ... */ }`	An `if` statement in which function `f()` is called, variable `i` is incremented, and variable `i` is tested against a value. The first two expressions within this comma expression are evaluated before the expression `i>1`. Regardless of the results of the first two expressions, the third is evaluated and its result determines whether the `if` statement is processed.

Statement	Effects
`func( ( ++a, f(a) ) );`	A function call to `func()` in which a is incremented, the resulting value is passed to a function `f()`, and the return value of `f()` is passed to `func()`. The function `func()` 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 255
- "Pointers to members (C++ only)" on page 259

Conditional expressions

A *conditional expression* is a compound expression that contains a condition that is implicitly converted to type `bool` in C++(*operand*$_1$), an expression to be evaluated if the condition evaluates to true (*operand*$_2$), and an expression to be evaluated if the condition has the value false (*operand*$_3$).

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 *Beginning of C only.*

In C, a conditional expression is not an lvalue, nor is its result.

Table 21. 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

C *End of C only.*

Types in conditional C++ expressions

C++ *Beginning of C++ only.*

In C++, a conditional expression is a valid lvalue if its type is not `void`, and its result is an lvalue.

Table 22. 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

C++ *End of C++ only.*

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

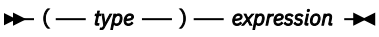
A cast operator is used for *explicit type conversions*. It converts the value of an expression to a specified type.

The following cast operators are supported:

- "Cast operator ()" on page 154
- "The static_cast operator (C++ only)" on page 156
- "The reinterpret_cast operator (C++ only)" on page 158
- "The const_cast operator (C++ only)" on page 159
- "The dynamic_cast operator (C++ only)" on page 160

Cast operator ()

Cast expression syntax



C In C, the result of this operation is not an lvalue.

C++ In C++, the cast result belongs to one of the following value categories:

- If `type` is an lvalue reference type or C++11 an rvalue reference to a function type, the cast result is an lvalue.
- C++11 If `type` is an rvalue reference to an object type, the cast result is an xvalue.
- In all other cases, the cast result is a C++11 (prvalue) 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++ *Beginning of 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.

C++ *End of C++ only.*

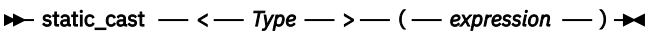
Related information

- "Type names" on page 91
- "Conversion functions (C++ only)" on page 317
- "Conversion constructors (C++ only)" on page 314
- "Lvalues and rvalues" on page 125

The static_cast operator (C++ only)

The *static_cast operator* converts a given expression to a specified type.

static_cast operator syntax



C++11 With the right angle bracket feature, you may specify a `template_id` as `Typr` in the `static_cast` operator with the >> token in place of two consecutive > tokens. For details, see "Class templates (C++ only)" on page 329.

The result of `static_cast<Type>(expression)` belongs to one of the following value categories:

- If Type is an lvalue reference type or C++11 an rvalue reference to a function type, `static_cast<Type>(expression)` is an lvalue.
- C++11 If Type is an rvalue reference to an object type, `static_cast<Type>(expression)` is an xvalue.
- In all other cases, `static_cast<Type>(expression)` is a C++11 (prvalue) rvalue.

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.

The compiler supports the following types of cast operations:

- An lvalue of type A to type B&, and the cast result is an lvalue of type B
- C++11 An lvalue or xvalue of type A to type B&&, and the cast result is an xvalue of type B
- A C++11 (prvalue) rvalue of pointer to A to pointer to B
- C++11 An lvalue of type A to type B&& if an xvalue of type A can be bound directly to a reference of type B&&
- An expression e to type T if the direct initialization `T t(e)` is valid.

To support the first three cast operations, the following conditions must be satisfied:

- A is a base class of B.
- There exists a standard conversion from a pointer to type B to a pointer to type A.
- Type B is the same as or more cv-qualified than type A.
- A is not a virtual base class or a base class of a virtual base class of B.

You can cast a C++11 (prvalue) rvalue of a pointer to member of A whose type is `cv1 T` to a C++11 (prvalue) rvalue of a pointer to member of B whose type is `cv2 T` if the following conditions are satisfied:

- B is a base class of A.
- There exists a standard conversion from a pointer to member of B whose type is T to a pointer to member of A whose type is T.
- `cv2` is the same or more cv-qualification than `cv1`.

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.

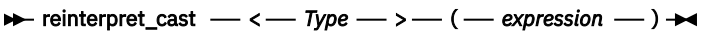
Related information

- “User-defined conversions (C++ only)” on page 313

The reinterpret_cast operator (C++ only)

A *reinterpret_cast operator* handles conversions between unrelated types.

reinterpret_cast operator syntax



C++11 With the right angle bracket feature, you may specify a `template_id` as `Typr` in the `reinterpret_cast` operator with the >> token in place of two consecutive > tokens. For details, see "Class templates (C++ only)" on page 329.

The result of `reinterpret_cast<Type>(expression)` belongs to one of the following value categories:

- If Type is an lvalue reference type or C++11 an rvalue reference to a function type, `reinterpret_cast<Type>(expression)` is an lvalue.
- C++11 If Type is an rvalue reference to an object type, `reinterpret_cast<Type>(expression)` is an xvalue.
- In all other cases, `reinterpret_cast<Type>(expression)` is a C++11 (prvalue) rvalue.

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++11 Given a type T and an lvalue expression x, the following two expressions for rvalue references have different syntax but the same semantics:

- `reinterpret_cast<T&&>(x)`
- `static_cast<T&&>(*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. It will also successfully compile optimized in C++, 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 313

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

C++11 With the right angle bracket feature, you may specify a `template_id` as Type in the `const_cast` operator with the >> token in place of two consecutive > tokens. For details, see "Class templates (C++ only)" on page 329.

The result of `const_cast<Type>(expression)` belongs to one of the following value categories:

- If Type is an lvalue reference to an object type, `const_cast<Type>(expression)` is an lvalue.
- C++11 If Type is an rvalue reference to an object type, `const_cast<Type>(expression)` is an xvalue.
- In all other cases, `const_cast<Type>(expression)` is a C++11 (prvalue) rvalue.

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.

If a pointer to T1 can be converted to a pointer to T2 using `const_cast<T2>`, where T1 and T2 are object types, you can also make the following types of conversions:

- An lvalue of type T1 to an lvalue of type T2 using `const_cast<T2&>`
- C++11 An lvalue or xvalue of type T1 to an xvalue of type T2 using `const_cast<T2&&>`
- C++11 A prvalue of class type T1 to an xvalue of type T2 using `const_cast<T2&&>`

If a conversion from a C++11 (prvalue) rvalue of type pointer to T1 to type pointer to T2 casts away constness, the following types of conversions also cast away constness:

- An lvalue of type T1 to an lvalue of type T2

- C++11 An expression of type T1 to an xvalue of type T2
- A C++11 (prvalue) rvalue of type pointer to data member of X of type T1 to type pointer to data member of Y of type T2

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 cannot 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 79

The dynamic_cast operator (C++ only)

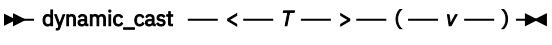
The *dynamic_cast* operator checks the following types of conversions at run time:

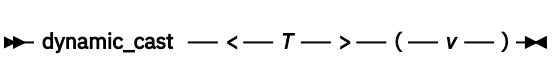
- A pointer to a base class to a pointer to a derived class
- An lvalue referring to a base class to an lvalue reference to a derived class

- An C++11 xvalue referring to a base class to an rvalue reference to a derived class

The *dynamic_cast* operator performs type conversions at runtime. 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++.

dynamic_cast operator syntax





C++11 With the right angle bracket feature, you may specify a `template_id` as T in the `dynamic_cast` operator with the >> token in place of two consecutive > tokens. For details, see "Class templates (C++ only)" on page 329.

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.

The following rules apply to the `dynamic_cast<T>(v)` expression:

- If T is a pointer type, v must be a C++11 (prvalue) rvalue, and `dynamic_cast<T>(v)` is a C++11 (prvalue) rvalue of type T.
- If T is an lvalue reference type, v must be an lvalue, and `dynamic_cast<T>(v)` is an lvalue of the type that is referred by T.
- C++11 If T is an rvalue reference type, `dynamic_cast<T>(v)` is an xvalue of the type that is referred by T.

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>`.

The `dynamic_cast` operator requires RunTime Type Information (RTTI) to be generated, which must be explicitly specified at compile time through a compiler option.

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  : SPP:0000 :1aefQPADEV0001TSTUSR  369019:220:0:6c
Address of bobj: SPP:0000 :1aefQPADEV0001TSTUSR  369019:220:0:6c
```

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

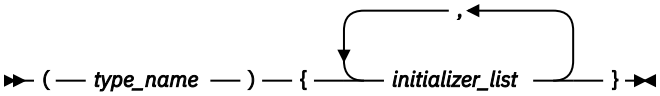
- “Derivation (C++ only)” on page 276
- “User-defined conversions (C++ only)” on page 313

Compound literal expressions

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



The *type_name* can be any data type, including 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.

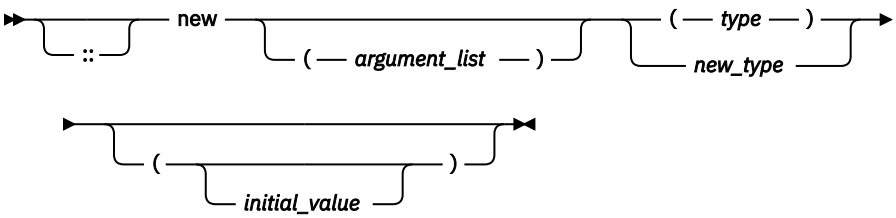
Related information

- "String literals" on page 33

new expressions (C++ only)

The new operator provides dynamic storage allocation.

new operator syntax



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 216

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 function rather than to a constructor.

Related information

- “delete expressions (C++ only)” on page 166
- “Scope resolution operator :: (C++ only)” on page 130
- “Overview of constructors and destructors” 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” 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:

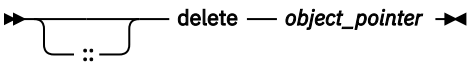
```
Operator new failed: no storage is available.
```

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.

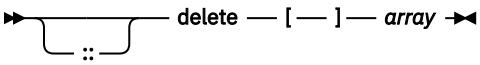
delete operator syntax



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



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" on page 299
- "The void type" on page 57

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

- "Exception handling (C++ only)" on page 367
- "The void type" on page 57

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 23. Precedence and associativity of postfix operators

Rank	Right associative?	Operator function	Usage
1	yes	C++ global scope resolution	:: *name_or_qualified name*
1		C++ class or namespace scope resolution	*class_or_namespace* :: *member*
2		member selection	*object* . *member*
2		member selection	*pointer* -> *member*
2		subscripting	*pointer* [*expr*]
2		function call	*expr* (*expr_list*)
2		value construction	*type* (*expr_list*)
2		postfix increment	*lvalue* ++
2		postfix decrement	*lvalue* --
2	yes	C++ type identification	typeid (*type*)
2	yes	C++ type identification at runtime	typeid (*expr*)
2	yes	C++ conversion checked at compile time	static_cast < *type* > (*expr*)
2	yes	C++ conversion checked at runtime	dynamic_cast < *type* > (*expr*)
2	yes	C++ unchecked conversion	reinterpret_cast < *type* > (*expr*)
2	yes	C++ `const` conversion	const_cast < *type* > (*expr*)

Table 24. Precedence and associativity of unary operators

Rank	Right associative?	Operator function	Usage
3	yes	size of object in bytes	sizeof *expr*
3	yes	size of type in bytes	sizeof (*type*)
3	yes	prefix increment	++ *lvalue*
3	yes	prefix decrement	-- *lvalue*
3	yes	bitwise negation	~ *expr*
3	yes	not	! *expr*
3	yes	unary minus	- *expr*
3	yes	unary plus	+ *expr*
3	yes	address of	& *lvalue*

Table 24. Precedence and associativity of unary operators (continued)			
Rank	**Right associative?**	**Operator function**	**Usage**
3	yes	indirection or dereference	* *expr*
3	yes	C++ create (allocate memory)	new *type*
3	yes	C++ create (allocate and initialize memory)	new *type* (*expr_list*) *type*
3	yes	C++ create (placement)	new *type* (*expr_list*) *type* (*expr_list*)
3	yes	C++ destroy (deallocate memory)	delete *pointer*
3	yes	C++ destroy array	delete [ ] *pointer*
3	yes	type conversion (cast)	(*type*) *expr*

Table 25. Precedence and associativity of binary operators			
Rank	**Right associative?**	**Operator function**	**Usage**
4		C++ member selection	*object .* ptr_to_member*
4		C++ member selection	*object ->* ptr_to_member*
5		multiplication	*expr * expr*
5		division	*expr / expr*
5		modulo (remainder)	*expr % expr*
6		binary addition	*expr + expr*
6		binary subtraction	*expr - expr*
7		bitwise shift left	*expr << expr*
7		bitwise shift right	*expr >> expr*
8		less than	*expr < expr*
8		less than or equal to	*expr <= expr*
8		greater than	*expr > expr*
8		greater than or equal to	*expr >= expr*
9		equal	*expr == expr*
9		not equal	*expr != expr*
10		bitwise AND	*expr & expr*
11		bitwise exclusive OR	*expr ^ expr*
12		bitwise inclusive OR	*expr	expr*
13		logical AND	*expr && expr*
14		logical inclusive OR	*expr || expr*

Table 25. Precedence and associativity of binary operators (continued)			
Rank	**Right associative?**	**Operator function**	**Usage**
15		conditional expression	*expr* ? *expr* : *expr*
16	yes	simple assignment	*lvalue* = *expr*
16	yes	multiply and assign	*lvalue* *= *expr*
16	yes	divide and assign	*lvalue* /= *expr*
16	yes	modulo and assign	*lvalue* %= *expr*
16	yes	add and assign	*lvalue* += *expr*
16	yes	subtract and assign	*lvalue* -= *expr*
16	yes	shift left and assign	*lvalue* <<= *expr*
16	yes	shift right and assign	*lvalue* >>= *expr*
16	yes	bitwise AND and assign	*lvalue* &= *expr*
16	yes	bitwise exclusive OR and assign	*lvalue* ^= *expr*
16	yes	bitwise inclusive OR and assign	*lvalue* |= *expr*
17	yes	C++ throw expression	throw *expr*
18		comma (sequencing)	*expr* , *expr*

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()`.

Reference collapsing (C++11)

Note: IBM supports selected features of C++11, known as C++0x before its ratification. IBM will continue to develop and implement the features of this standard. The implementation of the language level is based on IBM's interpretation of the standard. Until IBM's implementation of all the C++11 features is complete, including the support of a new C++11 standard library, the implementation may change from release to release. IBM makes no attempt to maintain compatibility, in source, binary, or listings and other compiler interfaces, with earlier releases of IBM's implementation of the new C++11 features.

Before C++11, references to references are ill-formed in the C++ language. In C++11, the rules of reference collapsing apply when you use references to references through one of the following contexts:

- A `decltype` specifier
- A `typedef` name
- A template type parameter

You can define a variable `var` whose declared type TR is a reference to the type T, where T is also a reference type. For example,

```
// T denotes the int& type
typedef int& T;

// TR is an lvalue reference to T
typedef T& TR;

// The declared type of var is TR
TR var;
```

The actual type of `var` is listed in the following table for different cases, where neither TR nor T is qualified by cv-qualifiers.

Table 26. Reference collapsing without cv-qualifiers		
T	**TR**	**Type of `var`**
A	T	A
A	T&	A&
A	T&&	A&&
A&	T	A&
A&	T&	A&
A&	T&&	A&
A&&	T	A&&
A&&	T&	A&
A&&	T&&	A&&
Note: Reference collapsing does not apply in this case, because T and TR are not both reference types.		

The general rule in this table is that when T and TR are both reference types, but are not both rvalue reference types, `var` is of an lvalue reference type.

Example 1

```
typedef int& T;

// a has the type int&
T&& a;
```

In this example, T is of the `int&` type, and the declared type of a is T&&. After reference collapsing, the type of a is `int&`.

Example 2

```
template <typename T> void func(T&& a);
auto fp = func<int&&>;
```

In this example, the actual parameter of T is of the `int&&` type, and the declared type of a is T&&. An rvalue reference to an rvalue reference is formed. After reference collapsing, the type of a is `int&&`.

Example 3

```
auto func(int& a) -> const decltype(a)&;
```

In this example, `decltype(a)`, which is a trailing return type, refers to the parameter a, whose type is `int&`. After reference collapsing, the return type of `func` is`int&`.

You can define a variable `var` whose declared type TR is a reference to the type T, where T is also a reference type. If either TR or T is qualified by cv-qualifiers, then the actual type of `var` is listed in the following table for different cases.

The actual type of `var` is listed in the following table for different cases, where neither TR nor T is qualified by cv-qualifiers.

Table 27. Reference collapsing with cv-qualifiers		
T	**TR**	**Type of `var`**
A	const T	const A
const A	volatile T&	const volatile A&
A	const T&&	const A&&

Table 27. Reference collapsing with cv-qualifiers (continued)		
T	**TR**	**Type of `var`**
A&	const T	A&
const A&	volatile T&	const A&
const A&	T&&	const A&
A&&	const T	A&&
const A&&	volatile T&	const A&
const A&&	T&&	const A&&
Note: Reference collapsing does not apply in this case, because T and TR are not both reference types.		

The general rule of this table is that when T is a reference type, the type of `var` inherits only the cv-qualifiers from T.

Related information

- “The decltype(expression) type specifier (C++11)” on page 59
- “typedef definitions” on page 77
- “Type template parameters (C++ only)” on page 324

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” on page 175
- “Expression statements” on page 176
- “Block statements” on page 177
- “Selection statements” on page 177
- “Iteration statements” on page 183
- “Jump statements” on page 186
- Declaration statements
- C++ `try` blocks
- “Null statement” on page 190

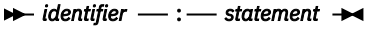
Related information

- “Data objects and declarations” on page 43
- “Function declarations” on page 191
- “try blocks (C++ only)” on page 367

Labeled statements

There are three kinds of labels: identifier, case, and default.

Labeled statement syntax



The label consists of the *identifier* and the colon (`:`) character.

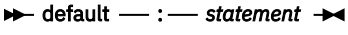
An identifier label may be used as the target of a goto statement. A goto statement can use a label before its definition. Identifier labels have their own namespace; you do not have to worry about identifier labels conflicting with other identifiers. However, you may not re-declare a label within a function.

IBM i An identifier label may also be used as the target of a `#pragma exception_handler` directive. See the *ILE C/C++ Programmer's Guide* for examples and more information about using the `#pragma exception_handler` directive.

Case and default label statements only appear in `switch` statements. These labels are accessible only within the closest enclosing `switch` statement.

case statement syntax

default statement syntax



The following are examples of labels:

```
comment_complete : ;            /* null statement label */
test_for_null : if (NULL == pointer)
```

© Copyright IBM Corp. 1993, 2015

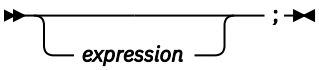
Related information

- “The goto statement” on page 189
- “The switch statement” on page 179

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

- “Expressions and operators” on page 125

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
```

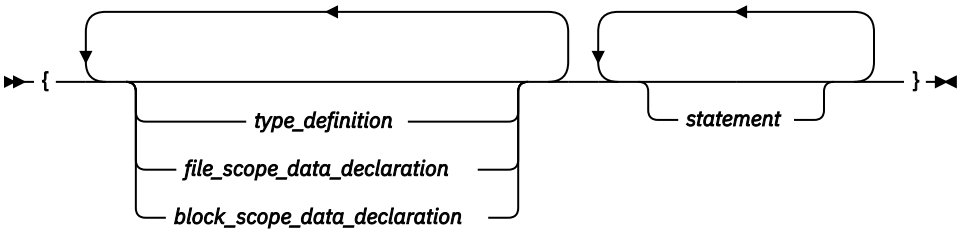
Related information

- “Data objects and declarations” on page 43
- “Expressions and operators” on page 125
- “Function call expressions” on page 132

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” on page 178

- "The switch statement" on page 179

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

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 55

The switch statement

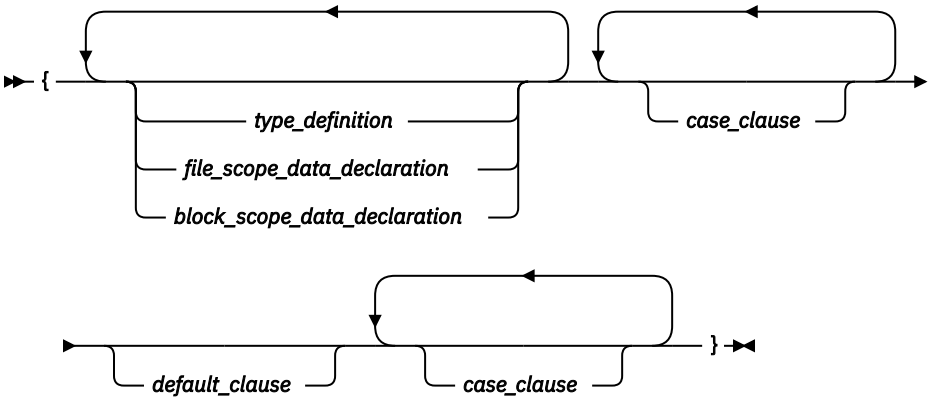
A *switch statement* is a selection statement that lets you transfer control to different statements within the `switch` body depending on the value of the switch expression. The `switch` expression must evaluate to an integral or enumeration value. The body of the `switch` statement contains *case clauses* 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

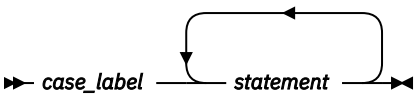
The *switch body* is enclosed in braces and can contain definitions, declarations, *case clauses*, and a *default clause*. Each case clause and default clause can contain statements.



Note: An initializer within a *type_definition*, *file_scope_data_declaration* or *block_scope_data_declaration* is ignored.

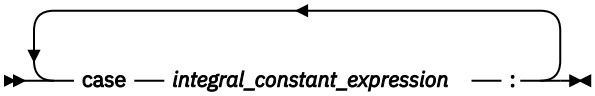
A *case clause* contains a *case label* followed by any number of statements. A case clause has the form:

Case clause syntax



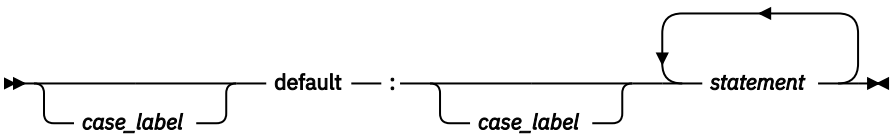
A *case label* contains the word `case` followed by an integral constant expression and a colon. The value of each integral constant expression must represent a different value; you cannot have duplicate `case` labels. Anywhere you can put one `case` label, you can put multiple `case` labels. A case 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.

Compiling with option **`CHECKOUT(*GENERAL)`** finds case labels that fall through when they should not.

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.

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:

```
/**
 ** 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

- **CHECKOUT(*GENERAL)** in the *ILE C/C++ Compiler Reference*

- Case and Default Labels
- “The break statement” on page 186

Iteration statements

Iteration statements consist of the following types of statements:

- “The while statement” on page 183
- “The do statement” on page 184
- “The for statement” on page 184

Related information

- “Boolean types” on page 55

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

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.

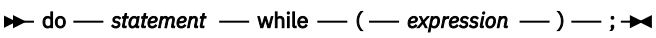
In the following example, `item[index]` triples and is printed out, as long as the value of the expression `++index` is less than `MAX_INDEX`. When `++index` evaluates to `MAX_INDEX`, the `while` statement ends.

```
/**
 ** 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



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 do 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:

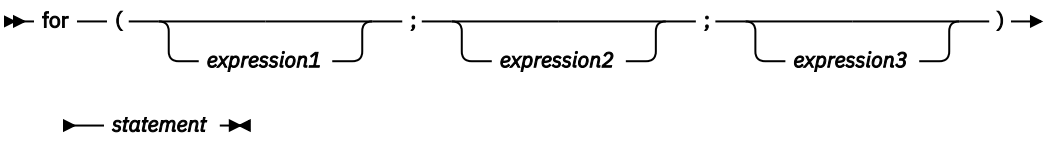
```
Value of i: 1
Value of i: 2
Value of i: 3
Value of i: 4
Value of i: 5
```

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



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.

Related information

- **LANGLVL** in the *ILE C/C++ Compiler Reference*

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

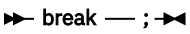
Jump statements consist of the following types of statements:

- "The break statement" on page 186
- "The continue statement" on page 186
- "The return statement" on page 188
- "The goto statement" on page 189

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



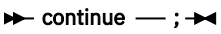
In an iterative statement, the `break` statement ends the loop and moves control to the next statement outside the loop. Within nested statements, the `break` statement ends only the smallest enclosing do, `for`, `switch`, or `while` statement.

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:



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.

```
/**
 ** This example shows a continue statement in a for statement.
 **/

#include <stdio.h>
#define  SIZE  5

int main(void)
{
   int i;
   static float rates[SIZE] = { 1.45, 0.05, 1.88, 2.00, 0.75 };

   printf("Rates over 1.00\n");
   for (i = 0; i < SIZE; i++)
   {
      if (rates[i] <= 1.00)  /*  skip rates <= 1.00  */
         continue;
      printf("rate = %.2f\n", rates[i]);
   }

   return(0);
}
```

The program produces the following output:

```
Rates over 1.00
rate = 1.45
rate = 1.88
rate = 2.00
```

The following example shows a `continue` statement in a nested loop. When the inner loop encounters a number in the array `strings`, that iteration of the loop ends. Processing continues with the third expression of the inner loop. The inner loop ends when the '\0' escape sequence is encountered.

```
/**
 ** This program counts the characters in strings that are part
 ** of an array of pointers to characters.  The count excludes
 ** the digits 0 through 9.
 **/

#include <stdio.h>
#define  SIZE  3

int main(void)
{
   static char *strings[SIZE] = { "ab", "c5d", "e5" };
   int i;
   int letter_count = 0;
   char *pointer;
   for (i = 0; i < SIZE; i++)             /* for each string   */
                                           /* for each each character */
      for (pointer = strings[i]; *pointer != '\0';
      ++pointer)
      {                                   /* if a number      */
         if (*pointer >= '0' && *pointer <= '9')
            continue;
         letter_count++;
      }
   printf("letter count = %d\n", letter_count);

   return(0);
}
```

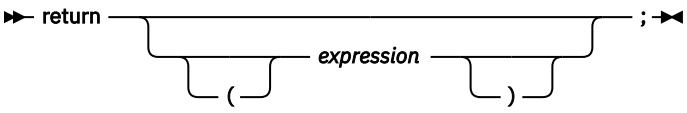
The program produces the following output:

```
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



A value-returning function should include a `return` statement, containing an *expression*. C++ 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. C++ 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[1];
                        /* 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.

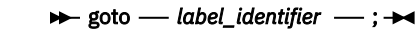
Related information

- “Function return type specifiers” on page 202
- “Function return values” on page 203

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



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.

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.

```
/**
 ** This example shows a goto statement that is used to
 ** jump out of a nested loop.
 **/

#include <stdio.h>
void display(int matrix[3][3]);

int main(void)
{
   int matrix[3][3]=    {1,2,3,4,5,2,8,9,10};
   display(matrix);
   return(0);
}

void display(int matrix[3][3])
{
   int i, j;

   for (i = 0; i < 3; i++)
      for (j = 0; j < 3; j++)
      {
         if ( (matrix[i][j] < 1) || (matrix[i][j] > 6) )
            goto out_of_bounds;
         printf("matrix[%d][%d] = %d\n", i, j, matrix[i][j]);
      }
   return;
   out_of_bounds: printf("number must be 1 through 6\n");
}
```

Related information

- “Labeled statements” on page 175

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 */
}
```

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" on page 191
- "Function storage class specifiers" on page 197
- "Function specifiers" on page 199
- "Function return type specifiers" on page 202
- "Function declarators" on page 204
- "Function attributes" on page 209
- "The main() function" on page 212
- "Function calls" on page 213
- "Default arguments in C++ functions (C++ only)" on page 217
- "Pointers to functions" on page 219

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.

© Copyright IBM Corp. 1993, 2015

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" on page 197, which specify linkage
- "Function return type specifiers" on page 202, which specify the data type of a value to be returned
- "Function specifiers" on page 199, which specify additional properties for functions
- "Function declarators" on page 204, which include function identifiers as well as lists of parameters

All function declarations have the form:

Function declaration syntax

```
>>-+------------------------+--+--------------------+---------->
   '-storage_class_specifier-'  '-function_specifier-'

>--return_type_specifier--function_declarator--;--------------><
```

C++11 **Note:** When *function_declarator* incorporates a trailing return type, *return_type_specifer* must be `auto`. For more information about trailing return type, see "Trailing return type (C++11)" on page 207.

IBM i In addition, for compatibility with C++, you can use *attributes* to modify the properties of functions. They are described in "Function attributes" on page 209.

Function definitions

The elements of a function definition are as follows:

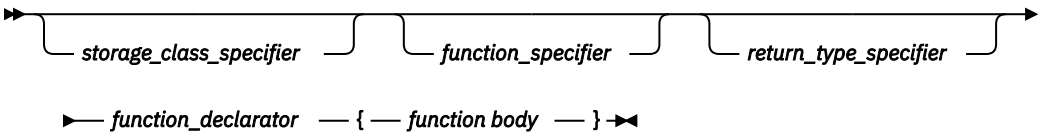
- "Function storage class specifiers" on page 197, which specify linkage
- "Function return type specifiers" on page 202, which specify the data type of a value to be returned
- "Function specifiers" on page 199, which specify additional properties for functions
- "Function declarators" on page 204, 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 300.
- C++ Try blocks, which are used in class functions; they are described in "try blocks (C++ only)" on page 367.

IBM i In addition, for compatibility with C++, you can use *attributes* to modify the properties of functions. They are described in "Function attributes" on page 209.

Function definitions take the following form:

C *Beginning of C only.*

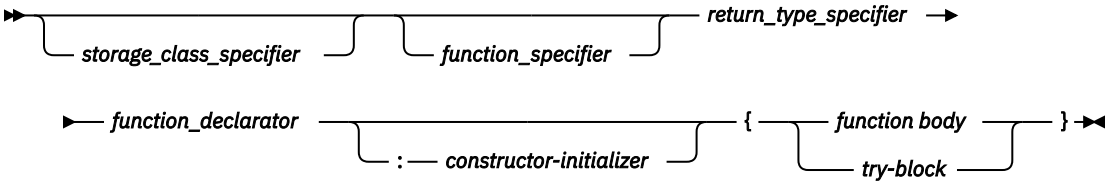
Function definition syntax



C *End of C only.*

C++ *Beginning of C++ only.*

Function definition syntax



C++11 **Note:** When function_declarator incorporates a trailing return type, return_type_specifer must be `auto`. For more information about trailing return type, see "Trailing return type (C++11)" on page 207.

C++ *End of C++ only.*

Explicitly defaulted functions (C++11)

Note: IBM supports selected features of C++11, known as C++0x before its ratification. IBM will continue to develop and implement the features of this standard. The implementation of the language level is based on IBM's interpretation of the standard. Until IBM's implementation of all the C++11 features is complete, including the support of a new C++11 standard library, the implementation may change from release to release. IBM makes no attempt to maintain compatibility, in source, binary, or listings and other compiler interfaces, with earlier releases of IBM's implementation of the new C++11 features.

Explicitly defaulted function declaration is a new form of function declaration that is introduced into the C++11 standard. You can append the `=default;` specifier to the end of a function declaration to declare that function as an explicitly defaulted function. The compiler generates the default implementations for explicitly defaulted functions, which are more efficient than manually programmed function implementations. A function that is explicitly defaulted must be a special member function and has no default arguments. Explicitly defaulted functions can save your effort of defining those functions manually.

You can declare both inline and out-of-line explicitly defaulted functions. For example:

```
class A{
public:
   A() = default;            // Inline explicitly defaulted constructor definition
   A(const A&);
   ~A() = default;           // Inline explicitly defaulted destructor definition
};

A::A(const A&) = default;    // Out-of-line explicitly defaulted constructor definition
```

You can declare a function as an explicitly defaulted function only if the function is a special member function and has no default arguments. For example:

```
class B {
public:
  int func() = default;     // Error, func is not a special member function.
  B(int, int) = default;    // Error, constructor B(int, int) is not
                            // a special member function.
  B(int=0) = default;       // Error, constructor B(int=0) has a default argument.
};
```

The explicitly defaulted function declarations enable more opportunities in optimization, because the compiler might treat explicitly defaulted functions as trivial.

Related information

- "Deleted functions (C++11)" on page 194
- "Special member functions (C++ only)" on page 299

Deleted functions (C++11)

Note: IBM supports selected features of C++11, known as C++0x before its ratification. IBM will continue to develop and implement the features of this standard. The implementation of the language level is based on IBM's interpretation of the standard. Until IBM's implementation of all the C++11 features is complete, including the support of a new C++11 standard library, the implementation may change from release to release. IBM makes no attempt to maintain compatibility, in source, binary, or listings and other compiler interfaces, with earlier releases of IBM's implementation of the new C++11 features.

Deleted function declaration is a new form of function declaration that is introduced into the C++11 standard. To declare a function as a deleted function, you can append the `=delete;` specifier to the end of that function declaration. The compiler disables the usage of a deleted function.

You can declare an implicitly defined function as a deleted function if you want to prevent its usage. For example, you can declare the implicitly defined copy assignment operator and copy constructor of a class as deleted functions to prevent object copy of that class.

```
class A{
public:
  A(int x) : m(x) {}
  A& operator = (const A &) = delete;     // Declare the copy assignment operator
                                          // as a deleted function.
  A(const A&) = delete;                   // Declare the copy constructor
                                          // as a deleted function.

private:
  int m;
};

int main(){
  A a1(1), a2(2), a3(3);
  a1 = a2;            // Error, the usage of the copy assignment operator is disabled.
  a3 = A(a2);         // Error, the usage of the copy constructor is disabled.
}
```

You can also prevent problematic conversions by declaring the undesirable conversion constructors and operators as deleted functions. The following example shows how to prevent undesirable conversions from `double` to a class type.

```
class B{
public:
  B(int){}
  B(double) = delete;   // Declare the conversioin constructor as a deleted function
};

int main(){
  B b1(1);
  B b2(100.1);          // Error, conversion from double to class B is disabled.
}
```

A deleted function is implicitly inline. A deleted definition of a function must be the first declaration of the function. For example:

```
class C {
public:
  C();
};

C::C() = delete;    // Error, the deleted definition of function C must be
                    // the first declaration of the function.
```

Related information

- “Explicitly defaulted functions (C++11)” on page 193

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 45

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 233

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. C An exception is inline functions, which are treated by default as having internal linkage; see "Linkage of inline functions" on page 200 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`.

C++ This use of `static` is deprecated in C++. Instead, place the function in the unnamed namespace.

Related information

- "Internal linkage" on page 17
- "Namespaces (C++ only)" on page 223

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.

C++ *Beginning of C++ only.*

An `extern` declaration cannot appear in class scope.

You can use the `extern` keyword with arguments that specify the type of linkage.

extern function storage class specifier syntax

All platforms support the following values for *linkage_specification*:

- C
- C++

IBM i See "Working with Multi-Language Applications" in the *ILE C/C++ Programmer's Guide* for additional language linkages supported by ILE C++.

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 function 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, the following series of statements is allowed:

  ```
  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 "C"{int func(int);}
  extern "C"{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 "C"{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 "C"{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
```

C++ *End of C++ only.*

Related information

- “External linkage” on page 17
- “Language linkage” on page 18
- “Class scope (C++ only)” on page 14
- “Namespaces (C++ only)” on page 223

Function specifiers

The available function specifiers for function definitions are:

- C++11 `constexpr`, which can be used to declare constexpr functions and constexpr constructors, and is described in “The constexpr specifier (C++11)” on page 63.
- `inline`, which instructs the compiler to expand a function definition at the point of a function call.
- C++ `explicit`, which can only be used for member functions of classes, and is described in “Explicit conversion constructors (C++ only)” on page 315.
- C++ `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” on page 200 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: The `inline` specifier is represented by the following keywords:

- C The `__inline__` keyword is supported at all language levels. C99 adds support for the `inline` keyword.
- C++ The `inline` and `__inline__` keywords are recognized at all language levels.

Related information

- "The noinline function attribute" on page 211
- **LANGLVL** in the *ILE C/C++ Compiler Reference*

Linkage of inline functions

C *Beginning of 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 and `foo` in file B are treated as separate functions: two function bodies are generated, and assigned two different addresses in memory:

```
// File A

#include <stdio.h>

__inline__ int foo(){
   return 3;
}

void g() {
   printf("foo called from g: return value = %d, address = %#p\n", foo(), &foo);
}


// File B

#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 = A100000000000000D8ED5D51EA000B68
foo called from g: return value = 3, address = A100000000000000D8ED5D51EA000B58
```

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:

```
// File A

#include <stdio.h>

__inline__ int foo(){
   return 6;
}

void g() {
   printf("foo called from g: return value = %d\n", foo());
}


// File B

#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.

C *End of C only.*

C++ *Beginning of 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 =  A100000000000000D8ED5D51EA000B58
foo called from g: return value = 3, address = A100000000000000D8ED5D51EA000B68
```

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:

```
// File A

#include <stdio.h>

inline int foo(){
   return 6;
}

void g() {
   printf("foo called from g: return value = %d, address = %#p\n", foo(), &foo);
}


// File B

#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 file A and file B are bound into a single ILE program, the output is:

```
foo called from main: return value = 6, address = A100000000000000F3551B782F000B38
foo called from g: return value = 6, address = A100000000000000F3551B782F000B38
```

The call to `foo` from `main` does not use the inline definition provided in file B, but rather calls `foo` as a regular external function defined in file A. 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:

```
// File A

#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());
}


// File B

#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 = A100000000000000F3551B782F000B38
foo called from main: return value = 24
address of x = A100000000000000F3551B782F000B38
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. 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 197
- "The extern storage class specifier" on page 197

C++ *End of C++ 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 53

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.

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 188
- “Overloading assignments (C++ only)” on page 238

- "Overloading subscripting (C++ only)" on page 240
- "The auto storage class specifier" on page 49

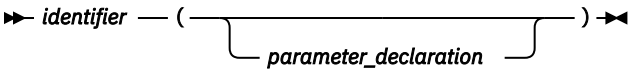
Function declarators

Function declarators consist of the following elements:

- An *identifier*, or name
- "Parameter declarations" on page 204, 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 "Exception handling (C++ only)" on page 367.
- C++ The type qualifiers `const` and `volatile`, which are used only in class member functions; they are described in "Constant and volatile member functions" on page 258.

C *Beginning of C only.*

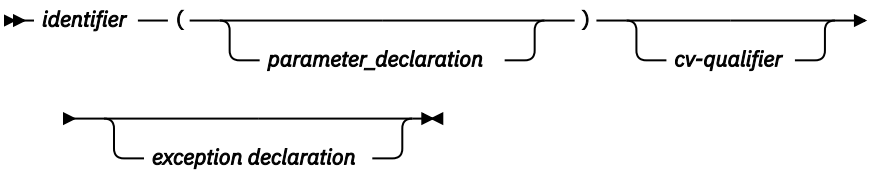
Function declarator syntax



C *End of C only.*

C++ *Beginning of C++ only.*

Function declarator syntax



C++ *End of C++ only.*

Related information

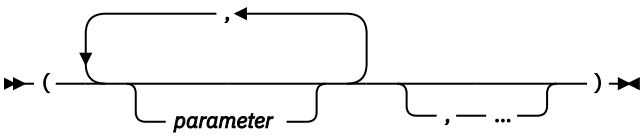
- "Default arguments in C++ functions (C++ only)" on page 217

Parameter declarations

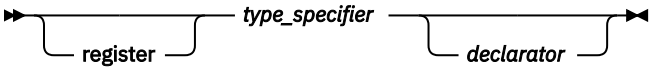
The function declarator includes the list of parameters that can be passed to the function when it is called by another function, or by itself.

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



C++ *Beginning of C++ only.*

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();
```

C++ *End of C++ only.*

C *Beginning of C only.*

An empty argument list in a function *definition* indicates 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`.

C *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 57
- “Type specifiers” on page 53
- “Type qualifiers” on page 79
- “Exception specifications (C++ only)” on page 378

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++ *Beginning of 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);
  ```

C++ *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, C99 allows you to 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 runtime.        */
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 49
- “Arrays” on page 97
- “Array subscripting operator [ ]” on page 149

Trailing return type (C++11)

Note: IBM supports selected features of C++11, known as C++0x before its ratification. IBM will continue to develop and implement the features of this standard. The implementation of the language level is based on IBM's interpretation of the standard. Until IBM's implementation of all the C++11 features is complete, including the support of a new C++11 standard library, the implementation may change from release to release. IBM makes no attempt to maintain compatibility, in source, binary, or listings and other compiler interfaces, with earlier releases of IBM's implementation of the new C++11 features.

The trailing return type feature removes a C++ limitation where the return type of a function template cannot be generalized if the return type depends on the types of the function arguments. For example, a and b are arguments of a function template `multiply(const A &a, const B &b)`, where a and b are of arbitrary types. Without the trailing return type feature, you cannot declare a return type for the `multiply` function template to generalize all the cases for a*b. With this feature, you can specify the return type after the function arguments. This resolves the scoping problem when the return type of a function template depends on the types of the function arguments.

```
Trailing return type syntax

>>-function_identifier--(--+-----------------------+--)--------->
                           '-parameter_declaration-'

>--+------------------+--+-----------------------+-------------->
   '-cv_qualifier_seq-'  '-exception_declaration-'

>-- -> -return_type-------------------------------------------><
```

Note:

- This syntax is not the one for a function declaration or definition. The auto placeholder occurs in the syntax for declarations and definitions where they specify return_type_specifier.
- As with function declarators without a trailing return type, this syntax might be used to declare a pointer or reference to function.
- More complex types might be formed by using the syntax of direct_declarator in place of function_identifier. For the details of direct_declarator, see “Overview of declarators” on page 89.

To use the trailing return type feature, declare a generic return type with the `auto` keyword before the function identifier, and specify the exact return type after the function identifier. For example, use the `decltype` keyword to specify the exact return type.

In the following example, the `auto` keyword is put before the function identifier add. The return type of add is `decltype(a + b)`, which depends on the types of the function arguments a and b.

```
// Trailing return type is used to represent
// a fully generic return type for a+b.
template <typename FirstType, typename SecondType>
auto add(FirstType a, SecondType b) -> decltype(a + b){
   return a + b;
}

int main(){
   // The first template argument is of the integer type, and
   // the second template argument is of the character type.
   add(1, 'A');

   // Both the template arguments are of the integer type.
   add(3, 5);
}
```

Note:

- When a trailing return type is used, the placeholder return type must be `auto`. For example, the statement `auto *f()->char` results in a compile-time error, because `auto *` is not allowed as the placeholder return type.
- The `auto` type specifier can be used with a function declarator with a trailing return type. Otherwise, the `auto` type specifier is used in accordance to the auto type deduction feature. For more information about auto type deduction, "The auto type specifier (C++11)" on page 57. Because a function declaration cannot have an initializer as required for auto type deduction, the `auto` type specifier cannot be used in a function declaration without a trailing return type. For declarations of pointers and references to functions, the `auto` type specifier can be used with either a corresponding trailing return type or an initializer. For details of pointers and references to functions, see "Pointers to functions" on page 219.
- The return type of a function cannot be any of the following types:
  - Function
  - Array
  - Incomplete class
- The return type of a function cannot define any of the following types:
  - `struct`
  - `class`
  - `union`
  - `enum`

  However, the return type can be any of these types if the type is not defined in the function declaration.

In addition, this feature makes your program more compact and elegant in cases where functions have complicated return types. Without this feature, programs might be complicated and error prone. See the following example:

```
template <class A, class B> class K{
   public:
      int i;
};

K<int, double> (*(*bar())())() {
   return 0;
}
```

You can use the trailing return type feature to make the code compact. See the following example:

```
template <class A, class B> class K{
   public:
      int i;
};

auto bar()->auto(*)()->K<int, double>(*)(){
   return 0;
}
```

This feature can also be used for member functions of classes. In the following example, the program is concise because the return type of the member function `bar`does not need to be qualified after using a trailing return type:

```
struct A{
   typedef int ret_type;
   auto bar() -> ret_type;
};

// ret_type is not qualified
auto A::bar() -> ret_type{
   return 0;
}
```

Another use of this feature is in writing perfect forwarding functions. That is, the forwarding function calls another function, and the return type of the forwarding function is the same as that of the called function. See the following example:

```
double number (int a){
   return double(a);
}

int number(double b){
   return int(b);
}

template <class A>
auto wrapper(A a) -> decltype(number(a)){
   return number(a);
}

int main(){
   // The return value is 1.000000.
   wrapper(1);

   // The return value is 1.
   wrapper(1.5);
}
```

In this example, the `wrapper` function and the `number` function have the same return type.

Related information

- “The auto type specifier (C++11)” on page 57
- “The decltype(expression) type specifier (C++11)” on page 59
- “Extensions for C++11 compatibility” on page 411
- “Member functions (C++ only)” on page 256
- “Member functions of class templates (C++ only)” on page 333
- “Function templates (C++ only)” on page 334
- “Function declarations” on page 191
- “Function definitions” on page 192
- “Pointers to functions” on page 219

Function attributes

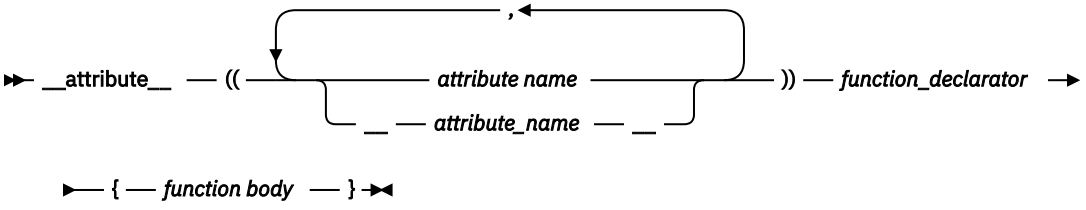
IBM i *Beginning of IBM Extension.*

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:

C *Beginning of C only.*

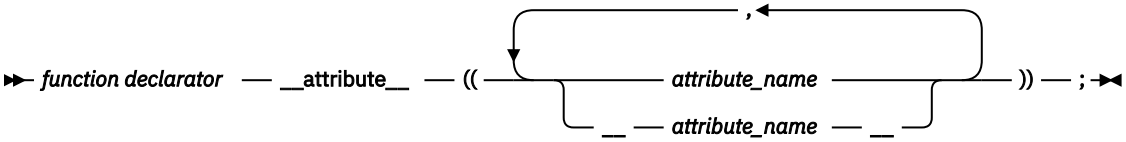
Function attribute syntax: function definition



C *End of C only.*

C++ *Beginning of C++ only.*

Function attribute syntax: function definition



C++ *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++ *Beginning of C++ only.*

In C++, the attribute specification must also follow any exception declaration that may be present for the function.

C++ *End of C++ only.*

C *Beginning of 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) { }
```

C *End of C only.*

C++ 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.

The following function attributes are supported:

- "The noinline function attribute" on page 211
- "The pure function attribute" on page 211
- "The weak function attribute" on page 211

Related information

- "Variable attributes" on page 113

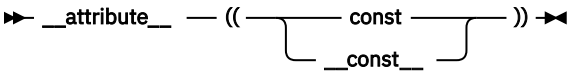
IBM i *End of IBM Extension.*

The const function attribute

IBM i *Beginning of IBM Extension.*

The `const` function attribute allows you to tell the compiler that the function can safely be called fewer times than indicated in the source code. The language feature provides you with an explicit way to help the compiler optimize code by indicating that the function does not examine any values except its arguments and has no effects except for its return value.

const function attribute syntax



The following kinds of functions should not be declared `const`:

- A function with pointer arguments which examines the data pointed to.
- A function that calls a non-`const` function.

Related information

- **#pragma isolated_call** in the *ILE C/C++ Compiler Reference*

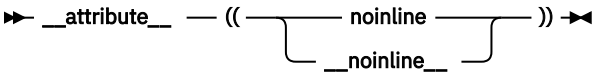
IBM i *End of IBM Extension.*

The noinline function attribute

IBM i *Beginning of IBM Extension.*

The `noinline` function attribute prevents the function to which it is applied from being inlined, regardless of whether the function is declared inline or non-inline. The attribute takes precedence over inlining compiler options and the `inline` keyword.

noinline function attribute syntax



Other than preventing inlining, the attribute does not remove the semantics of inline functions.

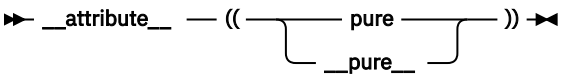
IBM i *End of IBM Extension.*

The pure function attribute

IBM i *Beginning of IBM Extension.*

The `pure` function attribute allows you to declare a function that can be called fewer times than what is literally in the source code. Declaring a function with the attribute `pure` indicates that the function has no effect except a return value that depends only on the parameters, global variables, or both.

pure function attribute syntax



Related information

- **#pragma isolated_call** in the *ILE C/C++ Compiler Reference*

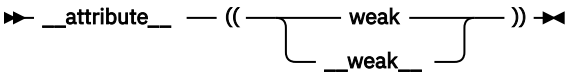
IBM i *End of IBM Extension.*

The weak function attribute

IBM i *Beginning of IBM Extension.*

The `weak` function attribute causes the symbol resulting from the function declaration to appear in the object file as a weak symbol, rather than a global one. The language feature provides the programmer writing library functions with a way to allow function definitions in user code to override the library function declaration without causing duplicate name errors.

weak function attribute syntax



Related information

- **#pragma weak** in the *ILE C/C++ Compiler Reference*
- "The weak variable attribute" on page 116

IBM i *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`.
- C++ `main` cannot be called from within a program.
- C++ The address of `main` cannot be taken.
- C++ The `main` function cannot be overloaded.
- C++11 The `main` function cannot be declared with the `constexpr` specifier.

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
`argc`	3
`argv[0]`	pointer to string "`backward`"
`argv[1]`	pointer to string "`string1`"
`argv[2]`	pointer to string "`string2`"
`argv[3]`	`NULL`

Related information

- "The extern storage class specifier" on page 50
- "The inline function specifier" on page 199
- "The static storage class specifier" on page 49
- "Function calls" on page 213

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" on page 214, which copies the *value* of an argument to the corresponding parameter in the called function
- "Pass by reference" on page 214, which passes the *address* of an argument to the corresponding parameter in the called function

C++ *Beginning of 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.

C++ *End of C++ only.*

C A function call expression is always an rvalue.

C++ A function call belongs to one of the following value categories depending on the result type of the function:

- An lvalue if the result type is an lvalue reference type or C++11 an rvalue reference to a function type
- C++11 An xvalue if the result type is an rvalue reference to an object type
- A C++11 (prvalue) rvalue in other cases

Related information

- “Function argument conversions” on page 122
- “Function call expressions” on page 132
- “Constructors (C++ only)” on page 300

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 *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.

C++ *Beginning of 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;
}
```

C++ *End of C++ only.*

C *Beginning of 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;
}
```

C *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++ *Beginning of 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.

C++ *End of C++ only.*

Related information

- “References (C++ only)” on page 99

- "The const_cast operator (C++ only)" on page 159

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 16 bytes
Constructor of S
Destructor of S
delete object
Array of size 2:
operator new with 48 bytes
Constructor of S
Constructor of S
Destructor of S
```

```
Destructor of S
delete object
```

Related information

- “new expressions (C++ only)” on page 163

Default arguments in C++ functions (C++ only)

You can provide default values for function parameters. For example:

```
#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 219

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 132
- “new expressions (C++ only)” on page 163

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.

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.

C++11 *Beginning of C++11 only.*

You can use a trailing return type in the declaration or definition of a pointer to a function. For example:

```
auto(*fp)()->int;
```

In this example, fp is a pointer to a function that returns int. You can rewrite the declaration of fp without using a trailing return type as int (*fp)(void). For more information about trailing return type, see "Trailing return type (C++11)" on page 207.

C++11 *End of C++11 only.*

Related information

- "Language linkage" on page 18
- "Pointers" on page 92
- "Pointer conversions" on page 120
- "The extern storage class specifier" on page 197

Constexpr functions (C++11)

Note: IBM supports selected features of C++11, known as C++0x before its ratification. IBM will continue to develop and implement the features of this standard. The implementation of the language level is based on IBM's interpretation of the standard. Until IBM's implementation of all the C++11 features is complete, including the support of a new C++11 standard library, the implementation may change from release to release. IBM makes no attempt to maintain compatibility, in source, binary, or listings and other compiler interfaces, with earlier releases of IBM's implementation of the new C++11 features.

A non-constructor function that is declared with a `constexpr` specifier is a `constexpr` function. A `constexpr` function is a function that can be invoked within a constant expression.

A `constexpr` function must satisfy the following conditions:

- It is not virtual.
- Its return type is a literal type.
- Each of its parameters must be of a literal type.
- When initializing the return value, each constructor call and implicit conversion is valid in a constant expression.
- Its function body is `= delete` or `= default`; otherwise, its function body must contain only the following statements:
  - `null` statements
  - `static_assert` declarations
  - `typedef` declarations that do not define classes or enumerations
  - `using` directives
  - `using` declarations
  - One `return` statement

When a nonstatic member function that is not a constructor is declared with the `constexpr` specifier, that member function is constant, and the `constexpr` specifier has no other effect on the function type. The class of which that function is a member must be a literal type.

The following examples demonstrate the usage of `constexpr` functions:

```
const int array_size1 (int x) {
  return x+1;
}
// Error, constant expression required in array declaration
int array[array_size1(10)];

constexpr int array_size2 (int x) {
  return x+1;
}
// OK, constexpr functions can be evaluated at compile time
// and used in contexts that require constant expressions.
int array[array_size2(10)];

struct S {
  S() { }
  constexpr S(int) { }
  constexpr virtual int f() { // Error, f must not be virtual.
    return 55;
  }
};

struct NL {
  ~NL() { }              // The user-provided destructor (even if it is trivial)
                         // makes the type a non-literal type.
};

constexpr NL f1() {          // Error, return type of f1 must be a literal type.
  return NL();
}

constexpr int f2(NL) {       // Error, the parameter type NL is not a literal type.
  return 55;
```

```
}

constexpr S f3() {
  return S();
}

enum { val = f3() };         // Error, initialization of the return value in f3()
                             // uses a non-constexpr constructor.

constexpr void f4(int x) {   // Error, return type should not be void.
  return;
}

constexpr int f5(int x) {    // Error, function body contains more than return statement.
  if (x<0)
    x = -x;
  return x;
}
```

When a function template is declared as a `constexpr` function, if the instantiation results in a function that does not satisfy the requirements of a `constexpr` function, the `constexpr` specifier is ignored. For example:

```
template <class C> constexpr NL f6(C c) {    // OK, the constexpr specifier ignored
  return NL();
}
void g() {
  f6(55); // OK, not used in a constant expression
}
```

A call to a `constexpr` function produces the same result as a call to an equivalent non-`constexpr` function in all respects, except that a call to a `constexpr` function can appear in a constant expression.

A `constexpr` function is implicitly inline.

The `main` function cannot be declared with the `constexpr` specifier.

Related information

- “The constexpr specifier (C++11)” on page 63
- “Generalized constant expressions (C++11)” on page 131

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.

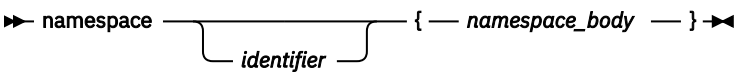
Related information

- "Scope resolution operator :: (C++ only)" on page 130

Defining namespaces (C++ only)

In order to uniquely identify a namespace, use the **namespace** keyword.

Namespace syntax



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 225
- "Inline namespace definitions (C++11)" on page 229

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 13

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.

© Copyright IBM Corp. 1993, 2015

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 13

Creating an alias for a nested namespace

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 223

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

- "Overloading (C++ only)" on page 233

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 16
- “Internal linkage” on page 17

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 269

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

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” on page 281
- “Inline namespace definitions (C++11)” on page 229

The using declaration and namespaces

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

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.

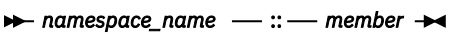
Related information

- “The using declaration and class members” on page 281

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



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 130

Inline namespace definitions (C++11)

Note: IBM supports selected features of C++11, known as C++0x before its ratification. IBM will continue to develop and implement the features of this standard. The implementation of the language level is based on IBM's interpretation of the standard. Until IBM's implementation of all the C++11 features are complete, including the support of a new C++11 standard library, the implementation may change from release to release. IBM makes no attempt to maintain compatibility, in source, binary, or listings and other compiler interfaces, with earlier releases of IBM's implementation of the new C++11 features and therefore they should not be relied on as a stable programming interface.

Inline namespace definitions are namespace definitions with an initial inline keyword. A namespace so defined is an inline namespace. You can define and specialize members of an inline namespace as if they were also members of the enclosing namespace.

Inline namespace definitions syntax

When an inline namespace is defined, a using directive is implicitly inserted into its enclosing namespace. While looking up a qualified name through the enclosing namespace, members of the inline namespace are brought in and found by the implicit using directive, even if that name is declared in the enclosing namespace.

For example, if you compile the following code with USE_INLINE_B defined, the output of the resulting executable is 1; otherwise, the output is 2.

```
namespace A {
#if USE_INLINE_B
    inline
#endif
    namespace B {
    int foo(bool) { return 1; }
    }
    int foo(int) { return 2; }
}
int main(void) {
    return A::foo(true);
}
```

The properties of inline namespace definitions are transitive; that is, you can use members of an inline namespace as if they were also members of any namespace in its enclosing namespace set, which consists of the innermost non-inline namespace enclosing the inline namespace, together with any intervening inline namespaces. For example:

```
namespace L {
    inline namespace M {
        inline namespace N {
            / *...* /
        }
    }
}
```

In this example, a namespace L contains an inline namespace M, which in turn contains another inline namespace N. The members of N can also be used as if they were members of the namespaces in its enclosing namespace set, i.e., L and M.

Note:

- Do not declare the namespace std, which is used for the C++ standard library, as an inline namespace.
- Do not declare a namespace to be an inline namespace if it is not inline in its first definition.
- You can declare an unnamed namespace as an inline namespace.

Using inline namespace definitions in explicit instantiation and specialization

You can explicitly instantiate or specialize each member of an inline namespace as if it were a member of its enclosing namespace. Name lookup for the primary template of an explicit instantiation

or specialization in a namespace, for example M, considers the inline namespaces whose enclosing namespace set includes M.

For example:

```
namespace L {
    inline namespace M {
        template <typename T> class C;
    }
    template <typename T> void f(T) { /*...*/ };
}
struct X { /*...*/ };
namespace L {
    template<> class C<X> { /*...*/ }; //template specialization
}
int main()
{
    L::C<X> r;
    f(r); // fine, L is an associated namespace of C
    return 0;
}
```

In this example, M is an inline namespace of its enclosing namespace L, class C is a member of inline namespace M, so L is an associated namespace of class C.

The following rules apply when you use inline namespace definitions in explicit instantiation and specialization:

- An explicit instantiation must be in an enclosing namespace of the primary template if the template name is qualified; otherwise, it must be in the nearest enclosing namespace of the primary template or a namespace in the enclosing namespace set.
- An explicit specialization declaration must first be declared in the namespace scope of the nearest enclosing namespace of the primary template, or a namespace in the enclosing namespace set. If the declaration is not a definition, it may be defined later in any enclosing namespace.

Using inline namespace definitions in library versioning

With inline namespace definitions, you can provide a common source interface for a library with several implementations, and a user of the library can choose one implementation to be associated with the common interface. The following example demonstrates the use of inline namespace in library versioning with explicit specialization.

```
//foo.h
#ifndef SOME_LIBRARY_FOO_H_
    #define SOME_LIBRARY_FOO_H_
    namespace SomeLibrary
    {
        #ifdef SOME_LIBRARY_USE_VERSION_2_
            inline namespace version_2 { }
        #else
            inline namespace version_1 { }
        #endif
    namespace version_1 {
        template <typename T> int foo(T a) {return 1;}
    }
    namespace version_2 {
        template <typename T> int foo(T a) {return 2;}
    }
    }
#endif
//myFooCaller.C
#include ”foo.h”
#include <iostream>
struct MyIntWrapper { int x;};
//Specialize SomeLibrary::foo()
//Should specialize the correct version of foo()
namespace SomeLibrary {
    template <> int foo(MyIntWrapper a) { return a.x;}
}
int main(void) {
    using namespace SomeLibrary;
    MyIntWrapper intWrap = { 4 };
```

```
    std::cout << foo(intWrap) + foo(1.0) << std::endl;
}
```

If you compile this example with SOME_LIBRARY_USE_VERSION_2_ defined, the output of the resulting executable is 6; otherwise, the output is 5. If the function call, `foo(intWrap)`, is qualified with one of the inline namespaces, then you need to ensure that the explicit specialization is effective.

Related information

- “Defining namespaces (C++ only)” on page 223
- “Extending namespaces (C++ only)” on page 224
- “The using directive (C++ only)” on page 227
- “The using declaration and namespaces” on page 228
- “Explicit instantiation (C++ only)” on page 343
- “Explicit specialization (C++ only)” on page 345
- “Extensions for C++11 compatibility” on page 411

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)" on page 233 and "Overloading operators (C++ only)" on page 235, 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 242.

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" on page 234
- "Derivation (C++ only)" on page 276

© Copyright IBM Corp. 1993, 2015

Restrictions on overloaded functions

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” on page 228
- “typedef definitions” on page 77
- “Type qualifiers” on page 79
- “Language linkage” on page 18

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.

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 133

Overloading increment and decrement operators

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 134
- "Decrement operator --" on page 134

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 141

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 320
- “Assignment operators” on page 141

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 132

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 149

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 133

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++ runtime 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 243.

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” on page 243
- “User-defined conversion sequences” on page 244
- “Ellipsis conversion sequences” on page 244

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 120
- “Pointer conversions” on page 120
- “Qualification conversions (C++ only)” on page 122
- “Integral conversions” on page 118
- “Floating-point conversions ” on page 118
- “Boolean conversions” on page 118

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 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

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 219
- “Pointers to members (C++ only)” on page 259
- “Function templates (C++ only)” on page 334
- “Explicit specialization (C++ only)” on page 345

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

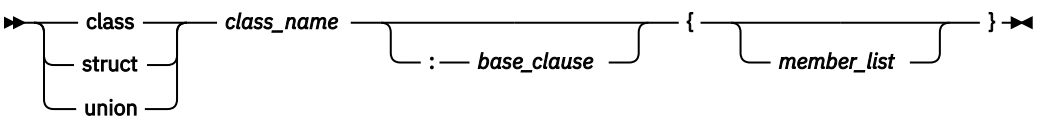
- “Structures and unions” on page 65
- “Class members and friends (C++ only)” on page 255
- “Inheritance (C++ only)” on page 275
- “Overloading (C++ only)” on page 233
- “Virtual functions (C++ only)” on page 291

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



© Copyright IBM Corp. 1993, 2015

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 255
- "Derivation (C++ only)" on page 276

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 65
- “References (C++ only)” on page 99
- “Scope of class names (C++ only)” on page 250

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:

```
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`:

```
#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

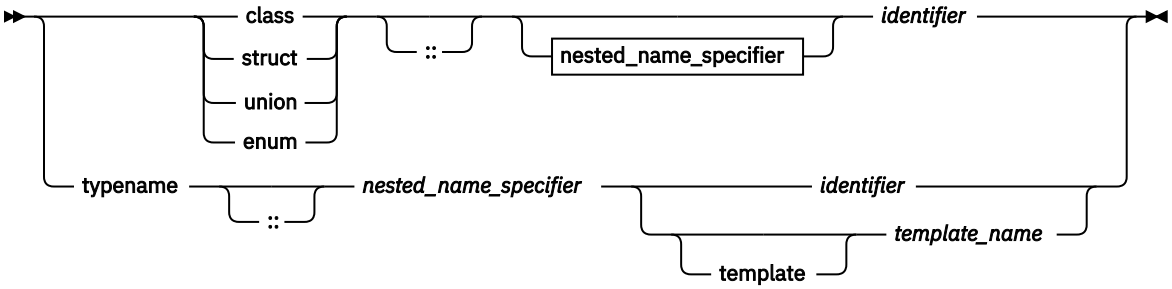
- "Structures and unions" on page 65

Scope of class names (C++ only)

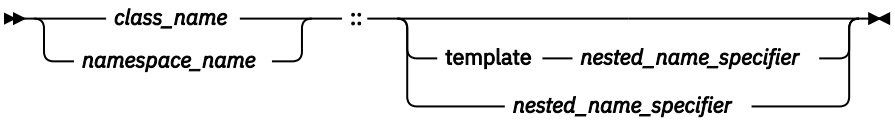
A class declaration introduces the class name into the scope where it is declared. Any class, object, function or other declaration of that name in an enclosing scope is hidden.

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 14
- "Incomplete class declarations (C++ only)" on page 251

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 255

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 14
- “Scope of class names (C++ only)” on page 250
- “Member access (C++ only)” on page 267
- “Static members (C++ only)” on page 263

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 256
- “The inline function specifier” on page 199

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 11
- “typedef definitions” on page 77

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.

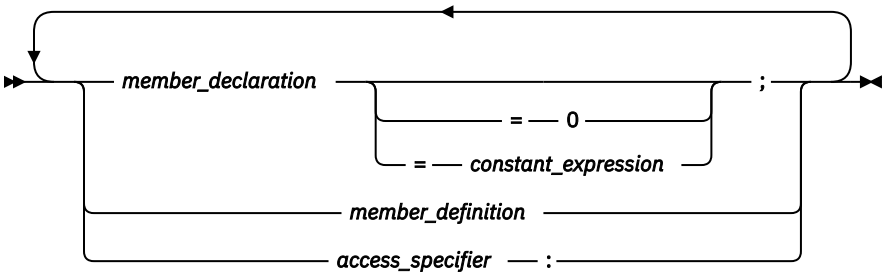
Related information

- "Member access (C++ only)" on page 267
- "Inherited member access (C++ only)" on page 279

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`.

© Copyright IBM Corp. 1993, 2015

A *member declaration* declares a class member for the class containing the declaration.

The order of allocation of nonstatic class members separated by an *access_specifier* is implementation-dependent.

The order of allocation of nonstatic class members separated by an *access_specifier* is implementation-dependent. The compiler allocates class members in the order in which they are declared.

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 247
- “Member access (C++ only)” on page 267
- “Inherited member access (C++ only)” on page 279
- “Static members (C++ only)” on page 263

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 267
- “Inherited member access (C++ only)” on page 279
- “Static members (C++ only)” on page 263

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 269
- “Static member functions (C++ only)” on page 265
- “The inline function specifier” on page 199
- “Local classes (C++ only)” on page 253

Constant and volatile member functions

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 79
- "The this pointer (C++ only)" on page 261

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 295

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 "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.

```
#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 14

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 152

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 by 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`:

```
#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**

``
#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();
}
``
	``
	#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);
	}
	``

Both examples produce the following output:

```
abcdefgh
abcdefghijkl
```

Related information

- “Overloading assignments (C++ only)” on page 238
- “Copy constructors (C++ only)” on page 319

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" on page 258
- "The static storage class specifier" on page 49
- "Class member lists (C++ only)" on page 255

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 49
- "Scope resolution operator :: (C++ only)" on page 130
- "Dot operator ." on page 132
- "Arrow operator ->" on page 133

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.

C++11 *Beginning of C++11 only.*

A static data member of a literal type can be declared with the `constexpr` specifier in the class definition, and the data member declaration must specify a constant initializer. For example:

```
struct Constants {
  static constexpr int bounds[] = { 42, 56 };
};

float a[Constants::bounds[0]][Constants::bounds[1]];
```

C++11 *End of C++11 only.*

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 17
- “Member access (C++ only)” on page 267
- “Local classes (C++ only)” on page 253

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:

```
#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 261

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 11
- “Class member lists (C++ only)” on page 255
- “Inherited member access (C++ only)” on page 279

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.

```
#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.

```
#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;
};
```

The compiler will warn you that the friend declaration of G must be an elaborated class name.

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.

C++11 *Beginning of C++11 only.*

Extended friend declarations

Note: IBM supports selected features of C++11, known as C++0x before its ratification. IBM will continue to develop and implement the features of the new standard. The implementation of the language level is based on IBM's interpretation of this standard. Until IBM's implementation of all the C++11 features are complete, including the support of a new C++ standard library, the implementation may change from release to release. IBM makes no attempt to maintain compatibility, in source, binary, or listings and other compiler interfaces, with earlier releases of IBM's implementation of the new C++11 features and therefore they should not be relied on as a stable programming interface.

Note: The syntactic form of extended friend declarations overlaps with the IBM old friend declaration syntax. This section is focused on the differences between the C++11 standard and the previous ISO C++ standard.

With this feature enabled, the class-key is no longer required in the context of friend declarations. This new syntax differs from the C++98 friend class declaration syntax, where the class-key is necessary as part of an elaborated-type-specifier. See the following example:

```
class F;
class G;
class X1 {
    //C++98 friend declarations remain valid in C++11.
    friend class F;
    //Error in C++98 for missing the class-key.
    friend G;
};
class X2 {
    //Error in C++98 for missing the class-key.
    //Error in C++11 for lookup failure (no previous class D declaration).
    friend D;
    friend class D;
};
```

In addition to functions and classes, you can also declare template parameters and basic types as friends. In this case, you cannot use an elaborated-type-specifier in the friend declaration. In the following example, you can declare the template parameter T as a friend of class F, and you can use the basic type `char` in friend declarations.

```
class C;
template <typename T, typename U> class F {
    //C++11 compiles sucessfully.
    //Error in C++98 for missing the class-key.
    friend T;
    //Error in both C++98 and C++11: a template parameter
    //must not be used in an elaborated type specifier.
    friend class U;
};
```

You can also declare `typedef` names as friends, but you still cannot use an elaborated-type-specifier in the friend declaration. The following example demonstrates that the `typedef` name D is declared as a friend of class `Base`.

```
class Derived;
typedef Derived D;
class C;
typedef C Ct;
class Base{
public:
    Base() : x(55) {}
    //C++11 compiles sucessfully.
    //Error in C++98 for missing the class-key.
    friend D;
    //Error in both C++98 and C++11: a typedef name
    //must not be used in an elaborated type specifier.
    friend class Ct;
private:
    int x;
};
struct Derived : public Base {
    int foo() { return this->x; }
};
int main() {
    Derived d;
    return d.foo();
}
```

This feature also introduces a new name lookup rule for friend declarations. If a friend class declaration does not use an elaborated-type-specifier, then the compiler also looks for the entity name in scopes outside the innermost namespace that encloses the friend declaration. Consider the following example:

```
struct T { };
namespace N {
    struct A {
        friend T;
    };
}
```

In this example, if this feature is in effect, the friend declaration statement does not declare a new entity T, but looks for T. If there is no T found, then the compiler issues an error. Consider another example:

```
struct T { };
namespace N {
    struct A {
        friend class T; //fine, no error
    };
}
```

In this example, the friend declaration statement does not look for T outside namespace N, nor does it find ::T. Instead, this statement declares a new class T in namespace N.

C++11 *End of C++11 only.*

Related information

- “Static member functions (C++ only)” on page 265
- “The inline function specifier” on page 199
- “Local classes (C++ only)” on page 253
- “Member access (C++ only)” on page 267
- “Inherited member access (C++ only)” on page 279
- “Extensions for C++11 compatibility” on page 411

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 250
- “Nested classes (C++ only)” on page 251
- “Local classes (C++ only)” on page 253

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 267

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.

© Copyright IBM Corp. 1993, 2015

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 285 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.)

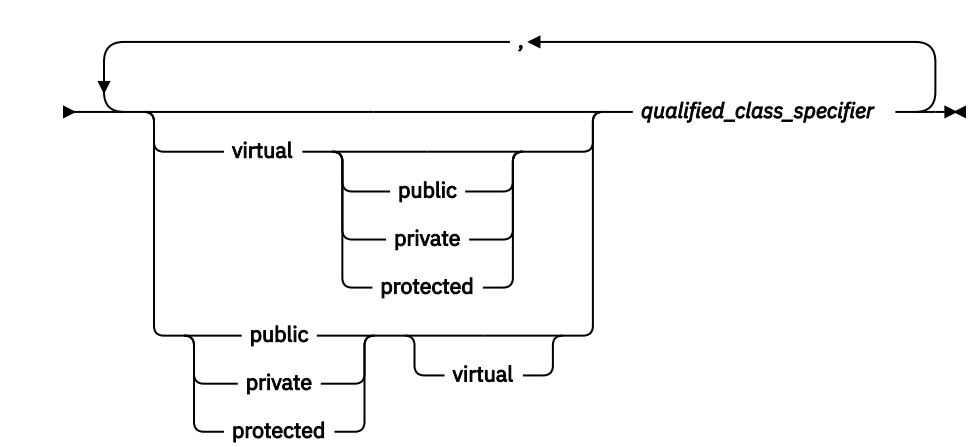
Polymorphic functions are functions that can be applied to objects of more than one type. In C++, polymorphic functions are implemented in two ways:

- 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 runtime. 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:

```
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:

```
#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:

```
#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 286
- “Inherited member access (C++ only)” on page 279
- “Incomplete class declarations (C++ only)” on page 251
- “Scope resolution operator :: (C++ only)” on page 130

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)” on page 279
- “Access control of base class members” on page 280

Related information

- “Member access (C++ only)” on page 267

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

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

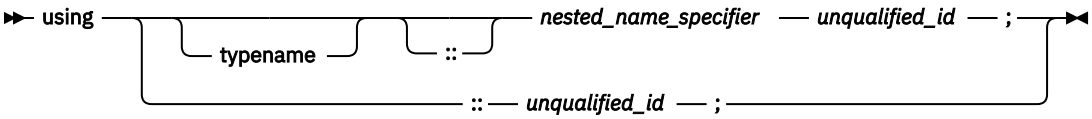
- “Member access (C++ only)” on page 267
- “Member scope (C++ only)” on page 258

The using declaration and class members

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



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 250
- "The using declaration and namespaces" on page 228

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

- “Overloading (C++ only)” on page 233
- “Name hiding (C++ only)” on page 15
- “The using declaration and class members” on page 281

Changing the access of a class member

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 267
- "Inherited member access (C++ only)" on page 279

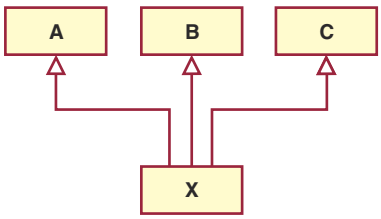
Multiple inheritance (C++ only)

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:

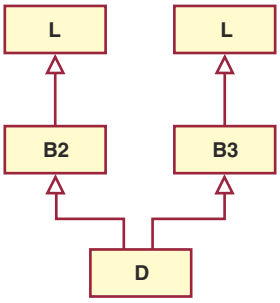


The order of derivation is relevant only to determine the order of default initialization by constructors and cleanup by destructors.

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.
```

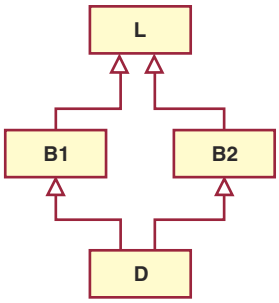
You can also avoid this ambiguity by using the base specifier `virtual` to declare a base class, as described in "Derivation (C++ only)" on page 276.

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.

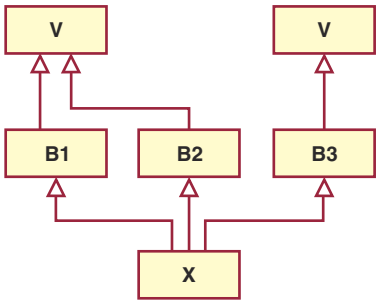
For example:




```
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 276

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 267
- “Protected members (C++ only)” on page 279
- “Access control of base class members” on page 280

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.

```
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 130
- “Virtual base classes (C++ only)” on page 286

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 runtime; 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 runtime. 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.

C++11 A function that has a deleted definition cannot override a function that does not have a deleted definition. Likewise, a function that does not have a deleted definition cannot override a function with a deleted definition.

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 295

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()`).

The compiler issues a warning for the direct call to the pure virtual function, but not for the indirect call.

Related information

- “Virtual functions (C++ only)” on page 291
- “Virtual function access (C++ only)” on page 295

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" on page 299
- "Constructors (C++ only)" on page 300
- "Destructors (C++ only)" on page 311
- "Conversion constructors (C++ only)" on page 314
- "Conversion functions (C++ only)" on page 317
- "Copy constructors (C++ only)" on page 319

C++11 A special member function is user-provided if it is user-declared but not explicitly defaulted, or deleted on its first declaration.

Overview of constructors and destructors

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

© Copyright IBM Corp. 1993, 2015

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 163
- “delete expressions (C++ only)” on page 166

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.

The compiler first implicitly defines the implicitly declared C++11 or explicitly defaulted constructors of the base classes and nonstatic data members of a class A before defining the implicitly declared C++11 or explicitly defaulted 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 C++11 or explicitly defaulted
- 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);
};
```

C++11 **Note:** You can declare default constructors as explicitly defaulted functions or deleted functions. For more information, see "Explicitly defaulted functions (C++11)" on page 193 and "Deleted functions (C++11)" on page 194.

Related information

- "Copy constructors (C++ only)" on page 319

Delegating constructors (C++11)

Note: IBM supports selected features of C++11, known as C++0x before its ratification. IBM will continue to develop and implement the features of this standard. The implementation of the language level is based on IBM's interpretation of the standard. Until IBM's implementation of all the C++11 features are complete, including the support of a new C++ standard library, the implementation may change from release to release. IBM makes no attempt to maintain compatibility, in source, binary, or listings and other compiler interfaces, with earlier releases of IBM's implementation of the new C++11 features and therefore they should not be relied on as a stable programming interface.

Before C++11, common initializations in multiple constructors of the same class could not be concentrated in one place in a robust, maintainable manner. To partially alleviate this problem in the existing C++ programs, you could use *assignment* instead of initialization or add a common initialization function.

With the delegating constructors feature, you can concentrate common initializations and post initializations in one constructor named target constructor. Delegating constructors can call the *target constructor* to do the initialization. A delegating constructor can also be used as the *target constructor* of one or more delegating constructors. You can use this feature to make programs more readable and maintainable.

Delegating constructors and target constructors present the same interface as other constructors. Target constructors do not need special handling to become the target of a delegating constructor. They are selected by overload resolution or template argument deduction. After the target constructor completes execution, the delegating constructor gets controls back.

In the following example, `A(T)` and `A(U)` both delegate to `A(T,U)`. This example demonstrates a typical usage of placing common initializations in a single constructor.

```
#include <cstdio>
template <typename T, typename U> struct A{
    const T t;
    const U u;
    static T tdef;
    static U udef;
    A(T t_, U u_): t(t_^u_), u(u_){}
    A(T t_):A(t_, udef){}
    A(U u_):A(tdef,u_){}
};
template <typename T, typename U>
T A<T,U>::tdef;
template <typename T, typename U>
U A<T,U>::udef;
int main(void){
    A<unsigned char, unsigned>::tdef = 42u & 0x0F;
    A<unsigned char, unsigned> a(42u & 0xF0);
    std::printf("%d\n", a.t);
    return 0;
}
```

The output of the example is:

```
42
```

In the example, `A(T)` and `A(U)` are delegating constructors, `A(T,U)` is the target constructor. The constant non-static data member `t` is initialized with an expression involving two parameters and the operator ^ in the non-delegating constructor.

A delegating constructor can be a target constructor of another delegating constructor, thus forming a delegating chain. The first constructor invoked in the construction of an object is called *principal constructor*. A constructor cannot delegate to itself directly or indirectly. The compiler can detect this violation if the constructors involved in a recursive chain of delegation are all defined in one translation unit. Consider the following example:

```
struct A{
    int x,y;
    A():A(42){}
    A(int x_):A() {x = x_;}
};
```

In the example, there is an infinitely recursive cycle that constructor `A()` delegates to constructor `A(int x_)`, and `A(int x_)` also delegates to `A()`. The compiler issues an error to indicate the violation.

You can use the delegating constructors feature interacting with other existing techniques:

- When several constructors have the same name, name and overload resolution can determine which constructor is the target constructor.
- When using delegating constructors in a template class, the deduction of template parameters works normally.

Related information

- “Extensions for C++11 compatibility” on page 411

Constexpr constructors (C++11)

Note: IBM supports selected features of C++11, known as C++0x before its ratification. IBM will continue to develop and implement the features of this standard. The implementation of the language level is based on IBM's interpretation of the standard. Until IBM's implementation of all the C++11 features is complete, including the support of a new C++11 standard library, the implementation may change from release to release. IBM makes no attempt to maintain compatibility, in source, binary, or listings and other compiler interfaces, with earlier releases of IBM's implementation of the new C++11 features.

A constructor that is declared with a `constexpr` specifier is a `constexpr` constructor. Previously, only expressions of built-in types could be valid constant expressions. With `constexpr` constructors, objects of user-defined types can be included in valid constant expressions.

Definitions of `constexpr` constructors must satisfy the following requirements:

- The containing class must not have any virtual base classes.
- Each of the parameter types is a literal type.
- Its function body is `= delete` or `= default`; otherwise, it must satisfy the following constraints:
  - It is not a function try block.
  - The compound statement in it must contain only the following statements:
    - `null` statements
    - `static_assert` declarations
    - `typedef` declarations that do not define classes or enumerations
    - `using` directives
    - `using` declarations
- Each nonstatic data member and base class subobject is initialized.
- Each constructor that is used for initializing nonstatic data members and base class subobjects is a `constexpr` constructor.

- Initializers for all nonstatic data members that are not named by a member initializer identifier are constant expressions.
- When initializing data members, all implicit conversions that are involved in the following context must be valid in a constant expression:
  - Calling any constructors
  - Converting any expressions to data member types

The implicitly defined default constructor performs the set of initializations of the class that would be performed by a user-written default constructor for that class with no initializer and an empty compound-statement. If that user-defined default constructor would satisfy the requirements of a `constexpr` constructor, the implicitly defined default constructor is a `constexpr` constructor.

A `constexpr` constructor is implicitly inline.

The following examples demonstrate the usage of `constexpr` constructors:

```
struct BASE {
};

struct B2 {
  int i;
};

//NL is a non-literal type.
struct NL {
  virtual ~NL() {
  }
};

int i = 11;

struct D1 : public BASE {
  //OK, the implicit default constructor of BASE is a constexpr constructor.
  constexpr D1() : BASE(), mem(55) { }

  //OK, the implicit copy constructor of BASE is a constexpr constructor.
  constexpr D1(const D1& d) : BASE(d), mem(55) { }

  //OK, all reference types are literal types.
  constexpr D1(NL &n) : BASE(), mem(55) { }

  //The conversion operator is not constexpr.
  operator int() const { return 55; }

private:
  int mem;
};

struct D2 : virtual BASE {
  //error, D2 must not have virtual base class.
  constexpr D2() : BASE(), mem(55) { }

private:
  int mem;
};

struct D3 : B2 {
  //error, D3 must not be a function try block.
  constexpr D3(int) try : B2(), mem(55) { } catch(int) { }

  //error, illegal statement is in body.
  constexpr D3(char) : B2(), mem(55) { mem = 55; }

  //error, initializer for mem is not a constant expression.
  constexpr D3(double) : B2(), mem(i) { }

  //error, implicit conversion is not constexpr.
  constexpr D3(const D1 &d) : B2(), mem(d) { }

  //error, parameter NL is a non-literal type.
  constexpr D3(NL) : B2(), mem(55) { }

private:
  int mem;
};
```

Related information

- “Generalized constant expressions (C++11)” on page 131
- “The constexpr specifier (C++11)” on page 63
- “Type specifiers” on page 53
- “Explicitly defaulted functions (C++11)” on page 193
- “Deleted functions (C++11)” on page 194

Explicit initialization with constructors (C++ only)

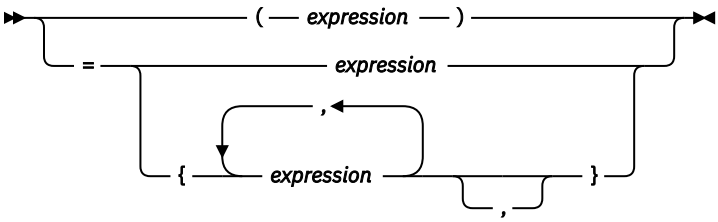
A class object with a constructor must be explicitly initialized or have a default constructor. Except for aggregate initialization, explicit initialization using a constructor is the only way to initialize non-static constant and reference class members.

A class object that has has only implicitly declared C++11 or explicitly defaulted constructors, 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:

```
// 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 100
- "Special member functions (C++ only)" on page 299

Initialization of base classes and members

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:

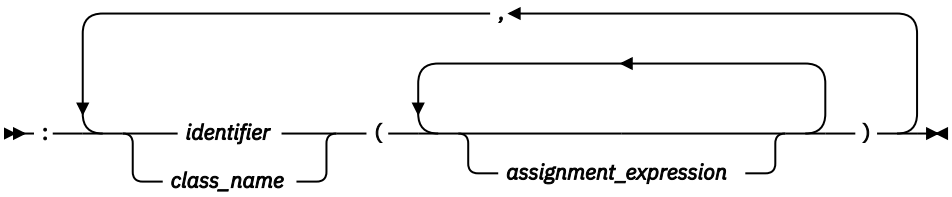
```
complx(double r, double i = 0.0) { re = r; im = i; }
```

Or a constructor can have an *initializer list* within the definition but prior to the constructor body:

```
complx(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 runtime 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.

C++0x *Beginning of C++0x only.*

If the delegating constructors feature is enabled, initialization can only be done within the non-delegating constructor. In other words, a delegating constructor cannot both delegate and initialize. Consider the following example:

```
struct A{
    int x,y;
    A(int x):x(x),y(0){}
    /* the following statement is not allowed */
    A():y(0),A(42) {}
};
```

Constructor `A()` delegates to `A(int x)`, but `A()` also does the initialization, which is not permitted. The compiler issues an error to indicate the violation.

For more information, see “Delegating constructors (C++11)” on page 302

C++0x *End of C++0x only.*

Related information

- “The typeid operator (C++ only)” on page 137
- “The dynamic_cast operator (C++ only)” on page 160

Construction order of derived class objects

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++ runtime calls the constructors to create an object of type D.

C++0x *Beginning of C++0x only.*

In the class body, if the delegating process exists, user code segments in the delegating constructors are executed after the completion of the target constructor. The inner most delegating constructor is executed first, then the next enclosing delegating constructor, until the outer most enclosing delegating constructor is executed.

Example:

```
#include <cstdio>
using std::printf;
class X{
public:
    int i,j;
    X();
    X(int x);
    X(int x, int y);
    ~X();
};
X::X(int x):i(x),j(23) {printf("X:X(int)\n");}
X::X(int x, int y): X(x+y) { printf("X::X(int,int)\n");}
X::X():X(44,11) {printf("X:X()\n");}
X::~X() {printf("X::~X()\n");}
int main(void){
    X x;
}
```

The output of the example is:

```
X::X(int)
X::X(int,int)
X:X()
X::~X()
```

For more information, see “Delegating constructors (C++11)” on page 302

C++0x *End of C++0x only.*

Related information

- “Virtual base classes (C++ only)” on page 286

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 C++11 or explicitly defaulteddestructors of the base classes and nonstatic data members of a class A before defining the implicitly declared C++11 or explicitly defaulted destructor of A.

A destructor of a class A is *trivial* if all the following are true:

- It is implicitly defined or C++11 or explicitly defaulted
- 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;
}
```

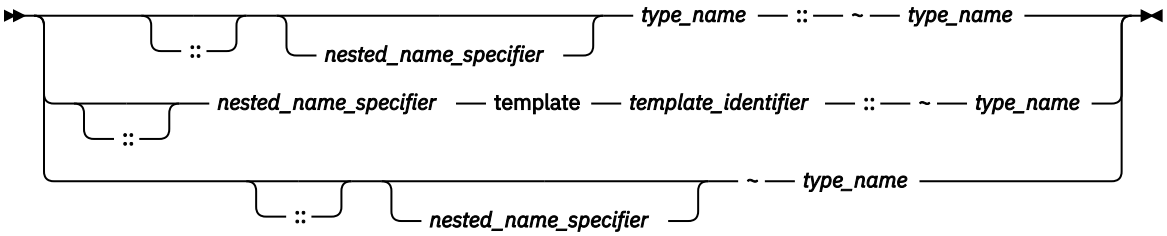
The statement `A* ap = new (p) A` dynamically creates a new object of type A not in the free store but in the memory allocated by p. The statement `delete [] p` will delete the storage allocated by p, but the runtime will still believe that the object pointed to by `ap` still exists until you explicitly call the destructor of A (with the statement `ap->A::~A()`).

C++11 **Note:** You can declare default constructors as explicitly defaulted functions or deleted functions. For more information, see “Explicitly defaulted functions (C++11)” on page 193 and “Deleted functions (C++11)” on page 194.

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

- "Class members and friends (C++ only)" on page 255
- "Scope of class names (C++ only)" on page 250

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, function 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.

C++11 You can declare user-defined conversions as deleted functions. For more information, see “Deleted functions (C++11)” on page 194

Related information

- “Type conversions” on page 117

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 163
- “The static_cast operator (C++ only)” on page 156

Explicit conversion constructors (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 conversion 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 are 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 314

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);
```

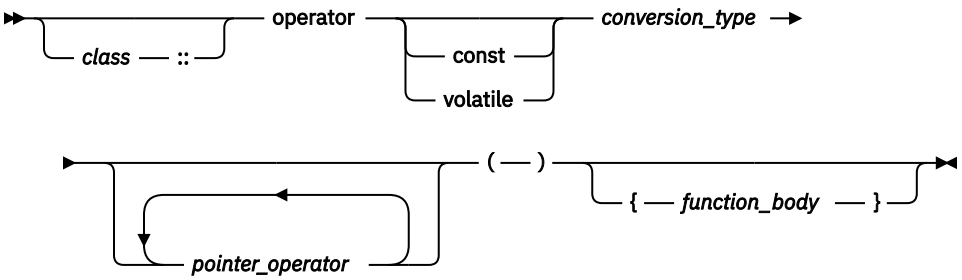
Related information

- “Conversion constructors (C++ only)” on page 314

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

- “Type conversions” on page 117
- “User-defined conversions (C++ only)” on page 313
- “Conversion constructors (C++ only)” on page 314
- “Type conversions” on page 117

Explicit conversion operators (C++11)

Note: IBM supports selected features of C++11, known as C++0x before its ratification. IBM will continue to develop and implement the features of this standard. The implementation of the language level is based on IBM's interpretation of the standard. Until IBM's implementation of all the C++11 features is complete, including the support of a new C++11 standard library, the implementation may change from release to release. IBM makes no attempt to maintain compatibility, in source, binary, or listings and other compiler interfaces, with earlier releases of IBM's implementation of the new C++11 features.

You can apply the `explicit` function specifier to the definition of a user-defined conversion function to inhibit unintended implicit conversions from being applied. Such conversion functions are called explicit conversion operators.

```
Explicit conversion operator syntax

>>-explicit--operator--conversion_type-------------------------->

>--+----------------------+--(--)--+----------+----------------->
   | .------------------. |        +-const----+
   | V                  | |        '-volatile-'
   '---pointer_operator-+-'

>--+---------------------+------------------------------------><
   '-{--function_body--}-'
```

The following example demonstrates both intended and unintended implicit conversions through a user-defined conversion function, which is not qualified with the `explicit` function specifier.

Example 1

```
#include <iostream>

template <class T> struct S {
  operator bool() const;   // conversion function
};

void func(S<int>& s) {

  // The compiler converts s to the bool type implicitly through
  // the conversion function. This conversion might be intended.
  if (s) { }
}

void bar(S<int>& p1, S<float>& p2) {
  // The compiler converts both p1 and p2 to the bool type implicitly
  // through the conversion function. This conversion might be unintended.
  std::cout << p1+p2 << std::endl;

  // The compiler converts both p1 and p2 to the bool type implicitly
  // through the conversion function and compares results.
  // This conversion might be unintended.
  if (p1==p2) { }
}
```

To inhibit unintended implicit conversions from being applied, you can define an explicit conversion operator by qualifying the conversion function in Example 1 with the `explicit` function specifier:

```
explicit operator bool() const;
```

If you compile the same code as Example 1 but with the explicit conversion operator, the compiler issues error messages for the following statements:

```
// Error: The call does not match any parameter list for "operator+".
std::cout << p1+p2 << std::endl;

// Error: The call does not match any parameter list for "operator==".
if(p1==p2)
```

If you intend to apply the conversion through the explicit conversion operator, you must call the explicit conversion operator explicitly as in the following statements, and then you can get the same results as Example 1.

```
std::cout << bool(p1)+bool(p2) << std::endl;

if(bool(p1)==bool(p2))
```

In contexts where a Boolean value is expected, such as when &&, ||, or the conditional operator is used, or when the condition expression of an `if` statement is evaluated, an explicit `bool` conversion operator can be implicitly invoked. So when you compile Example 1 with the previous explicit conversion operator, the compiler also converts `s` in the `func` function to the `bool` type through the explicit `bool` conversion operator implicitly. Example 2 also demonstrates this:

Example 2

```
struct T {
  explicit operator bool();   //explicit bool conversion operator
};

int main() {
  T t1;
  bool t2;

  // The compiler converts t1 to the bool type through
  // the explicit bool conversion operator implicitly.
  t1 && t2;

  return 0;
}
```

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&)`.

A program is ill-formed if it includes a class A whose copy constructor is implicitly defined C++11 or explicitly defaulted when one or more of the following conditions 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 C++11 or explicitly defaulted constructor of a class A if you initialize an object of type A or an object derived from class A.

An implicitly defined C++11 or explicitly defaulted 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.

C++11 **Note:** You can declare copy constructors as explicitly defaulted functions or deleted functions. For more information, see “Explicitly defaulted functions (C++11)” on page 193 and “Deleted functions (C++11)” on page 194.

Related information

- “Overview of constructors and destructors” 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 a copy assignment operator for a class A is implicitly defined C++11 or explicitly defaulted when 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.

C++11 **Note:** You can declare copy constructors as explicitly defaulted functions or deleted functions. For more information, see "Explicitly defaulted functions (C++11)" on page 193 and "Deleted functions (C++11)" on page 194.

Related information

- "Assignment operators" on page 141

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 342. 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 345.

IBM i For IBM i specific usage information, see "Using Templates in C++ Programs" in *ILE C/C++ Programmers Guide*.

Template declaration syntax

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 324.

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:

© Copyright IBM Corp. 1993, 2015

class Key<int> i;	class Key<char*> c;	class Key<mytype> m;

class Key
{
  int k;
  int * kptr;
  int length;
public:
  Key(int);
  // ...
};
	class Key
	{
	  char* k;
	  char** kptr;
	  int length;
	public:
	  Key(char*);
	  // ...
	};
		class Key
		{
		  mytype k;
		  mytype* kptr;
		  int length;
		public:
		  Key(mytype);
		  // ...
		};

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)" on page 324
- "Non-type template parameters (C++ only)" on page 324
- "Template template parameters (C++ only)" on page 325

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.

C++11 *Template parameter packs* can also be a kind of template parameter. For more information, see "Variadic templates (C++11)" on page 352.

Related information

- "Type qualifiers" on page 79
- "Lvalues and rvalues" on page 125

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 363

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
- C++11 `std::nullptr_t`

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)`.

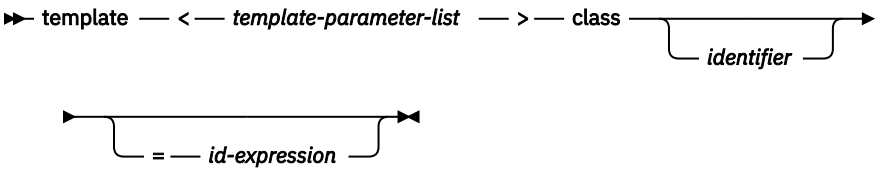
You may qualify a non-type template parameter with `const` or `volatile`.

You cannot declare a non-type template parameter as a floating point, class, or void type.

Non-type template parameters are not lvalues.

Template template parameters (C++ only)

Template template parameter declaration syntax



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 324

Default arguments for template parameters

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 324

Naming template parameters as friends (C++11)

Note: C++11 is a new version of the C++ programming language standard. IBM continues to develop and implement the features of the new standard. The implementation of the language level is based on IBM's interpretation of the standard. Until IBM's implementation of all the features of the C++11 standard is complete, including the support of a new C++ standard library, the implementation may change from release to release. IBM makes no attempt to maintain compatibility, in source, binary, or listings and other compiler interfaces, with earlier releases of IBM's implementation of the new features of the C++11 standard and therefore they should not be relied on as a stable programming interface.

In the C++11 standard, the extended friend declarations feature is introduced, with which you can declare template parameters as friends. This makes friend declarations inside templates easier to use.

If a friend declaration resolves to a template parameter, then you cannot use an elaborated-type-specifier in this friend declaration; otherwise, the compiler issues an error.

Related information

- “Friends (C++ only)” on page 269

Template arguments (C++ only)

There are three kinds of template arguments corresponding to the three types of template parameters:

- “Template type arguments (C++ only)” on page 327
- “Template non-type arguments (C++ only)” on page 327
- “Template template arguments (C++ only)” on page 328

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

- “Block/local scope” on page 12
- “No linkage” on page 18

- "Bit field members" on page 68
- "typedef definitions" on page 77

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()`.

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 128
- “References (C++ only)” on page 99
- “External linkage” on page 17
- “Static members (C++ only)” on page 263

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 350
- “Template instantiation (C++ only)” on page 342

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`).

C++11 *Template parameter packs* can also be a kind of parameter for class templates. For more information, see "Variadic templates (C++11)" on page 352.

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}
```

C++11 *Beginning of C++11 only.*

When the right angle bracket feature is enabled, the >> token is treated as two consecutive > tokens if both the following conditions are true:

- The >> token is in a context where one or more left angle brackets are active. A left angle bracket is active when it is not yet matched by a right angle bracket.
- The >> token is not nested within a delimited expression context.

If the first > token is in the context of a `template_parameter_list`, it is treated as the ending delimiter for the `template_parameter_list`. Otherwise, it is treated as the greater-than operator. The second > token terminates an enclosing `template_id` template_id construct or a different construct, such as the `const_cast`, `dynamic_cast`, `reinterpret_cast`, or `static_cast` operator. For example:

```
template<typename T> struct list {};
template<typename T>

struct vector
{
```

```
  operator T() const;
}

int main()
{
  // Valid, same as vector<vector<int> > v;
  vector<vector<int>> v;

  // Valid, treat the >> token as two consecutive > tokens.
  // The first > token is treated as the ending delimiter for the
  // template_parameter_list, and the second > token is treated as
  // the ending delimiter for the static_cast operator.
   const vector<int> vi = static_cast<vector<int>>(v);
}
```

A parenthesized expression is a delimited expression context. To use a bitwise shift operator inside template-argument-list, use parentheses to enclose the operator. For example:

```
template <int i> class X {};
template <class T> class Y {};

Y<X<(6>>1)>> y;          //Valid: 6>>1 uses the right shift operator
```

C++11 *End of C++11 only.*

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 247
- “Scope of class names (C++ only)” on page 250
- “Member functions (C++ only)” on page 256

Class template declarations and definitions

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
{ /* ... */ };
```

Related information

- “Class templates (C++ only)” on page 329

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 263

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)`.

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 269

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`.

C++11 *Beginning of C++11 only.*

Template parameter packs can be a kind of template parameter for function templates, and *function parameter packs* can be a kind of function parameter for function templates. For more information, see "Variadic templates (C++11)" on page 352.

You can use trailing return types for function templates, include those that have the following kinds of return types:

- Return types depending on the types of the function arguments
- Complicated return types

For more information, see "Trailing return type (C++11)" on page 207.

C++11 *End of C++11 only.*

Related information

- "Overload resolution (C++ only)" on page 242

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.

C++11 *Beginning of C++11 only.*

If a template type parameter of a function template is a cv-unqualified rvalue reference, but the argument in the function call is an lvalue, the corresponding lvalue reference is used instead of the rvalue reference. However, if the template type parameter is a cv-qualified rvalue reference, and the argument in the function call is an lvalue, the template instantiation fails. For example:

```
template <class T> double func1(T&&);
template <class T> double func2(const T&&);

int var;

// The compiler calls func1<int&>(int&)
double a = func1(var);

// The compiler calls func1<int>(int&&)
double b = func1(1);

// error
double c = func2(var);

// The compiler calls func2<int>(const int&&)
double d = func2(1);
```

In this example, the template type parameter of the function template `func1` is a cv-unqualified rvalue reference, and the template type parameter of the function template `func2` is a cv-qualified rvalue reference. In the initialization of variable a, the template argument `var` is an lvalue, so the lvalue reference type `int&` is used in the instantiation of the function template `func1`. In the initialization of variable b, the template argument 1 is an rvalue, so the rvalue reference type `int&&` remains in the template instantiation. In the initialization of c, the template type parameter T&& is cv-qualified, but `var` is an lvalue, so `var` cannot be bound to the rvalue reference T&&.

C++11 *End of C++11 only.*

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.

C++11 Template argument deduction also applies to the variadic templates feature. For more information, see “Variadic templates (C++11)” on page 352.

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'`.

Partial ordering of function templates

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 345
- “new expressions (C++ only)” on page 163

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 has two forms: explicit instantiation and implicit instantiation.

Related information

- "Template specialization (C++ only)" on page 345

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*.

C++0x *Beginning of C++0x only.*

The compiler does not need to generate the specialization for nonclass, noninline entities when an explicit instantiation declaration is present.

C++0x *End of C++0x only.*

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.

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 includes two forms: explicit instantiation declaration and explicit instantiation definition.

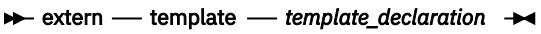
Note: IBM supports selected features of C++11, known as C++0x before its ratification. IBM will continue to develop and implement the features of the new standard. The implementation of the language level is based on IBM's interpretation of this standard. Until IBM's implementation of all the C++11 features are complete, including the support of a new C++ standard library, the implementation may change from release to release. IBM makes no attempt to maintain compatibility, in source, binary, or listings and other compiler interfaces, with earlier releases of IBM's implementation of the new C++11 features and therefore they should not be relied on as a stable programming interface.

C++11 *Beginning of C++11 only.*

Explicit instantiation declaration

The explicit instantiation declarations feature is introduced in the C++11 standard. With this feature, you can suppress the implicit instantiation of a template specialization or its members. The extern keyword is used to indicate explicit instantiation declaration. The usage of extern here is different from that of a storage class specifier.

Explicit instantiation declaration syntax



You can provide an explicit instantiation declaration for a template specialization if an explicit instantiation definition of the template exists in other translation units or later in the same file. If one translation unit contains the explicit instantiation definition, other translation units can use the specialization without having the specialization instantiated multiple times. The following example demonstrates this concept:

```
//sample1.h:
template<typename T, T val>
union A{
```

```
    T foo();
};
extern template union A<int, 55>;
template<class T, T val>
T A<T,val>::foo(void){
    return val;
}
//sampleA.C"
#include "sample1.h"
template union A<int,55>;
//sampleB.C:
#include "sample1.h"
int main(void){
    return A<int, 55>().foo();
}
```

sampleB.C uses the explicit instantiation definition of *A<int, 55>().foo()* in *sampleA.C*.

If an explicit instantiation declaration of a function or class is declared, but there is no corresponding explicit instantiation definition anywhere in the program, the compiler issues an error message. See the following example:

```
// sample2.C
template <typename T, T val>
struct A{
    virtual T foo();
    virtual T bar();
}
extern template int A<int,55>:foo();
template <class T, T val>
T A<T,val>::foo(void){
    return val;
}
template <class T, T val>
T A<T,val>::bar(void){
    return val;
}
int main(void){
    return A<int,55>().bar();
}
```

When you use explicit instantiation declaration, pay attention to the following restrictions:

- You can name a static class member in an explicit instantiation declaration, but you cannot name a static function because a static function cannot be accessed by name in other translation units.
- Explicit instantiation declarations have no effect on inline functions. An inline function is still implicitly instantiated even if an explicit instantiation declaration of the inline function is present, but no out-of-line copy of the inline function is generated in this case.
- The explicit instantiation declaration of a class is not equivalent to the explicit instantiation declaration of each of its members.

C++11 *End of C++11 only.*

Explicit instantiation definition

An explicit instantiation definition is an instantiation of a template specialization or a its members.

Explicit instantiation definition syntax

The following are examples of explicit instantiation definitions:

```
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 definition

namespace N {
   template<class T> void f(T&) { }
}
```

```
template void N::f<int>(int&);
// The explicit instantiation definition is in namespace N.

int* p = 0;
template<class T> T g(T = &p);
template char g(char);                  // explicit instantiation definition

template <class T> class X {
   private:
      T v(T arg) { return arg; };
};

template int X<int>::v(int);    // explicit instantiation definition

template<class T> T g(T val) { return val;}
template<class T> void Array<T>::mf() { }
```

An explicit instantiation definition of a template is in the same namespace where you define the template.

Access checking rules do not apply to the arguments in the explicit instantiation definitions. Template arguments in an explicit instantiation definition can be private types or objects. In this example, you can use the explicit instantiation definition `template int X<int>::v(int)` even though the member function is declared to be private.

The compiler does not use default arguments when you explicitly instantiate a template. In this example, you can use the explicit instantiation definition `template char g(char)` even though the default argument is an address of the type `int`.

Note: You cannot use the `inline` or C++11 `constexpr` specifier in an explicit instantiation of a function template or a member function of a class template.

C++11 *Beginning of C++11 only.*

Explicit instantiation and inline namespace definitions

Inline namespace definitions are namespace definitions with an initial `inline` keyword. Members of an inline namespace can be explicitly instantiated or specialized as if they were also members of the enclosing namespace. For more information, see “Inline namespace definitions (C++11)” on page 229

C++11 *End of C++11 only.*

Related information

- “Extensions for C++11 compatibility” on page 411

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 342

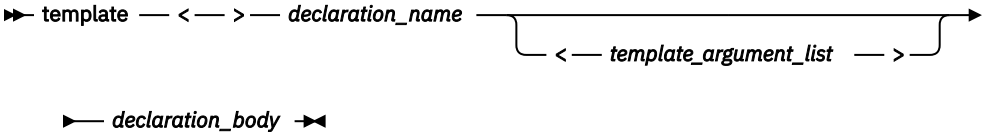
Explicit specialization (C++ only)

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



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()`.

C++0x *Beginning of C++0x only.*

Explicit specialization and inline namespace definitions

Inline namespace definitions are namespace definitions with an initial inline keyword. Members of an inline namespace can be explicitly instantiated or specialized as if they were also members of the enclosing namespace. For more information, see “Inline namespace definitions (C++11)” on page 229

C++0x *End of C++0x only.*

Related information

- “Function templates (C++ only)” on page 334
- “Class templates (C++ only)” on page 329
- “Member functions of class templates (C++ only)” on page 333
- “Static data members and templates (C++ only)” on page 332

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 334

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 332

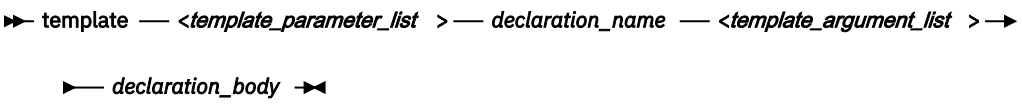
Partial specialization (C++ only)

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



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.

Related information

- “Template parameters (C++ only)” on page 324
- “Template arguments (C++ only)” on page 326

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.

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.

C++11 Partial specialization also applies to the variadic templates feature. For more information, see “Variadic templates (C++11)” on page 352.

.

Variadic templates (C++11)

Note: IBM supports selected features of C++11, known as C++0x before its ratification. IBM will continue to develop and implement the features of this standard. The implementation of the language level is based on IBM's interpretation of the standard. Until IBM's implementation of all the C++11 features is complete, including the support of a new C++11 standard library, the implementation may change from release to release. IBM makes no attempt to maintain compatibility, in source, binary, or listings and other compiler interfaces, with earlier releases of IBM's implementation of the new C++11 features.

Before C++11, templates had a fixed number of parameters that must be specified in the declaration of the templates. Templates could not directly express a class or function template that had a variable number of parameters. To partially alleviate this problem in the existing C++ programs, you could use overloaded function templates that had a different number of parameters or extra defaulted template parameters.

With the variadic templates feature, you can define class or function templates that have any number (including zero) of parameters. To achieve this goal, this feature introduces a kind of parameter called *parameter pack* to represent a list of zero or more parameters for templates.

The variadic template feature also introduces *pack expansion* to indicate that a parameter pack is expanded.

Two existing techniques, *template argument deduction* and *partial specialization*, can also apply to templates that have parameter packs in their parameter lists.

Related information

- “The sizeof operator” on page 138
- “Template parameters (C++ only)” on page 324
- “Template arguments (C++ only)” on page 326
- “Class templates (C++ only)” on page 329
- “Function templates (C++ only)” on page 334
- “Template argument deduction (C++ only)” on page 335
- “Partial specialization (C++ only)” on page 350
- “Extensions for C++11 compatibility” on page 411

Parameter packs (C++11)

A *parameter pack* can be a type of parameter for templates. Unlike previous parameters, which can only bind to a single argument, a parameter pack can pack multiple parameters into a single parameter by placing an ellipsis to the left of the parameter name.

In the template definition, a parameter pack is treated as a single parameter. In the template instantiation, a parameter pack is expanded and the correct number of the parameters are created.

According to the context where a parameter pack is used, the parameter pack can be either a *template parameter pack* or a *function parameter pack*.

Template parameter packs

A *template parameter pack* is a template parameter that represents any number (including zero) of template parameters. Syntactically, a template parameter pack is a template parameter specified with an ellipsis. Consider the following example.

```
template<class...A> struct container{};
template<class...B> void func();
```

In this example, A and B are template parameter packs.

According to the type of the parameters contained in a template parameter pack, there are three kinds of template parameter packs:

- Type parameter packs
- Non-type parameter packs
- Template template parameter packs

A type parameter pack represents zero or more type template parameters. Similarly, a non-type parameter pack represents zero or more non-type template parameters.

Note: Template template parameter packs are not supported in 7.3 ILE C++ Compiler.

The following example shows a type parameter pack:

```
template<class...T> class X{};

X<> a;                    // the parameter list is empty
X<int> b;                 // the parameter list has one item
X<int, char, float> c;    // the parameter list has three items
```

In this example, the type parameter pack T is expanded into a list of zero or more type template parameters.

The following example shows a non-type parameter pack:

```
template<bool...A> class X{};

X<> a;                    // the parameter list is empty
X<true> b;                // the parameter list has one item
X<true, false, true> c;   // the parameter list has three items
```

In this example, the non-type parameter pack A is expanded into a list of zero or more non-type template parameters.

In a context where template arguments can be deduced; for example, function templates and class template partial specializations, a template parameter pack does not need to be the last template parameter of a template. In this case, you can declare more than one template parameter pack in the template parameter list. However, if template arguments cannot be deduced, you can declare at most one template parameter pack in the template parameter list, and the template parameter pack must be the last template parameter. Consider the following example:

```
// error
template<class...A, class...B>struct container1{};

// error
template<class...A,class B>struct container2{};
```

In this example, the compiler issues two error messages. One error message is for class template `container1` because `container1` has two template parameter packs A and B that cannot be deduced. The other error message is for class template `container2` because template parameter pack A is not the last template parameter of `container2`, and A cannot be deduced.

Default arguments cannot be used for a template parameter pack. Consider the following example:

```
template<typename...T=int> struct foo1{};
```

In this example, the compiler issues an error message because the template parameter pack T is given a default argument `int`.

Function parameter packs

A *function parameter pack* is a function parameter that represents zero or more function parameters. Syntactically, a function parameter pack is a function parameter specified with an ellipsis.

In the definition of a function template, a function parameter pack uses a template parameter pack in the function parameters. The template parameter pack is expanded by the function parameter pack. Consider the following example:

```
template<class...A> void func(A...args)
```

In this example, A is a template parameter pack, and `args` is a function parameter pack. You can call the function with any number (including zero) of arguments:

```
func();                 // void func();
func(1);                // void func(int);
func(1,2,3,4,5);        // void func(int,int,int,int,int);
func(1,'x', aWidget);   // void func(int,char,widget);
```

A *function parameter pack* is a *trailing function parameter pack* if it is the last function parameter of a function template. Otherwise, it is a *non-trailing function parameter pack* . A function template can have trailing and non-trailing function parameter packs. A non-trailing function parameter pack can be deduced only from the explicitly specified arguments when the function template is called. If the function template is called without explicit arguments, the non-trailing function parameter pack must be empty, as shown in the following example:

```
#include <cassert>

template<class...A, class...B> void func(A...arg1,int sz1, int sz2, B...arg2)
{
   assert( sizeof...(arg1) == sz1);
   assert( sizeof...(arg2) == sz2);
}

int main(void)
{
   //A:(int, int, int), B:(int, int, int, int, int)
   func<int,int,int>(1,2,3,3,5,1,2,3,4,5);

   //A: empty, B:(int, int, int, int, int)
   func(0,5,1,2,3,4,5);
   return 0;
}
```

In this example, function template `func` has two function parameter packs `arg1` and `arg2`. `arg1` is a non-trailing function parameter pack, and `arg2` is a trailing function parameter pack. When `func` is called with three explicitly specified arguments as `func<int,int,int>(1,2,3,3,5,1,2,3,4,5)`, both `arg1` and `arg2` are deduced successfully. When `func` is called without explicitly specified arguments as `func(0,5,1,2,3,4,5)`, `arg2` is deduced successfully and `arg1` is empty. In this example, the template parameter packs of function template `func` can be deduced, so `func` can have more than one template parameter pack.

Pack expansion (C++11)

A *pack expansion* is an expression that contains one or more parameter packs followed by an ellipsis to indicate that the parameter packs are expanded. Consider the following example:

```
template<class...T> void func(T...a){};
template<class...U> void func1(U...b){
    func(b...);
}
```

In this example, `T...` and `U...` are the corresponding pack expansions of the template parameter packs T and U, and `b...` is the pack expansion of the function parameter pack b.

A pack expansion can be used in the following contexts:

- Expression list
- Initializer list
- Base specifier list

- Member initializer list
- Template argument list
- Exception specification list

Expression list

Example:

```
#include <cstdio>
#include <cassert>

template<class...A> void func1(A...arg){
    assert(false);
}

void func1(int a1, int a2, int a3, int a4, int a5, int a6){
    printf("call with(%d,%d,%d,%d,%d,%d)\n",a1,a2,a3,a4,a5,a6);
}

template<class...A> int func(A...args){
    int size = sizeof...(A);
    switch(size){
        case 0: func1(99,99,99,99,99,99);
        break;
        case 1: func1(99,99,args...,99,99,99);
        break;
        case 2: func1(99,99,args...,99,99);
        break;
        case 3: func1(args...,99,99,99);
        break;
        case 4: func1(99,args...,99);
        break;
        case 5: func1(99,args...);
        break;
        case 6: func1(args...);
        break;
        default:
        func1(0,0,0,0,0,0);
    }
    return size;
}

int main(void){
    func();
    func(1);
    func(1,2);
    func(1,2,3);
    func(1,2,3,4);
    func(1,2,3,4,5);
    func(1,2,3,4,5,6);
    func(1,2,3,4,5,6,7);
    return 0;
}
```

The output of this example:

```
call with (99,99,99,99,99,99)
call with (99,99,1,99,99,99)
call with (99,99,1,2,99,99)
call with (1,2,3,99,99,99)
call with (99,1,2,3,4,99)
call with (99,1,2,3,4,5)
call with (1,2,3,4,5,6)
call with (0,0,0,0,0,0)
```

In this example, the switch statement shows the different positions of the pack expansion `args...` within the expression lists of the function `func1`. The output shows each call of the function `func1` to indicate the expansion.

Initializer list

Example:

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

```
void printarray(int arg[], int length){
    for(int n=0; n<length; n++){
        printf("%d ",arg[n]);
    }
    printf("\n");
}

template<class...A> void func(A...args){
    const int size = sizeof...(args) +5;
    printf("size %d\n", size);
    int res[sizeof...(args)+5]={99,98,args...,97,96,95};
    printarray(res,size);
}

int main(void)
{
    func();
    func(1);
    func(1,2);
    func(1,2,3);
    func(1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20);
    return 0;
}
```

The output of this example:

```
size 5
99 98 97 96 95
size 6
99 98 1 97 96 95
size 7
99 98 1 2 97 96 95
size 8
99 98 1 2 3 97 96 95
size 25
99 98 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 97 96 95
```

In this example, the pack expansion `args...` is in the initializer list of the array `res`.

Base specifier list

Example:

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

struct a1{};
struct a2{};
struct a3{};
struct a4{};

template<class X> struct baseC{
    baseC() {printf("baseC primary ctor\n");}
};
template<> struct baseC<a1>{
    baseC() {printf("baseC a1 ctor\n");}
};
template<> struct baseC<a2>{
    baseC() {printf("baseC a2 ctor\n");}
};
template<> struct baseC<a3>{
    baseC() {printf("baseC a3 ctor\n");}
};
template<> struct baseC<a4>{
    baseC() {printf("baseC a4 ctor\n");}
};

template<class...A> struct container : public baseC<A>...{
    container(){
        printf("container ctor\n");
    }
};

int main(void){
    container<a1,a2,a3,a4> test;
    return 0;
}
```

The output of this example:

```
baseC a1 ctor
baseC a2 ctor
baseC a3 ctor
baseC a4 ctor
container ctor
```

In this example, the pack expansion `baseC<A>...` is in the base specifier list of the class template `container`. The pack expansion is expanded into four base classes`baseC<a1>`, `baseC<a2>`, `baseC<a3>`, and `baseC<a4>`. The output shows that all the four base class templates are initialized before the instantiation of the class template `container`

Base specifier list

Example:

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

struct a1{};
struct a2{};
struct a3{};
struct a4{};

template<class X> struct baseC{
    baseC(int a) {printf("baseC primary ctor: %d\n", a);}
};
template<> struct baseC<a1>{
    baseC(int a) {printf("baseC a1 ctor: %d\n", a);}
};
template<> struct baseC<a2>{
    baseC(int a) {printf("baseC a2 ctor: %d\n", a);}
};
template<> struct baseC<a3>{
    baseC(int a) {printf("baseC a3 ctor: %d\n", a);}
};
template<> struct baseC<a4>{
    baseC(int a) {printf("baseC a4 ctor: %d\n", a);}
};

template<class...A> struct container : public baseC<A>...{
    container(): baseC<A>(12)...{
        printf("container ctor\n");
    }
};

int main(void){
    container<a1,a2,a3,a4> test;
    return 0;
}
```

The output of this example:

```
baseC a1 ctor:12
baseC a2 ctor:12
baseC a3 ctor:12
baseC a4 ctor:12
container ctor
```

In this example, the pack expansion `baseC<A>(12)...` is in the member initializer list of the class template `container`. The constructor initializer list is expanded to include the call for each base class `baseC<a1>(12)`, `baseC<a2>(12)`, `baseC<a3>(12)`, and `baseC<a4>(12)`.

Template argument list

Example:

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

template<int val> struct value{
    operator int(){return val;}
};
```

```
template <typename...I> struct container{
    container(){
        int array[sizeof...(I)]={I()...};
        printf("container<");
        for(int count = 0; count<sizeof...(I); count++){
            if(count>0){
                printf(",");
            }
            printf("%d", array[count]);
        }
        printf(">\n");
    }
};

template<class A, class B, class...C> void func(A arg1, B arg2, C...arg3){
    container<A,B,C...> t1;  // container<99,98,3,4,5,6>
    container<C...,A,B> t2;  // container<3,4,5,6,99,98>
    container<A,C...,B> t3;  // container<99,3,4,5,6,98>
}

int main(void){
    value<99> v99;
    value<98> v98;
    value<3> v3;
    value<4> v4;
    value<5> v5;
    value<6> v6;
    func(v99,v98,v3,v4,v5,v6);
    return 0;
}
```

The output of this example:

```
container<99,98,3,4,5,6>
container<3,4,5,6,99,98>
container<99,3,4,5,6,98>
```

In this example, the pack expansion `C...` is expanded in the context of template argument list for the class template `container`.

Exception specification list

Example:

```
struct a1{};
struct a2{};
struct a3{};
struct a4{};
struct a5{};
struct stuff{};

template<class...X> void func(int arg) throw(X...){
    a1 t1;
    a2 t2;
    a3 t3;
    a4 t4;
    a5 t5;
    stuff st;

    switch(arg){
        case 1:
            throw t1;
            break;
        case 2:
            throw t2;
            break;
        case 3:
            throw t3;
            break;
        case 4:
            throw t4;
            break;
        case 5:
            throw t5;
            break;
        default:
            throw st;
```

```
            break;
    }
}

int main(void){
    try{
        // if the throw specification is correctly expanded, none of
        // these calls should trigger an exception that is not expected
        func<a1,a2,a3,a4,a5,stuff>(1);
        func<a1,a2,a3,a4,a5,stuff>(2);
        func<a1,a2,a3,a4,a5,stuff>(3);
        func<a1,a2,a3,a4,a5,stuff>(4);
        func<a1,a2,a3,a4,a5,stuff>(5);
        func<a1,a2,a3,a4,a5,stuff>(99);
    }
    catch(...){
        return 0;
    }
    return 1;
}
```

In this example, the pack expansion `X...` is expanded in the context of exception specification list for the function template `func`.

If a parameter pack is declared, it must be expanded by a pack expansion. An appearance of a name of a parameter pack that is not expanded is incorrect. Consider the following example:

```
template<class...A> struct container;
template<class...B> struct container<B>{}
```

In this example, the compiler issues an error message because the template parameter pack B is not expanded.

Pack expansion cannot match a parameter that is not a parameter pack. Consider the following example:

```
template<class X> struct container{};

template<class A, class...B>
// Error, parameter A is not a parameter pack
void func1(container<A>...args){};

template<class A, class...B>
// Error, 1 is not a parameter pack
void func2(1...){};
```

If more than one parameter pack is referenced in a pack expansion, each expansion must have the same number of arguments expanded from these parameter packs. Consider the following example:

```
struct a1{}; struct a2{}; struct a3{}; struct a4{}; struct a5{};

template<class...X> struct baseC{};
template<class...A1> struct container{};
template<class...A, class...B, class...C>
struct container<baseC<A,B,C...>...>:public baseC<A,B...,C>{};

int main(void){
    container<baseC<a1,a4,a5,a5,a5>, baseC<a2,a3,a5,a5,a5>,
              baseC<a3,a2,a5,a5,a5>,baseC<a4,a1,a5,a5,a5> > test;
    return 0;
}
```

In this example, the template parameter packs A, B, and C are referenced in the same pack expansion `baseC<A,B,C...>...`. The compiler issues an error message to indicate that the lengths of these three template parameter packs are mismatched when expanding them during the template instantiation of the class template `container`.

Partial specialization (C++11)

Partial specialization is a fundamental part of the variadic templates feature. A basic partial specialization can be used to access the individual arguments of a parameter pack. The following example shows how to use partial specialization for variadic templates:

```
// primary template
template<class...A> struct container;

// partial specialization
template<class B, class...C> struct container<B,C...>{};
```

When the class template `container` is instantiated with a list of arguments, the partial specialization is matched in all cases where there are one or more arguments. In that case, the template parameter `B` holds the first parameter, and the pack expansion `C...` contains the rest of the argument list. In the case of an empty list, the partial specialization is not matched, so the instantiation matches the primary template.

A pack expansion must be the last argument in the argument list for a partial specialization. Consider the following example:

```
template<class...A> struct container;

// partial specialization
template<class B, class...C> struct container<C...,B>{};
```

In this example, the compiler issues an error message because the pack expansion `C...` is not the last argument in the argument list for the partial specialization. A partial specialization can have more than one template parameter pack in its parameter list. Consider the following example:

```
template<typename T1, typename T2> struct foo{};
template<typename...T> struct bar{};

// partial specialization
template<typename...T1,typename...T2> struct bar<foo<T1,T2>...>{};
```

In this example, the partial specialization has two template parameter packs T1 and T2 in its parameter list.

To access the arguments of a parameter pack, you can use partial specialization to access one member of the parameter pack in the first step, and recursively instantiate the remainder of the argument list to get all the elements, as shown in the following example:

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

struct a1{}; struct a2{}; struct a3{}; struct a4{}; struct a5{};
struct a6{}; struct a7{}; struct a8{}; struct a9{}; struct a10{};

template<typename X1, typename X2> struct foo{foo();};
template<typename X3, typename X4> foo<X3,X4>::foo(){cout<<"primary foo"<<endl;};
template<> struct foo<a1,a2>{foo(){cout<<"ctor foo<a1,a2>"<<endl;}};
template<> struct foo<a3,a4>{foo(){cout<<"ctor foo<a3,a4>"<<endl;}};
template<> struct foo<a5,a6>{foo(){cout<<"ctor foo<a5,a6>"<<endl;}};
template<> struct foo<a7,a8>{foo(){cout<<"ctor foo<a7,a8>"<<endl;}};
template<> struct foo<a9,a10>{foo(){cout<<"ctor foo<a9,a10>"<<endl;}};

template<typename...T>struct bar{bar{}{cout<<"bar primary"<<endl;}};

template<typename A, typename B, typename...T1, typename...T2>
struct bar<foo<A,B>,foo<T1,T2>...>{
    foo<A,B> data;
    bar<foo<T1,T2>...>data1;
};

template<> struct bar<foo<a9,a10> > {bar(){cout<<"ctor bar<foo<a9,a10>>"<<endl;}};

int main(){
    bar<foo<a1,a2>,foo<a3,a4>,foo<a5,a6>,foo<a7,a8>,foo<a9,a10> > t2;
```

```
    return 0;
}
```

The output of the example:

```
ctor foo<a1,a2>
ctor foo<a3,a4>
ctor foo<a5,a6>
ctor foo<a7,a8>
ctor bar<foo<a9,a10>
```

Template argument deduction (C++11)

Parameter packs can be deduced by template argument deduction in the same way as other normal template parameters. The following example shows how template argument deduction expands packs from a function call:

```
template<class...A> void func(A...args){}

int main(void){
    func(1,2,3,4,5,6);
    return 0;
}
```

In this example, the function argument list is `(1,2,3,4,5,6)`. Each function argument is deduced to the type `int`, so the template parameter pack A is deduced to the following list of types: `(int,int,int,int,int,int)`. With all the expansions, the function template `func` is instantiated as `void func(int,int,int,int,int,int)`, which is the template function with the expanded function parameter pack.

In this example, if you change the function call statement `func(1,2,3,4,5,6)` to `func()`, template argument deduction deduces that the template parameter pack Ais empty:

```
template<class...A> void func(A...args){}

int main(void){
    func();
    return 0;
}
```

Template argument deduction can expand packs from a template instantiation, as shown in the following example:

```
#include <cstdio>

template<int...A> struct container{
    void display(){printf("YIKES\n");}
};

template<int B, int...C> struct container<B,C...>{
    void display(){
        printf("spec %d\n",B);
        container<C...>test;
        test.display();
    }
 };

template<int C> struct container<C>{
    void display(){printf("spec %d\n",C);}
};

int main(void)
{
   printf("start\n\n");
   container<1,2,3,4,5,6,7,8,9,10> test;
   test.display();
   return 0;
}
```

The output of this example:

```
start

spec 1
spec 2
spec 3
spec 4
spec 5
spec 6
spec 7
spec 8
spec 9
spec 10
```

In this example, the partial specialization of the class template `container` is `template<int B, int...C> struct container<B,C...>`. The partial specialization is matched when the class template is instantiated to `container<1,2,3,4,5,6,7,8,9,10>`. Template argument deduction deduces the template parameter pack `C` and the parameter `B` from the argument list of the partial specialization. Template argument deduction then deduces the parameter `B` to be 1, the pack expansion `C...` to a list: `(2,3,4,5,6,7,8,9,10)`, and the template parameter pack `C` to the following list of types:`(int,int,int,int,int,int,int,int,int)`.

If you change the statement `container<1,2,3,4,5,6,7,8,9,10> test` to `container<1> test`, template argument deduction deduces that the template parameter pack `C` is empty.

Template argument deduction can expand packs after the explicit template arguments are found. Consider the following example:

```
#include <cassert>

template<class...A> int func(A...arg){
    return sizeof...(arg);
}

int main(void){
    assert(func<int>(1,2,3,4,5) == 5);
    return 0;
}
```

In this example, the template parameter pack `A` is deduced to a list of types: `(int,int,int,int,int)` using the explicit argument list and the arguments in the function call.

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.

C++11 *Beginning of C++11 only.*

The decltype feature can interact with template dependent names. If the operand *expression* in the decltype(*expression*) type specifier is dependent on template parameters, the compiler cannot determine the validity of *expression* before the template instantiation, as shown in the following example:

```
template <class T, class U> int h(T t, U u, decltype(t+u) v);
```

In this example, the compiler issues an error message if the operand `t+u` is invalid after the instantiation of the function template h.

For more information, see “The decltype(expression) type specifier (C++11)” on page 59

C++11 *End of C++11 only.*

Related information

- “Template instantiation (C++ only)” on page 342

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 324

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.

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.

IBM i For IBM i specific usage information, see "Handling Exceptions in a Program" in *ILE C/C++ Programmers Guide*.

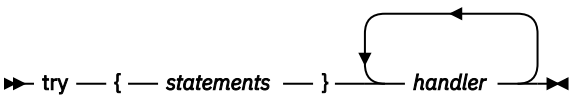
The exception handling mechanism is made up of the following elements:

- `try` blocks
- `catch` blocks
- `throw` expressions
- “Exception specifications (C++ only)” on page 378

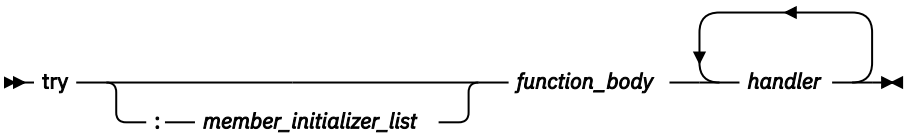
try blocks (C++ only)

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
```

© Copyright IBM Corp. 1993, 2015

```
      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" on page 306

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.
```

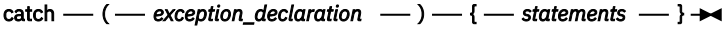
In the above example, if `spec_err` is thrown within the inner try block (in this case, from `func2()`), the exception is caught by the inner catch block, and, assuming this catch block does not transfer control, `func3()` is called. If `spec_err` is thrown after the inner try block (for instance, by `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



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 79
- “Member access (C++ only)” on page 267

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 runtime 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 runtime 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 cow; }
   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
```

C++0x *Beginning of C++0x only.*

If the delegating process exists and an exception occurs in the body of a target constructor, the exception can be caught by an appropriate handler in the try block of the delegating constructor. The following example demonstrates this:

```
#include <cstdio>
using std::printf;
int global_argc;
struct A{
    int _x;
    A();
    A(int);
};
A::A(int x):_x((printf("In A::A(int) initializer for A::_x.\n"),x)){
    printf("In A::A(int) constructor body.\n");
    if(global_argc % 2 !=0){
        printf("Will throw.\n");
        throw 0;
    }
    printf("Will not throw.\n");
}
A::A() try:A((printf("In A::A() initializer for delegating to A::A(int).\n"),42)){
    printf("In A::A() function-try-block body.\n");
}
catch(...){
    printf("In catch(...) handler for A::A() function-try-block.\n");
}
int main(int argc, char **argv){
    printf("In main().\n");
    global_argc = argc;
    try{
        A a;
        printf("Back in main().\n");
    }
    catch(...){
        printf("In catch(...) handler for try-block in main().\n");
    }
    return 0;
}
```

The example can produce different output depending on how many arguments are passed on the invocation of the resulting program. With an even number of arguments, the exception is thrown. The output is:

```
In main().
In A::A() initializer for delegating to A:A(int).
In A::A(int) initializer for A::_x.
In A::A(int) constructor body.
Will throw.
In catch(...) handler for A::A() function-try-block.
In catch(...) handler for try-block in main().
```

With an odd number of arguments, there is no exception thrown. The output is:

```
In main().
In A::A() initializer for delegating to A::A(int).
In A::A(int) initializer for A::_x.
In A::A(int) constructor body.
Will not throw.
```

```
In A::A() function-try-block body.
Back in main().
```

For more information, see “Delegating constructors (C++11)” on page 302

C++0x *End of C++0x only.*

Related information

- “The main() function” on page 212
- “The static storage class specifier” on page 49
- “Namespaces (C++ only)” on page 223
- “Destructors (C++ only)” on page 311

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

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 120
- “Type qualifiers” on page 79
- “References (C++ only)” on page 99

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 runtime cannot find a matching handler in the current scope, the runtime 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 runtime finds a matching handler in a dynamically surrounding try block: the try block in the `main()` function.

If the runtime cannot find a matching handler in the program, it calls the `terminate()` function.

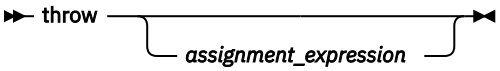
Related information

- “try blocks (C++ only)” on page 367

throw expressions (C++ only)

You use a *throw expression* to indicate that your program has encountered an exception.

throw expression syntax



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.

IBM i The IBM i has a restriction that the object thrown can be no larger than 16MB in size.

Related information

- "Incomplete types" on page 44

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++ runtime 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++ runtime 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++ runtime 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()`.)

C++0x *Beginning of C++0x only.*

When the delegating constructors feature is enabled, if an exception is thrown in the body of a delegating constructor, the destructors of the objects constructed through target constructor will be invoked automatically. The destructors must be called in such a way that it calls the destructors of subobjects as appropriate. In particular, it should call the destructors for virtual base classes if the virtual base classes are created through the target constructor.

If an exception is thrown in the body of a delegating constructor, the destructor is invoked for the object created by the target constructor. If an exception escapes from a non-delegating constructor, the unwinding mechanism will call the destructors for the completely constructed subobjects. The following example demonstrates this:

```
#include<iostream>
class D{
public:
   D():D('a') { printf("D:D().\n");}
   D(char) try: D(55){
   printf("D::D(char). Throws.\n");
   throw 0;
   }
   catch(...){
   printf("D::D(char).Catch block.\n");
   }
   D(int i_):i(i_) {printf("D::D(int).\n");}
   ~D() {printf("D::~D().\n");}
private:
   int i;
};
int main(void){
    D d;
     return 0;
}
```

The output of the example is:

```
D::D(int).
D::D(char).Throws.
D::~D().
D::D(char).Catch block.
```

In this example, an exception occurs in the delegating constructor `D:D(char)`, so destructor `D:~D()` is invoked for object d.

For more information, see “Delegating constructors (C++11)” on page 302

C++0x *End of C++0x only.*

Exception specifications (C++ only)

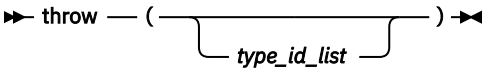
C++ provides a mechanism to ensure that a given function is limited to throwing only a specified list of exceptions. An exception specification at the beginning of any function acts as a guarantee to the function's caller that the function will throw only the exceptions contained in the exception specification.

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



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, abstract class type, C++11 rvalue reference 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 44
- "Function declarations and definitions" on page 191
- "Pointers to functions" on page 219
- "Special member functions (C++ only)" on page 299

Special exception handling functions

Not all thrown errors can be caught and successfully dealt with by a catch block. In some situations, the best way to handle an exception is to terminate the program. Two special library functions are implemented in C++ to process exceptions not properly handled by catch blocks or exceptions thrown outside of a valid try block. These functions are:

- "The unexpected() function (C++ only)" on page 380
- "The terminate() function (C++ only)" on page 381

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++ runtime 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++ runtime 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++ runtime 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” on page 380
- “The set_unexpected() and set_terminate() functions” on page 382

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;

void f() {
  printf("Function f()\n");
  throw "Exception thrown from f()";
}

void g() { printf("Function g()\n"); }
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" on page 382

The set_unexpected() and set_terminate() functions

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 no 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 runtime, 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.

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" on page 385, which replace tokens in the current file with specified replacement tokens
- "File inclusion directives" on page 392, which imbed files within the current file
- "Conditional compilation directives" on page 395, which conditionally compile sections of the current file
- "Message generation directives" on page 399, which control the generation of diagnostic messages
- C++ "Assertion directives" on page 401, which specify attributes of the system the program is to run on
- "The null directive (#)" on page 402, which performs no action
- "Pragma directives" on page 402, 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" on page 385, which defines a macro
- "The #undef directive" on page 390, which removes a macro definition

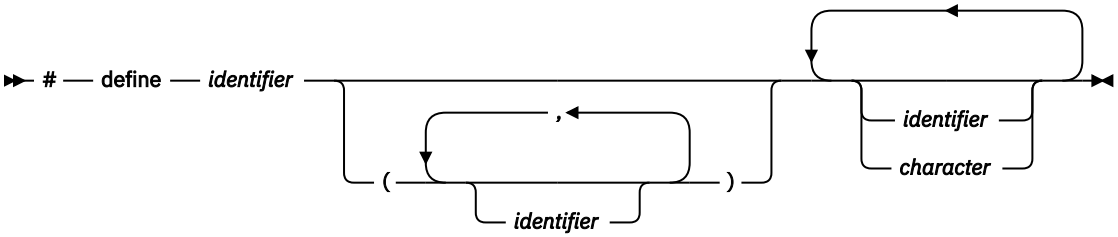
Standard predefined macros and macros that are predefined for IBM i are described in "Predefined macros" in the *ILE C/C++ Compiler Reference*.

The #define directive

A *preprocessor define directive* directs the preprocessor to replace all subsequent occurrences of a macro with specified replacement tokens.

© Copyright IBM Corp. 1993, 2015

#define directive syntax



The #define directive can contain:

- "Object-like macros" on page 386
- "Function-like macros" on page 387

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.
- C++ The compiler does not type-check a macro, including macro arguments.

Related information

- "The const type qualifier" on page 82

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
```

C++0x *Beginning of C++0x only.*

In C++0x, the diagnostic for object-like macros in the C99 preprocessor is adopted to provide a common preprocessor interface for C and C++ compilers. The C++0x compiler issues a warning message if there are no white spaces between an object-like macro name and its replacement list in a macro definition. For more information, see "C99 preprocessor features adopted in C++11 (C++11)" on page 403.

C++0x *End of C++0x only.*

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. ILE C++ supports function-like macros with a variable number of arguments, as a language extension for compatibility with C C++0x and as part of C++0x.

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.

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. */
```

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:

```
/**
 ** This example illustrates #define directives.
 **/

#include <stdio.h>

#define SQR(s)  ((s) * (s))
#define PRNT(a,b) \
  printf("value 1 = %d\n", a); \
  printf("value 2 = %d\n", b) ;

int main(void)
{
  int x = 2;
  int y = 3;

     PRNT(SQR(x),y);

  return(0);
}
```

After being interpreted by the preprocessor, this program is replaced by code equivalent to the following:

```
#include <stdio.h>

int main(void)
{
  int x = 2;
  int y = 3;

     printf("value 1 = %d\n", ( (x) * (x) ) );
     printf("value 2 = %d\n", y);

  return(0);
}
```

This program produces the following output:

```
value 1 = 4
value 2 = 3
```

Related information

- “Operator precedence and associativity” on page 167
- “Parenthesized expressions ( )” on page 129

Variadic macro extensions

Variadic macro extensions refer to two extensions to C99 and Standard C++ related to macros with variable number of arguments. One extension is a mechanism for renaming the variable argument identifier from __VA_ARGS__ to a user-defined identifier. The other extension provides a way to remove the dangling comma in a variadic macro when no variable arguments are specified.

The following examples demonstrate the use of an identifier in place of __VA_ARGS__. The first definition of the macro debug exemplifies the usual usage of __VA_ARGS__. The second definition shows the use of the identifier args in place of __VA_ARGS__.

```
#define debug1(format, …)  printf(format, ## __VA_ARGS__)
#define debug2(format, args …)  printf(format, ## args)
```

Invocation	Result of macro expansion
debug1("Hello %s\n","World");	printf("Hello %s\n","World");
debug2("Hello %s\n","World");	printf("Hello %s\n","World");

The preprocessor removes the trailing comma if the variable arguments to a function macro are omitted or empty and the comma followed by ## precedes the variable argument identifier in the function macro definition.

C++0x *Beginning of C++0x only.*

In C++0x, the variadic macros feature and changes concerning empty macro arguments are adopted from the C99 preprocessor to provide a common preprocessor interface for C and C++ compilers. Variadic macros and empty macro arguments are supported in C++0x. For more information, see "C99 preprocessor features adopted in C++11 (C++11)" on page 403.

C++0x *End of C++0x only.*

The #undef directive

A *preprocessor undef directive* causes the preprocessor to end the scope of a preprocessor definition.

#undef directive syntax

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"

Invocation	Result of macro expansion
`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 402

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 385

File inclusion directives

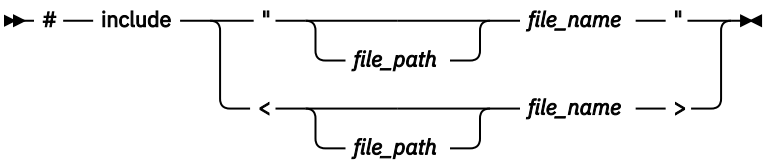
File inclusion directives consist of:

- "The #include directive" on page 392, which inserts text from another source file
- IBM i "The #include_next directive" on page 395, which causes the compiler to omit the directory of the including file from the search path when searching for include files

The #include directive

A *preprocessor include directive* causes the preprocessor to replace the directive with the contents of the specified file.

#include directive syntax



Using the #include Directive when Compiling Source in a Data Management File

The following table indicates the search path the compiler takes for source physical files. See the default file names and search paths below.

Table 28. Search Path for Source Physical Files

Filename	Member	File	Library
mbr	*mbr*	default file	default search
file/mbr[1]	*mbr*	*file*	default search

Table 28. Search Path for Source Physical Files (continued)			
Filename	**Member**	**File**	**Library**
mbr.file	*mbr*	*file*	default search
lib/file/mbr	*mbr*	*file*	*lib*
lib/file(mbr)	*mbr*	*file*	*lib*
Note: [1] If the include file format <file/mbr.h> is used, the compiler searches for *mbr* in the file in the library list first. If *mbr* is not found, then the compiler searches for *mbr.h* in the same file in the library list. Only "h" or "H" are allowed as member name extensions.			

If library and file are not specified, the preprocessor uses a specific search path depending on which delimiter surrounds the *filename*. The < > delimeter specifies the name as a system include file. The " " delimiter specifies the name as a user include file.

The following describes the search paths for the #include directive used by the compiler.

- Default file names when the library and file are not named (member name only):

  Include type
  Default File Name

  <>
  QCSRC for the C compiler, STD for the C++ compiler

  ” "
  The source file of the root source member, where root source member is the library, file, and member determined by the SRCFILE option of the Create Module or Create Bound Program commands.

- Default search paths when the filename is not library qualified

  Include type
  Search path

  <>
  Searches the current library list (*LIBL)

  " "
  Checks the library containing the root source member; if not there, the compiler searches the user portion of the library list, using either the filename specified or the file name of the root source member (if no filename is specified); if not found, the compiler searches the library list (*LIBL) using the specified filename.

- Search paths when the filename is library qualified (lib/file/mbr)

  Include Type
  Search Path

  <>
  Searches for lib/file/mbr only

  " "
  Searches for the member in the library and file named. If not found, searches the user portion of the library list, using the file and member names specified.

User includes are treated the same as system includes when the *SYSINCPATH option has been specified with the Create Module or Create Bound Program commands.

The preprocessor resolves macros on a #include directive. After macro replacement, the resulting token sequence must consist of a file name enclosed in either double quotation marks or the characters < and >. For example:

```
#define MONTH <july.h>
#include MONTH
```

Usage

If there are a number of declarations used by several files, you can place all these definitions in one file and `#include` that file in each file that uses the definitions. For example, the following file `defs.h` contains several definitions and an inclusion of an additional file of declarations:

```
/* defs.h */
#define TRUE 1
#define FALSE 0
#define BUFFERSIZE 512
#define MAX_ROW 66
#define MAX_COLUMN 80
int hour;
int min;
int sec;
#include "mydefs.h"
```

You can imbed the definitions that appear in defs.h with the following directive:

```
#include "defs.h"
```

One of the ways you can combine the use of preprocessor directives is demonstrated in the following example. A `#define` is used to define a macro that represents the name of the C or C++ standard I/O header file. A `#include` is then used to make the header file available to the C or C++ program.

```
#define IO_HEADER   <stdio.h>
      .
      .
      .
#include IO_HEADER   /* equivalent to specifying #include <stdio.h>  */
      .
      .
      .
```

C++0x *Beginning of C++0x only.*

In C++0x, the changes to header and include file names in the C99 preprocessor are adopted to provide a common preprocessor interface for C and C++ compilers. The first character of a header file name in an `#include` directive must not be a digit in C++0x. For more information, see “C99 preprocessor features adopted in C++11 (C++11)” on page 403.

C++0x *End of C++0x only.*

Using the #include Directive When Compiling Source in an Integrated File System File

You can use the SRCSTMF keyword to specify an Integrated File System file at compile time. The #include processing differs from source physical file processing in that the library list is not searched. The search path specified by the INCLUDE environment variable (if it is defined), and the compiler's default search path are used to resolve header files.

The compiler's default include path is /QIBM/include.

#include files use the delimeters " " or <>.

When attempting to open the include file, the compiler searches in turn each directory in the search path until the file is found or all search directories have been exhausted.

The algorithm to search for include files is:

```
if file is fully qualified (a slash / starts the name) then
  attempt to open the fully qualified file
else
  if "" is delimeter, check job's current directory
  if not found:
     loop through the list of directories specified in the INCLUDE
          environment variable and then the default include path
```

```
    until the file is found or the end of the include path is encountered
endif
```

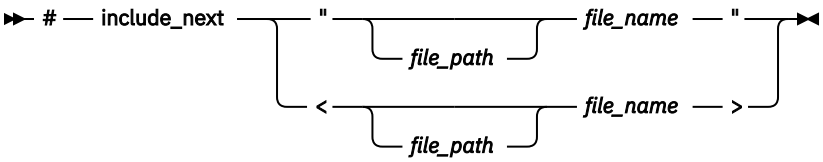
For more information, refer to *Using the ILE C/C++ Stream Functions with the IBM i Integrated File System* in *ILE C/C++ Programmer's Guide*.

The #include_next directive

IBM i *Beginning of 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.

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.

IBM i *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" on page 397, which conditionally include or suppress portions of source code, depending on the result of a constant expression
- "The #ifdef directive" on page 397, which conditionally includes source text if a macro name is defined
- "The #ifndef directive" on page 398, which conditionally includes source text if a macro name is not defined
- "The #else directive" on page 398, which conditionally includes source text if the previous `#if`, `#ifdef`, `#ifndef`, or `#elif` test fails
- "The #endif directive" on page 398, 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.

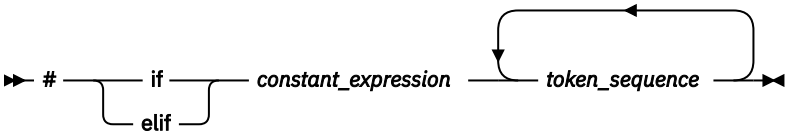
When a block following a preprocessor directive is to be ignored, the tokens are examined only to identify preprocessor directives within that block so that the conditional nesting level can be determined. All tokens other than the name of the directive are ignored.

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. C++0x In C++0x, arithmetic is performed using long long int type. See "C99 preprocessor features adopted in C++11 (C++11)" on page 403 for detailed information.
- 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:

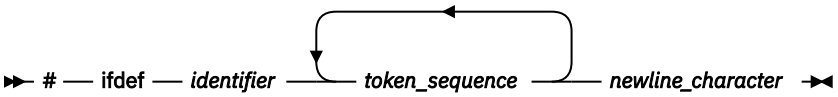
```
#if TEST >= 1
   printf("i = %d\n", i);
   printf("array[i] = %d\n", array[i]);
#elif TEST < 0
   printf("array subscript out of bounds \n");
#endif
```

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 following example defines MAX_LEN to be 75 if EXTENDED is defined for the preprocessor. Otherwise, MAX_LEN is defined to be 50.

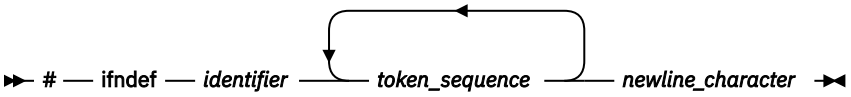
```
#ifdef EXTENDED
#   define MAX_LEN 75
#else
#   define MAX_LEN 50
#endif
```

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 following the condition are passed on to the compiler.

#ifndef directive syntax



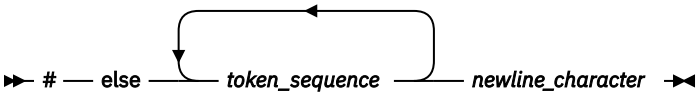
An identifier must follow the #ifndef keyword. The following example defines MAX_LEN to be 50 if EXTENDED is not defined for the preprocessor. Otherwise, MAX_LEN is defined to be 75.

```
#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:

```
/**
 ** This example contains preprocessor
 ** conditional compilation directives.
 **/

#include <stdio.h>

int main(void)
{
   static int array[ ] = { 1, 2, 3, 4, 5 };
   int i;

   for (i = 0; i <= 4; i++)
   {
      array[i] *= 2;

#if TEST >= 1
      printf("i = %d\n", i);
      printf("array[i] = %d\n",
      array[i]);
#endif

   }
   return(0);
}
```

Message generation directives

Message generation directives include the following:

- "The #error directive" on page 399, which defines text for a compile-time error message
- "The #warning directive" on page 400, which defines text for a compile-time warning message
- "The #line directive" on page 400, which supplies a line number for compiler messages

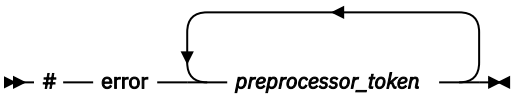
Related information

- "Conditional compilation directives" on page 395

The #error directive

A *preprocessor error directive* causes the preprocessor to generate an error message and causes the compilation to fail.

#error directive syntax



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

#if BUFFER_SIZE < 256
#error "BUFFER_SIZE is too small."
#endif
```

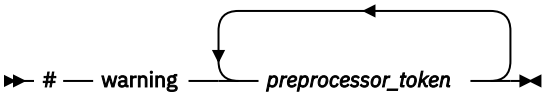
generates the error message:

```
BUFFER_SIZE is too small.
```

The #warning directive

A *preprocessor warning directive* causes the preprocessor to generate a warning message but allows compilation to continue. The argument to #warning is not subject to macro expansion.

#warning directive syntax

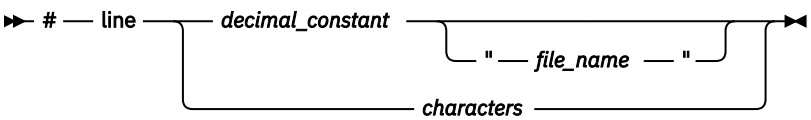


The preprocessor #warning directive is a language extension. The implementation preserves multiple white spaces.

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 ILE 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:

```
/**
 ** 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
```

C++0x *Beginning of C++0x only.*

In C++0x, the increased limit for #line directive from the C99 preprocessor are adopted to provide a common preprocessor interface for C and C++ compilers. The upper limit of #line <integer> preprocessor directives has been increased from 32,767 to 2,147,483,647 for the C++ preprocessor in conformance with the C99 preprocessor. For more information, see "C99 preprocessor features adopted in C++11 (C++11)" on page 403.

C++0x *End of C++0x only.*

Note: IBM i ILE C++ compiler supports #line directive only if DBGVIEW(*NONE) takes effective.

Related information

- **__C99_MAX_LINE_NUMBER** in the ILE C/C++ Compiler Reference
- **DBGVIEW** in the ILE C/C++ Compiler Reference

Assertion directives

C++ *Beginning of C++ only.*

An *assertion directive* is an alternative to a macro definition, used to define the computer or system the compiled program will run on. Assertions are usually predefined, but you can define them with the #assert preprocessor directive.

#assert directive syntax

The *predicate* represents the assertion entity you are defining. The *answer* represents a value you are assigning to the assertion. You can make several assertions using the same predicate and different

answers. All the answers for any given predicate are simultaneously true. For example, the following directives create assertions regarding font properties:

```
#assert font(arial)
#assert font(blue)
```

Once an assertion has been defined, the assertion predicate can be used in conditional directives to test the current system. The following directive tests whether `arial` or `blue` is asserted for `font`:

```
#if #font(arial) || #font(blue)
```

You can test whether any answer is asserted for a predicate by omitting the answer in the conditional:

```
#if #font
```

Assertions can be cancelled with the `#unassert` directive. If you use the same syntax as the `#assert` directive, the directive cancels only the answer you specify. For example, the following directive cancels the `arial` answer for the `font` predicate:

```
#unassert font(arial)
```

An entire predicate is cancelled by omitting the answer from the `#unassert` directive. The following directive cancels the `font` directive altogether:

```
#unassert font
```

Related information

- "Conditional compilation directives" on page 395

C++ *End of C++ only.*

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
```

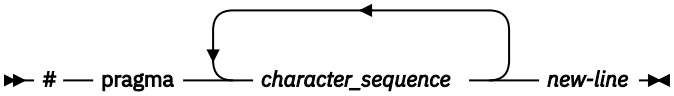
Related information

- "The # operator" on page 390

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 *new-line* character must terminate a pragma directive.

The *character_sequence* on a pragma is subject to macro substitutions. For example,

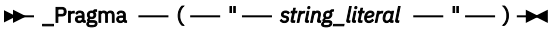
```
#define XX_ISO_DATA isolated_call(LG_ISO_DATA)
// ...
#pragma XX_ISO_DATA
```

More than one pragma construct can be specified on a single pragma directive. The compiler ignores unrecognized pragmas.

The _Pragma preprocessing operator

The unary operator `_Pragma`, which is a C99 feature, allows a preprocessor macro to be contained in a pragma directive.

_Pragma operator syntax



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)
```

C++0x *Beginning of C++0x only.*

In C++0x, the `_Pragma` operator feature of the C99 preprocessor is adopted to provide a common preprocessor interface for C and C++ compilers. The `_Pragma` operator is an alternative method of specifying the `#pragma` directive. For more information, see "C99 preprocessor features adopted in C++11 (C++11)" on page 403.

C++0x *End of C++0x only.*

C99 preprocessor features adopted in C++11 (C++11)

Note: IBM supports selected features of C++11, known as C++0x before its ratification. IBM will continue to develop and implement the features of this standard. The implementation of the language level is based on IBM's interpretation of this standard. Until IBM's implementation of all the C++11 features are complete, including the support of a new C++ standard library, the implementation may change from release to release. IBM makes no attempt to maintain compatibility, in source, binary, or listings and other compiler interfaces, with earlier releases of IBM's implementation of the new C++11 features and therefore they should not be relied on as a stable programming interface.

In the C++11 standard, several C99 preprocessor features are adopted to provide a common preprocessor interface for C and C++ compilers. This eases porting C source files to the C++ compiler and eliminates some subtle semantic differences that exist between the old C and C++ preprocessors, thus avoiding preprocessor compatibility issues or diverging preprocessor behaviors.

The following C99 preprocessor features are adopted in C++11:

- Preprocessor arithmetic with extended integer types
- Mixed string literal concatenation
- Diagnostic for header files and include names
- Increased limit for `#line` directives
- Diagnostic for object-like macro definitions
- The `_Pragma` operator

- Variadic macros and empty macro arguments

Preprocessor arithmetic with extended integer types

In the C89, C++98, and C++03 preprocessors, integer literals that have `int` or `unsigned int` type are widened to `long` or `unsigned long`. However, in the C99 and C++11 preprocessors, all `signed` and `unsigned` integer types (character types included) are widened to `long long` or `unsigned long long` under normal circumstances in ILE C/C++.

If this feature is enabled, and `OPTION(*NOLONGLONG)` is set, the preprocessor still uses `long long` or `unsigned long long` representations for all integral and character literals in preprocessor controlling expressions. The following example illustrates the case that `wchar_t`, whose underlying type is `unsigned short` or `unsigned int`, depending on LOCALETYPE, is widened to `unsigned long long`.

```
#if L'\x0' - L'\x1' < 0
#error non-C++11 preprocessor arithmetic.
#else
#error C++11 preprocessor arithmetic! L'\x0' and L'\x1' are widened to \
        unsigned long long
#endif
```

The following example shows a case where the long long support is enabled, this feature causes different inclusion branches to be chosen between the non-C++11 preprocessor and the C++11 preprocessor.

```
#if ~0ull == 0u + ~0u
#error C++11 preprocessor arithmetic! 0u has the same representation as 0ull, \
        hence ~0ull == 0u + ~0u
#else
#error non-C++11 preprocessor arithmetic. 0ul does not have the same representation as 0ull,\
        hence ~0ull != 0u + ~0u
#endif
```

Mixed string literal concatenation

Regular strings can be concatenated with wide-string literals, for example:

```
#include <wchar.h>
#include <stdio.h>
int main()
{
    wprintf(L"Guess what? %ls\n", "I can now concate" L"nate regular strings\
    and wide strings!");
    printf("Guess what? %ls\n", L"I can now concate" "nate strings\
    this way too!");
    return 0;
}
```

This example prints the following output when it is executed:

```
Guess what? I can now concatenate regular strings and wide strings!
Guess what? I can now concatenate strings this way too!
```

Diagnostic for header files and include names

When this feature is enabled, if the first character of a header file name in an `#include` directive is a digit, the compiler issues a warning message. Consider the following example:

```
//inc.C
#include "0x/mylib.h"
int main()
{
    return 0;
}
```

When compiling or preprocessing this example with this feature enabled, the compiler issues the following warning message:

```
"inc.C", line 1.10: CZP0893(10) The header file name "0x/mylib.h" in the #include directive
shall not start with a digit.
```

Increased limit for #line directives

The upper limit of the #line <integer> preprocessor directives has been increased from 32,767 to 2,147,483,647 for the C++11 preprocessor in conformance with the C99 preprocessor.

```
#line 1000000 //Valid in C++11, but invalid in C++98
int main()
{
    return 0;
}
```

Diagnostic for object-like macro definitions

If there is no white space between object-like macro name and its replacement list in a macro definition, the C++11 compiler issues a warning message. Consider the following example:

```
//w.C
//With LANGLVL(*EXTENDED0X), '$' is not part of the macro name,
//thus it begins the replacement list
#define A$B c
#define STR2( x ) # x
#define STR( x ) STR2( x )
char x[] = STR( A$B );
```

When compiling or preprocessing this example with this feature enabled, the compiler issues the following warning message:

```
"w.C", line 1.10: CZP0891(10) Missing white space between the identifier "A" and the
replacement list.
```

The _Pragma operator

The _Pragma operator is an alternative method of specifying #pragma directives. For example, the following two statements are equivalent:

```
#pragma comment(copyright, "IBM 2013")
_Pragma("comment(copyright, \"IBM 2013\")")
```

The string IBM 2013 is inserted into the C++ object file when the following code is compiled:

```
_Pragma("comment(copyright, \"IBM 2013\")")
int main()
{
   return 0;
}
```

Variadic macros and empty macro arguments

Variadic macros and empty macro arguments are supported in C99 and C++11. This feature enables a mechanism that renames the variable argument identifier from __VA_ARGS__ to a user-defined identifier. Consider the following example:

```
#define debug(...) fprintf(stderr, __VA_ARGS__)
#define showlist(...) puts(#__VA_ARGS__)
#define report(test, ...) ((test)?puts(#test): printf(__VA_ARGS__))
debug("Flag");
debug("X = %d\n", x);
showlist(The first, second, and third items.);
report(x>y, "x is %d but y is %d", x, y);
```

This example is expanded to the following code after preprocessing:

```
fprintf(stderr, "Flag");
fprintf(stderr, "X = %d\n", x);
puts("The first, second, and third items.");
((x>y)?puts("x>y"): printf("x is %d but y is %d", x, y));
```

Related information

- "Integer literals" on page 25

- “String literals” on page 33
- “The #include directive” on page 392
- “The #line directive” on page 400
- “The #define directive” on page 385
- “The _Pragma preprocessing operator” on page 403
- “Extensions for C++11 compatibility” on page 411

The ILE C language extensions

This topic presents the ILE C extensions in the following categories:

- “C99 features as extensions to C89” on page 407
- “Extensions for GNU C compatibility” on page 408
- “Extensions for decimal floating-point support” on page 408

C99 features as extensions to C89

The following features are enabled by default when you compile with the **LANGLVL(*EXTENDED)** option, which is the default language level. For more information, see the **LANGLVL** option in the *ILE C/C++ Compiler Reference*.

Table 29. Default C99 features as extensions to C89

Language feature	**Discussed in:**
`long long` data type	“Integral types” on page 54
Flexible array members at the end of a structure or union	“Flexible array members ” on page 67
Variable arguments in function-like macros	“Function-like macros” on page 387
C++ style comments	“Comments” on page 40
integer constant type rules	“Integer literals” on page 25
`_Pragma` operator	“The _Pragma preprocessing operator” on page 403
Mixed declarations and code	“Overview of data declarations and definitions” on page 45
Trailing comma allowed in enum declaration	“Enumeration type definition” on page 72
Conversion of array to pointer not limited to lvalues	“Pointer conversions” on page 120
Variable length arrays	“Variable length arrays” on page 98
Compound literals	“Compound literal expressions” on page 162
Designated initializers	“Designated initializers for aggregate types (C only)” on page 102
`_Bool` data type	“Boolean types” on page 55
Hexadecimal floating-point literals	“Hexadecimal floating-point literals” on page 30
`__func__` predefined identifier	“The __func__ predefined identifier” on page 24
Duplicate type qualifiers	“Type qualifiers” on page 79
Non-constant initialization of aggregate types	“Initialization of structures and unions” on page 104
Empty arguments in function-like macros	“Function-like macros” on page 387
Increased limit for `#line` directive	“The #line directive” on page 400
Flexible array members at the end of a structure or union	“Flexible array members ” on page 67

© Copyright IBM Corp. 1993, 2015

Table 29. Default C99 features as extensions to C89 (continued)	
Language feature	**Discussed in:**
Static and type qualifiers in parameter array declarators	“Static array indices in function parameter declarations (C only)” on page 206

The following features are enabled when you compile with the specified compile option.

Table 30. Default C99 features as extensions to C89, with individual option controls		
Language feature	**Discussed in:**	**Individual option control**
Digraphs	“Digraph characters” on page 39	OPTION(*DIGRAPH)

Related information

- "Invoking the compiler" in the *ILE C/C++ Compiler Reference*

Extensions for GNU C compatibility

The following feature is enabled by default at all language levels.

Table 31. Default ILE C extensions for GNU C compatibility	
Language feature	**Discussed in:**
`#include_next` preprocessor directive	“The #include_next directive” on page 395

The following features are enabled by default when you compile with the **LANGLVL(*EXTENDED)** option, which is the default language level. For more information, see the **LANGLVL** option in the *ILE C/C++ Compiler Reference*.

Table 32. Default ILE C extensions for GNU C compatibility	
Language feature	**Discussed in:**
`__alignof__` operator	“The __alignof__ operator” on page 138
`__typeof__` operator	“The __typeof__ operator” on page 140
Dollar signs in identifiers	“Characters in identifiers” on page 23
The __thread storage class specifier	“The __thread storage class specifier” on page 52

Related information

- "Invoking the compiler" in the *ILE C/C++ Compiler Reference*

Extensions for decimal floating-point support

The following feature requires compilation with the use of an additional option.

Language feature	**Discussed in:**	**Required compilation option**
Decimal floating-point types	“Real floating-point types” on page 55, “Decimal floating-point literals” on page 31, “Floating-point conversions ” on page 118	**LANGLVL(*EXTENDED)**

The ILE C++ language extensions

This topic presents the ILE C++ extensions to Standard C++ in the following categories:

- "General IBM extensions" on page 409
- "Extensions for C99 compatibility" on page 409
- "Extensions for GNU C compatibility" on page 410
- "Extensions for GNU C++ compatibility" on page 410
- "Extensions for C++11 compatibility" on page 411
- "Extensions for decimal floating-point support" on page 412

General IBM extensions

The following feature is enabled with the **LANGLVL(*EXTENDED)** option, which is the default language level. For more information, see the **LANGLVL** option in the *ILE C/C++ Compiler Reference*.

Language feature	Discussed in:
Non-C99 IBM `long long` extension	"Integral types" on page 54

Related information

- **OPTION(LONGLONG)** in the ILE C/C++ Compiler Reference

Extensions for C99 compatibility

ILE C++ adds support for the following C99 language features. The features are enabled with the **LANGLVL(*EXTENDED)** option, which is the default language level. For more information, see the **LANGLVL** option in the *ILE C/C++ Compiler Reference*.

Table 33. Default C99 features as extensions to Standard C++

Language feature	Discussed in:
Flexible array members at the end of a structure or union	"Flexible array members " on page 67
`_Pragma` operator	"The _Pragma preprocessing operator" on page 403
Additional predefined macro names	*ILE C/C++ Compiler Reference*
Empty arguments in function-like macros	"Function-like macros" on page 387
`__func__` predefined identifier	"The __func__ predefined identifier" on page 24
Hexadecimal floating-point literals	"Hexadecimal floating-point literals" on page 30
Trailing comma allowed in `enum` declaration	"Enumeration type definition" on page 72
The `restrict` type qualifier	"The restrict type qualifier(C++ only)" on page 83
Variable length arrays	"Variable length arrays" on page 98
Compound literals	"Compound literal expressions" on page 162
Variable arguments in function-like macros	"Function-like macros" on page 387

The following feature is only enabled by a specific compiler option.

© Copyright IBM Corp. 1993, 2015

Table 34. C99 features as extensions to Standard C++	
Language feature	**Discussed in:**
Universal character names	“The Unicode standard (C++ only)” on page 39

Extensions for GNU C compatibility

The following features are enabled with the **LANGLVL(*EXTENDED)** option, which is the default language level. For more information, see the **LANGLVL** option in the *ILE C/C++ Compiler Reference*.

Table 35. Default ILE C++ extensions for compatibility with GNU C	
Language feature	**Discussed in:**
Placement of flexible array members anywhere in structure or union	“Flexible array members ” on page 67
Static initialization of flexible array members of aggregates	“Flexible array members ” on page 67
`__alignof__` operator	“The __alignof__ operator” on page 138
`__typeof__` operator	“The __typeof__ operator” on page 140
Generalized lvalues	“Lvalues and rvalues” on page 125
Function attributes	“Function attributes” on page 209
`#include_next` preprocessor directive	“The #include_next directive” on page 395
Alternate keywords	“Keywords for language extensions” on page 22
`__extension__` keyword	“Keywords for language extensions” on page 22
Type attributes	“Type attributes” on page 84
Variable attributes	“Variable attributes” on page 113
Zero-extent arrays	“Zero-extent array members” on page 67
Variadic macro extensions	“Variadic macro extensions” on page 389
`#warning` preprocessor directive	“The #warning directive” on page 400
`#assert`, `#unassert` preprocessor directives	“Assertion directives” on page 401

The following feature requires compilation with the use of an additional option.

Table 36. ILE C++ extensions for GNU C compatibility, requiring additional compiler options	
Language feature	**Discussed in:**
Dollar signs in identifiers	“Characters in identifiers” on page 23

Extensions for GNU C++ compatibility

The following features are enabled by default when you compile with the **LANGLVL(*EXTENDED)**, which is the default language level.

Table 37. ILE C++ language extensions for compatibility with GNU C++	
Language feature	**Discussed in:**
Template instantiations declared as `extern`	“Template instantiation (C++ only)” on page 342

Table 37. ILE C++ language extensions for compatibility with GNU C++ (continued)	
Language feature	**Discussed in:**
The __thread storage class specifier	“The __thread storage class specifier” on page 52

The GNU C++ language extension "Template instantiations declared as `extern`" is documented as explicit instantiation declaration feature which is introduced in the C++0x standard. You can use the option **LANGLVL(*EXTENDED0X)** to control the explicit instantiation declaration for the same behavior as extern template. For more information, see "Explicit instantiation declaration" in the ILE C/C++ Language Reference.

Related information

- **LANGLVL** in the ILE C/C++ Compiler Reference

Extensions for C++11 compatibility

Note:

IBM supports selected features of C++11, known as C++0x before its ratification. IBM will continue to develop and implement the features of the new standard. The implementation of the language level is based on IBM's interpretation of this standard. Until IBM's implementation of all the C++11 features are complete, including the support of a new C++11 standard library, the implementation may change from release to release. IBM makes no attempt to maintain compatibility, in source, binary, or listings and other compiler interfaces, with earlier releases of IBM's implementation of the new C++11 features and therefore they should not be relied on as a stable programming interface.

The following features are part of a continual phased release process leading towards full compliance with C++11. They can be enabled by the **LANGLVL(*EXTENDED0X)** option.

Table 38. IBM ILE C++ language extensions for compatibility with C++11	
Language feature	**Discussed in:**
Auto type deduction	“The auto type specifier (C++11)” on page 57
C99 preprocessor features adopted in C++11	“C99 preprocessor features adopted in C++11 (C++11)” on page 403
Decltype	“The decltype(expression) type specifier (C++11)” on page 59
Defaulted and deleted functions	“Explicitly defaulted functions (C++11)” on page 193 “Deleted functions (C++11)” on page 194
Delegating constructors	“Delegating constructors (C++11)” on page 302
Explicit conversion operators	“Explicit conversion operators (C++11)” on page 318
Explicit instantiation declarations	“Explicit instantiation (C++ only)” on page 343
Extended friend declarations	“Friends (C++ only)” on page 269
Generalized constant expressions	“Generalized constant expressions (C++11)” on page 131
Inline namespace definitions	“Inline namespace definitions (C++11)” on page 229
Nullptr	“Null pointers ” on page 95

Table 38. IBM ILE C++ language extensions for compatibility with C++11 (continued)	
Language feature	**Discussed in:**
Reference collapsing	“Reference collapsing (C++11)” on page 171
Right angle brackets	“Class templates (C++ only)” on page 329
Rvalue references	“References (C++ only)” on page 99
Scoped enumerations	“Enumerations” on page 72
static_assert	“static_assert declaration (C++11)” on page 47
Trailing return type	“static_assert declaration (C++11)” on page 47
Variadic templates	“Variadic templates (C++11)” on page 352

Note: You can also use the **LANGLVL(*EXTENDED)** option to enable the explicit instantiation declarations feature.

Related information

- **LANGLVL** in the *ILE C/C++ Compiler Reference*

Extensions for decimal floating-point support

The following features are enabled by default when you compile with the **LANGLVL(*EXTENDED)** option, which is the default language level. For more information, see the **LANGLVL** option in the *ILE C/C++ Compiler Reference*.

Language feature	**Discussed in:**
Decimal floating-point types	“Real floating-point types” on page 55, “Decimal floating-point literals” on page 31, “Floating-point conversions ” on page 118

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 grant 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:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan Ltd.
1623-14, Shimotsuruma, Yamato-shi
Kanagawa 242-8502 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:

IBM Corporation
Software Interoperability Coordinator, Department YBWA
3605 Highway 52 N
Rochester, MN 55901
U.S.A.

© Copyright IBM Corp. 1993, 2015

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information is for planning purposes only. The information herein is subject to change before the products described become available.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

© (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs.

© Copyright IBM Corp. _enter the year or years_.

Programming interface information

This ILE C/C++ Compiler Reference publication documents intended Programming Interfaces that allow the customer to write programs to obtain the services of IBM i.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at "Copyright and trademark information" at www.ibm.com/legal/copytrade.shtml.

Adobe, the Adobe logo, PostScript, and the PostScript logo are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, and/or other countries.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Java and all Java-based trademarks and logos are trademarks of Oracle, Inc. in the United States, other countries, or both.

Other product and service names might be trademarks of IBM or other companies.

Terms and conditions

Permissions for the use of these publications is granted subject to the following terms and conditions.

Personal Use: You may reproduce these publications for your personal, noncommercial use provided that all proprietary notices are preserved. You may not distribute, display or make derivative works of these publications, or any portion thereof, without the express consent of IBM.

Commercial Use: You may reproduce, distribute and display these publications solely within your enterprise provided that all proprietary notices are preserved. You may not make derivative works of these publications, or reproduce, distribute or display these publications or any portion thereof outside your enterprise, without the express consent of IBM.

Except as expressly granted in this permission, no other permissions, licenses or rights are granted, either express or implied, to the publications or any information, data, software or other intellectual property contained therein.

IBM reserves the right to withdraw the permissions granted herein whenever, in its discretion, the use of the publications is detrimental to its interest or, as determined by IBM, the above instructions are not being properly followed.

You may not download, export or re-export this information except in full compliance with all applicable laws and regulations, including all United States export laws and regulations.

IBM MAKES NO GUARANTEE ABOUT THE CONTENT OF THESE PUBLICATIONS. THE PUBLICATIONS ARE PROVIDED "AS-IS" AND WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, AND FITNESS FOR A PARTICULAR PURPOSE.

Index

Special Characters

__align 80
__cdecl 219
__func__ 24
__typeof__ operator 140
__VA_ARGS__ 387, 389
_Pragma 403
- (subtraction operator) 144
- (unary minus operator) 135
-- (decrement operator) 134
-> (arrow operator) 133
, (comma operator) 150
:: (scope resolution operator) 130
! (logical negation operator) 135
!= (not equal to operator) 146
? : (conditional operators) 152
. (dot operator) 132
[ ] (array subscript operator) 149
* (indirection operator) 136
* (multiplication operator) 143
*= (compound assignment operator) 142
\ continuation character 34, 385
\ escape character 38
& (address operator) 135
& (bitwise AND operator) 147
& (reference declarator) 99
&& (logical AND operator) 148
&= (compound assignment operator) 142
preprocessor directive character 385
preprocessor operator 390
(macro concatenation) 391
% (remainder) 143
^ (bitwise exclusive OR operator) 147
^= (compound assignment operator) 142
+ (addition operator) 144
+ (unary plus operator) 134
++ (increment operator) 134
+= (compound assignment operator) 142
< (less than operator) 145
<< (left-shift operator) 144
<<= (compound assignment operator) 142
<= (less than or equal to operator) 145
= (simple assignment operator) 142
== (equal to operator) 146
> (greater than operator) 145
>= (greater than or equal to operator) 145
>> (right-shift operator) 144
>>= (compound assignment operator) 142
| (bitwise inclusive OR operator) 148
|| (logical OR operator) 149
~ (bitwise negation operator) 135
/ (division operator) 143
/= (compound assignment operator) 142
$ 23, 37

A

aborting functions 382
abstract classes 294, 295
access rules
    base classes 280
    class types 247, 267
    friends 274
    members 267
    multiple access 287
    protected members 279
    virtual functions 295
access specifiers
    in class derivations 280
accessibility 267, 287
addition operator (+) 144
address operator (&) 135
aggregate types
    initialization 104, 305
alias
    type-based aliasing 93
alignment
    bit fields 69
    structure members 66
    structures 114
    structures and unions 80
alignof operator 138
allocation
    expressions 163
    functions 216
ambiguities
    base and derived member names 287
    base classes 286
    resolving 176, 289
    virtual function calls 294
AND operator, bitwise (&) 147
AND operator, logical (&&) 148
argc (argument count)
    example 212
arguments
    default 217
    evaluation 218
    macro 387
    main function 212
    of catch blocks 373
    passing 191, 213
    passing by reference 214
    passing by value 214
    trailing 387, 389
argv (argument vector)
    example 212
arithmetic conversions 117
arithmetic types
    type compatibility 57
arrays
    array-to-pointer conversions 121
    as function parameter 206

arrays *(continued)*
declaration 206, 256
description 97
flexible array member 66, 67
initializing 108
multidimensional 98
subscripting operator 149
type compatibility 99
variable length 91, 98
zero-extent 66, 67
ASCII character codes 38
asm 21
assignment operator (=)
compound 142
pointers 94
simple 142
associativity of operators 167
atexit function 381
auto storage class specifier 49

B

base classes
abstract 295
access rules 280
ambiguities 286, 287
direct 285
indirect 276, 285
initialization 306
multiple access 287
pointers to 278
virtual 286, 290
base list 285
best viable function 242
binary expressions and operators 141
binding
direct 113
dynamic 291
static 291
virtual functions 291
bit fields
as structure member 66
type name 140
bitwise negation operator (~) 135
block statement 177
block visibility 12
Boolean
conversions 118
data types 55
literals 28
break statement 186
built-in data types 43

C

candidate functions 233, 242
case label 179
cast expressions 154
catch blocks
argument matching 373
order of catching 373
char type specifier 56
character
character *(continued)*
data types 56
literals 32
multibyte 34, 37
character set
extended 37
source 36
class members
access operators 132
access rules 267
class member list 255
declaration 256
initialization 306
order of allocation 256
class templates
declaration and definition 332
distinction from template class 329
explicit specialization 348
member functions 333
static data members 332
classes
abstract 295
access rules 267
aggregate 248
base 276
base list 276
class objects 43
class specifiers 247
class templates 329
declarations
incomplete 251, 256
derived 276
friends 269
inheritance 275
keywords 247
local 253
member functions 256
member lists 255
member scope 258
nested 251, 272
overview 247
polymorphic 247
scope of names 250
static members 263
this pointer 261
using declaration 281
virtual 286, 291
comma
in enumerator list 72
comments 40
compatibility
C89 and C99 407
data types 45
ILE C and GCC 408
ILE C++ and C99 409
ILE C++ and GCC 410
user-defined types 77
compatible types
across source files 77
arithmetic types 57
arrays 99
in conditional expressions 152
composite types
across source files 77

compound
assignment 142
expression 143
literal 162
statement 177
types 43
concatenation
macros 391
multibyte characters 34
conditional compilation directives
elif preprocessor directive 397
else preprocessor directive 398
endif preprocessor directive 398
examples 398
if preprocessor directive 397
ifdef preprocessor directive 397
ifndef preprocessor directive 398
conditional expression (? :) 143, 152
const
casting away constness 215
function attribute 210
member functions 258
object 125
placement in type name 91
qualifier 79
vs. #define 386
const_cast 159, 215
constant
const expressions 131
constant expressions 72, 128
constant initializers 255
constexpr
functions 220
constexpr constructor 303
constructors
converting 314–316
copy 319
exception handling 376
exception thrown in constructor 369
initialization
explicit 305
nontrivial 301, 311
overview 299
trivial 301, 311
continuation character 34, 385
continue statement 186
conversion
constructors 314, 315
function 317
implicit conversion sequences 243
operators 318
conversion sequence
ellipsis 244
implicit 243
standard 243
user-defined 244
conversions
arithmetic 117
array-to-pointer 121
Boolean 118
cast 154
explicit keyword 316
floating-point 118
function arguments 122
conversions *(continued)*
function-to-pointer 121
integral 118
lvalue-to-rvalue 120, 126, 243
pointer 120
pointer to derived class 290
pointer to member 260
qualification 122
references 122
standard 117
user-defined 313
void pointer 121
copy assignment operators 320
copy constructors 319
covariant virtual functions 292
cv-qualifier
in parameter type specification 234
syntax 79

D

data members
description 256
scope 258
static 264
data type
auto 57
data types
aggregates 43
Boolean 55
built-in 43
character 56
compatible 45
composite 45
compound 43
enumerated 72
floating 55
incomplete 44
integral 54
scalar 43
user-defined 43, 65
void 57
deallocation
expressions 166
functions 216
decimal
floating-point constants 31
decimal integer literals 27
declaration
static_assert 47
declarations
classes 247, 251
description 45
duplicate type qualifiers 80
friend specifier in member list 269
friends 274
pointers to members 259
resolving ambiguous statements 176
syntax 46, 91
unsubscripted arrays 98
declarative region 11
declarators
description 89
reference 99

decltype
specifier 59
decrement operator (--) 134
default
clause 179, 180
label 180
default constructor 300
defaulted functions 193
define preprocessor directive 385
defined unary operator 397
definitions
description 45
Inline namespace 229
macro 385
member function 257
tentative 46
Delegating constructors 302
delete operator 166
deleted functions 194
dependent names 362
dereferencing operator 136
derivation
array type 97
public, protected, private 280
derived classes
catch block 373
construction order 309
pointers to 278
designated initializer
union 104
designator
union 104
destructors
exception handling 376
exception thrown in destructor 369
overview 299
pseudo 312, 313
digraph characters 39
direct base class 285
division operator (/) 143
do statement 184
dollar sign 23, 37
dot operator 132
double type specifier 55
downcast 161
dynamic binding 291
dynamic_cast 160

E

EBCDIC character codes 38
elaborated type specifier 250
elif preprocessor directive 397
ellipsis
conversion sequence 244
in function declaration 205
in function definition 205
in macro argument list 387
else
preprocessor directive 398
statement 178
enclosing class 257, 272
endif preprocessor directive 398
entry point
entry point *(continued)*
program 212
enum
keyword 72
enumerations
compatibility 77
declaration 72
initialization 106
trailing comma 72
enumerator 73
equal to operator (==) 146
error preprocessor directive 399
escape character \ 38
escape sequence
alarm \a 38
backslash \\ 38
backspace \b 38
carriage return \r 38
double quotation mark \" 38
form feed \f 38
horizontal tab \t 38
new-line \n 38
question mark \? 38
single quotation mark \' 38
vertical tab \v 38
exception handling
argument matching 373
catch blocks
arguments 373
constructors 376
destructors 376
example, C++ 382
exception objects 367
function try blocks 367
handlers 367, 369
order of catching 373
rethrowing exceptions 375
set_terminate 382
set_unexpected 382
special functions 380
stack unwinding 376
terminate function 381
throw expressions 368, 374
try blocks 367
try exceptions 369
unexpected function 380
exceptions
declaration 369
function try block handlers 369
specification 378
exclusive OR operator, bitwise (^) 147
explicit
instantiation, templates 343
keyword 314–316, 318
specializations, templates 345, 347
type conversions 154
explicit specializations 346
exponent 29
expressions
allocation 163
assignment 141
binary 141
cast 154
comma 150

expressions *(continued)*
conditional 152
deallocation 166
description 125
integer constant 128
new initializer 165
parenthesized 129
pointer to member 152
primary 127
resolving ambiguous statements 176
statement 176
throw 167, 374
unary 133
Extensions
C++0x 411
extern storage class specifier
with variable length arrays 98

F

file inclusion 392, 395
file scope data declarations
unsubscripted arrays 98
flexible array member 67
float type specifier 55
floating-point
constant 30, 31
conversions 118
literal 28
promotion 119
floating-point types 55
for statement 184
free store
delete operator 166
new operator 163
friend
access rules 274
implicit conversion of pointers 281
member functions 256
nested classes 272
relationships with classes when templates are involved 333
scope 272
specifier 269
virtual functions 293
function attributes 209
function designator 126
function specifier
explicit 314–316
function templates
explicit specialization 348
function try blocks
handlers 369
function-like macro 387
functions
allocation 216
arguments
conversions 122
block 191
body 191
calling 213
calls
as lvalue 126
class templates 333
functions *(continued)*
constexpr 220
conversion function 317
deallocation 216
declaration
C++ 258
examples 195
multiple 196
parameter names 206
default arguments
evaluation 218
restrictions 218
definition
examples 195
exception handling 380
exception specification 378
friends 269
function call operator 191
function templates 334
function-to-pointer conversions 121
inline 199, 257
library functions 191
main 212
name
diagnostic 24
overloading 233
parameters 213
pointers to 219
polymorphic 276
predefined identifier 24
prototype 191
return statements 188
return type 191, 202, 203
return value 191, 203
signature 204
specifiable attributes 209
specifiers 199
template function
template argument deduction 335
type name 91
virtual 258, 291, 294

G

Generalized 131
global variable
uninitialized 101
goto statement
restrictions 189
greater than operator (>) 145
greater than or equal to operator (>=) 145

H

handlers 369
hexadecimal
floating-point constants 30
hexadecimal integer literals 28
hidden names 248, 250

I

identifiers

identifiers *(continued)*
case sensitivity 23
id-expression 90, 129
labels 175
linkage 17
namespace 15
predefined 24
reserved 21–23
special characters 23, 37
if
preprocessor directive 397
statement 178
ifdef preprocessor directive 397
ifndef preprocessor directive 398
implicit conversion
Boolean 118
floating-point 118
integral 118
lvalue 126
pointer to derived class 278, 281
pointers to base class 279
types 117
implicit instantiation
templates 342
include preprocessor directive 392
include_next preprocessor directive 395
inclusive OR operator, bitwise (|) 148
incomplete type
as structure member 66, 67
class declaration 251
incomplete types 44
increment operator (++) 134
indentation of code 385
indirect base class 276, 285
indirection operator (*) 136
information hiding 11, 12, 255, 279
inheritance
multiple 276, 285
overview 275
initialization
aggregate types 104
auto object 101
base classes 306
class members 306
extern object 101
references 122
register object 102
static data members 265
static object 101
union member 105
initializer lists 100, 306
initializers
aggregate types 104
enumerations 106
unions 105
inline
function specifier 199
functions 199, 257
Inline namespace definitions (C++0x) 229
integer
constant expressions 72, 128
data types 54
literals 25
promotion 119
integral
conversions 118

K

keywords
exception handling 367
language extension 22
template 323, 363, 364
underscore characters 22

L

label
implicit declaration 12
in switch statement 180
statement 175
language extension
C99 407
GNU C 407
left-shift operator (<<) 144
less than operator (<) 145
less than or equal to operator (<=) 145
line preprocessor directive 400
linkage
auto storage class specifier 49
const cv-qualifier 82
extern storage class specifier 19, 51
external 17
in function definition 197
inline member functions 257
internal 17, 49, 197
language 18
multiple function declarations 196
none 18
register storage class specifier 52
specifications 18
static storage class specifier 50
weak symbols 116
linking to non-C++ programs 18
literals
Boolean 28
character 32
compound 162
floating-point 28
integer
decimal 27
hexadecimal 28
octal 28
pointer 35, 95
string 33
local
classes 253
type names 254
logical operators
! (logical negation) 135
&& (logical AND) 148
|| (logical OR) 149
long double type specifier 55
long type specifier 54
lvalues
casting 154
conversions 120, 126, 243

M

macro
  definition
    __typeof__ operator 140
  function-like 387
  invocation 387
  object-like 386
  variable argument 387, 389
main function
  arguments 212
  example 212
member functions
  const and volatile 258
  definition 257
  friend 256
  special 258
  static 265
  this pointer 261, 295
member lists 248, 255
members
  access 267
  access control 284
  class member access operators 132
  data 256
  pointers to 152, 259
  protected 279
  scope 258
  static 252, 263
  virtual functions 258
modifiable lvalue 125, 141, 142
modulo operator (%) 143
multibyte character
  concatenation 34
multicharacter literal 32
multidimensional arrays 98
multiple
  access 287
  inheritance 276, 285
multiplication operator (*) 143
mutable storage class specifier 51

N

name binding 362
name hiding
  accessible base class 290
  ambiguities 288
names
  conflicts 15
  hidden 130, 248, 250
  local type 254
  mangling 19
  resolution 12, 281, 289
namespace
  class names 250
  context 15
  of identifiers 15
namespaces
  alias 223, 224
  declaring 223
  defining 223
  explicit access 228
  extending 224
namespaces *(continued)*
  friends 227
  member definitions 226
  namespace scope object
    exception thrown in constructor 369
  overloading 224
  unnamed 225
  user-defined 13
  using declaration 228
  using directive 227
narrow character literal 32
nested classes
  friend scope 272
  scope 251
new operator
  default arguments 218
  description 163
  initializer expression 165
  placement syntax 164
  set_new_handler function 165
not equal to operator (!=) 146
null
  character \0 33
  pointer 107
  pointer constants 121
  preprocessor directive 402
  statement 190
number sign (#)
  preprocessor directive character 385
  preprocessor operator 390

O

object-like macro 386
objects
  class
    declarations 248
  description 43
  lifetime 11
  namespace scope
    exception thrown in constructor 369
  restrict-qualified pointer 83
  static
    exception thrown in destructor 369
octal integer literals 28
one's complement operator (~) 135
operator functions 235
operator specifier
  explicit 318
operators
  __typeof__ 140
  - (subtraction) 144
  - (unary minus) 135
  -- (decrement) 134
  -> (arrow) 133
  ->* (pointer to member) 152
  , (comma) 150
  :: (scope resolution) 130
  ! (logical negation) 135
  != (not equal to) 146
  ? : (conditional) 152
  . (dot) 132
  .* (pointer to member) 152
  () (function call) 132, 191

operators *(continued)*
* (indirection) 136
* (multiplication) 143
/ (division) 143
& (address) 135
& (bitwise AND) 147
&& (logical AND) 148
% (remainder) 143
^ (bitwise exclusive OR) 147
+ (addition) 144
++ (increment) 134
< (less than) 145
<< (left- shift) 144
<= (less than or equal to) 145
= (simple assignment) 142
== (equal to) 146
> (greater than) 145
>= (greater than or equal to) 145
>> (right- shift) 144
| (bitwise inclusive OR) 148
|| (logical OR) 149
alternative representations 36
assignment
copy assignment 320
associativity 167
binary 141
bitwise negation operator (~) 135
compound assignment 142
const_cast 159
converting 318
defined 397
delete 166
dynamic_cast 160
equality 146
new 163
overloading
binary 238
unary 236
pointer to member 152, 260
precedence
examples 170
type names 91
preprocessor
390
391
pragma 403
reinterpret_cast 158
relational 145
scope resolution 278, 288, 294
sizeof 138
static_cast 156
typeid 137
unary 133
unary plus operator (+) 134
OR operator, logical (||) 149
overload resolution
resolving addresses of overloaded functions 244
overloading
description 233
function templates 340
functions
restrictions 234
operators
assignment 238
overloading *(continued)*
operators *(continued)*
binary 238
class member access 241
decrement 237
function call 239
increment 237
subscripting 240
unary 236
overriding virtual functions
covariant virtual function 292

P

packed
structure member 68
variable attribute 115
parenthesized expressions 91, 129
pass by reference 99, 214
pass by value 214
placement syntax 164
pointer
literal 35, 95
pointer to member
conversions 260
declarations 259
operators 152, 260
pointers
conversions 120, 121, 290
cv-qualified 92
dereferencing 93
description 92
generic 121
null 107
pointer arithmetic 93
restrict-qualified 83
this 261
to functions 219
to members 152, 259
type-qualified 92
void* 120
polymorphism
polymorphic classes 247, 292
polymorphic functions 276
postfix
++ and -- 134
pound sign (#)
preprocessor directive character 385
preprocessor operator 390
pragma operator 403
pragmas
_Pragma 403
preprocessor directive 402
precedence of operators 167
predefined identifier 24
prefix
++ and -- 134
decimal floating-point constants 31
hexadecimal floating-point constants 30
hexadecimal integer literals 28
octal integer literals 28
preprocessor directives
conditional compilation 395
preprocessing overview 385

preprocessor directives *(continued)*
special character 385
warning 400
preprocessor features
C99
adopte 403
preprocessor operator
_Pragma 403
390
391
primary expressions 127
promotions
integral and floating-point 119
pseudo-destructors 312
punctuators
alternative representations 36
pure specifier 255, 258, 294, 295
pure virtual functions 295

Q

qualification conversions 122
qualified name 130, 252
qualifiers
const 79
in parameter type specification 234
restrict 83
volatile 79, 84

R

reference
collapsing 171
reference collapsing 171
references
as return types 203
binding 110, 111
conversions 122
declarator 136
description 99
initialization 110
register storage class specifier 52
reinterpret_cast 158
remainder operator (%) 143
restrict
in parameter type specification 234
return statement 188, 203
return type
reference as 203
size_t 138
right-shift operator (>>) 144
RTTI support 137, 161
rvalues 125

S

scalar types 43, 92
scope
class 14
class names 250
description 11
enclosing and nested 12
friends 272
scope *(continued)*
function 12
function prototype 13
global 13
global namespace 13
identifiers 15
local (block) 12
local classes 253
macro names 390
member 258
nested classes 251
scope resolution operator
ambiguous base classes 288
description 130
inheritance 278
virtual functions 294
sequence point 151
set_new_handler function 165
set_terminate function 382
set_unexpected function 380, 382
shift operators << and >> 144
short type specifier 54
side effect 84
signed type specifiers
char 56
int 54
long 54
long long 54
size_t 138
sizeof operator
with variable length arrays 99
space character 385
special characters 37
special member functions 258
specifier
constexpr 63
specifiers
access control 280
inline 199
pure 258
storage class 48
splice preprocessor directive ## 391
stack unwinding 376
standard type conversions 117
statements
block 177
break 186
continue 186
do 184
expressions 176
for 184
goto 189
if 178
labels 175
null 190
resolving ambiguities 176
return 188, 203
selection 178, 179
switch 179
while 183
static
binding 291
data members 264
in array declaration 206

static *(continued)*
initialization of data members 265
member functions 265
members 252, 263
storage class specifier
linkage 50
with variable length arrays 98
static storage class specifier 17
static_assert declaration (C++0x) 47
static_cast 156
storage class specifiers
auto 49
extern 50, 197
mutable 51
register 52
static 49, 197
storage duration
auto storage class specifier 49
extern storage class specifier 51
register storage class specifier 52
static
exception thrown in destructor 369
string
literal 33
terminator 33
stringize preprocessor directive # 390
struct type specifier 66
structures
alignment 80
as base class 281
as class type 247, 248
compatibility 77
flexible array member 66, 67
identifier (tag) 66
initialization 104
members
alignment 66
incomplete types 67
layout in memory 66, 104
packed 68
padding 66
zero-extent array 66
namespaces within 15
packed 66
unnamed members 104
subscript declarator
in arrays 97
subscripting operator
in type name 91
subtraction operator (-) 144
suffix
decimal floating-point constants 31
floating-point literals 28
hexadecimal floating-point constants 30
integer literal constants 25
switch statement 179

T

tags
enumeration 72
structure 66
union 66
template arguments
deduction 335, 361
deduction, non-type 339
deduction, type 337
non-type 327
template 328
type 327
variadic 352
template keyword 364
templates
arguments
non-type 327
type 327
class
declaration and definition 332
distinction from template class 329
explicit specialization 348
member functions 333
static data members 332
declaration 323
dependent names 362
explicit specializations
class members 347
definition and declaration 347
function templates 348
function
argument deduction 339
overloading 340
partial ordering 341
function templates
type template argument deduction 337
instantiation
explicit 343
implicit 342
name binding 362
parameters
default arguments 325
friend 326
non-type 324
template 325
type 324
partial specialization
matching 351
parameter and argument lists 351
point of definition 363
point of instantiation 363
relationship between classes and their friends 333
scope 347
specialization 323, 342, 345
variadic
argument, deduction 361
pack, expansion 354
parameter, packs 352
partial, specialization 360
temporary objects 373
tentative definition 46
terminate function
set_terminate 382
this pointer 261, 295
throw expressions
argument matching 373
rethrowing exceptions 375
within nested try blocks 368
tokens

tokens *(continued)*
alternative representations for operators and punctuators 36
trailing return 207
translation unit 11
trigraph sequences 40
truncation
integer division 143
try blocks
nested 368
try keyword 367
type attributes 84
type name
__typeof__ operator 140
local 254
qualified 130, 252
typename keyword 363
type qualifiers
const 79, 82
const and volatile 90
duplicate 80
in function parameters 234
restrict 79, 83
volatile 79
type specifier
class types 247
elaborated 250
overriding 116
type specifiers
_Bool 55
bool 55
char 56
double 55
enumeration 72
float 55
int 54
long 54
long double 55
long long 54
short 54
unsigned 54
void 57
wchar_t 54, 56
typedef specifier
class declaration 254
local type names 254
pointers to members 260
qualified type name 252
with variable length arrays 99
typeid operator 137
typename keyword 363
types
class 247
conversions 154
type-based aliasing 93

U

unary expressions 133
unary operators
minus (-) 135
plus (+) 134
undef preprocessor directive 390
underscore character 22, 23
unexpected function
set_unexpected 382
Unicode 39
unions
as class type 247, 248
compatibility 77
initialization 105
specifier 66
unnamed members 104
universal character name 23, 32, 39
unnamed namespaces 225
unsigned type specifiers
char 56
int 54
long 54
long long 54
short 54
unsubscripted arrays
description 98, 206
user-defined conversions 313
user-defined data types 43, 65
using declarations
changing member access 284
overloading member functions 282
using directive 227

V

variable attributes 113
variable length array
as function parameter 99, 213, 242
sizeof 128
type name 91
variably modified types 98, 181
virtual
base classes 277, 286, 290
virtual functions
access 295
ambiguous calls 294
overriding 294
pure specifier 295
visibility
block 12
class members 268
void
in function definition 202, 205
pointer 120, 121
volatile
member functions 258
qualifier 79, 84

W

warning preprocessor directive 400
wchar_t type specifier 32, 54, 56
weak symbol 116
while statement 183
white space 21, 40, 385, 391
wide characters
literals 32
wide string literal 34

Z

zero-extent array 67

Product Number: 5770-WDS

SC09-7852-04