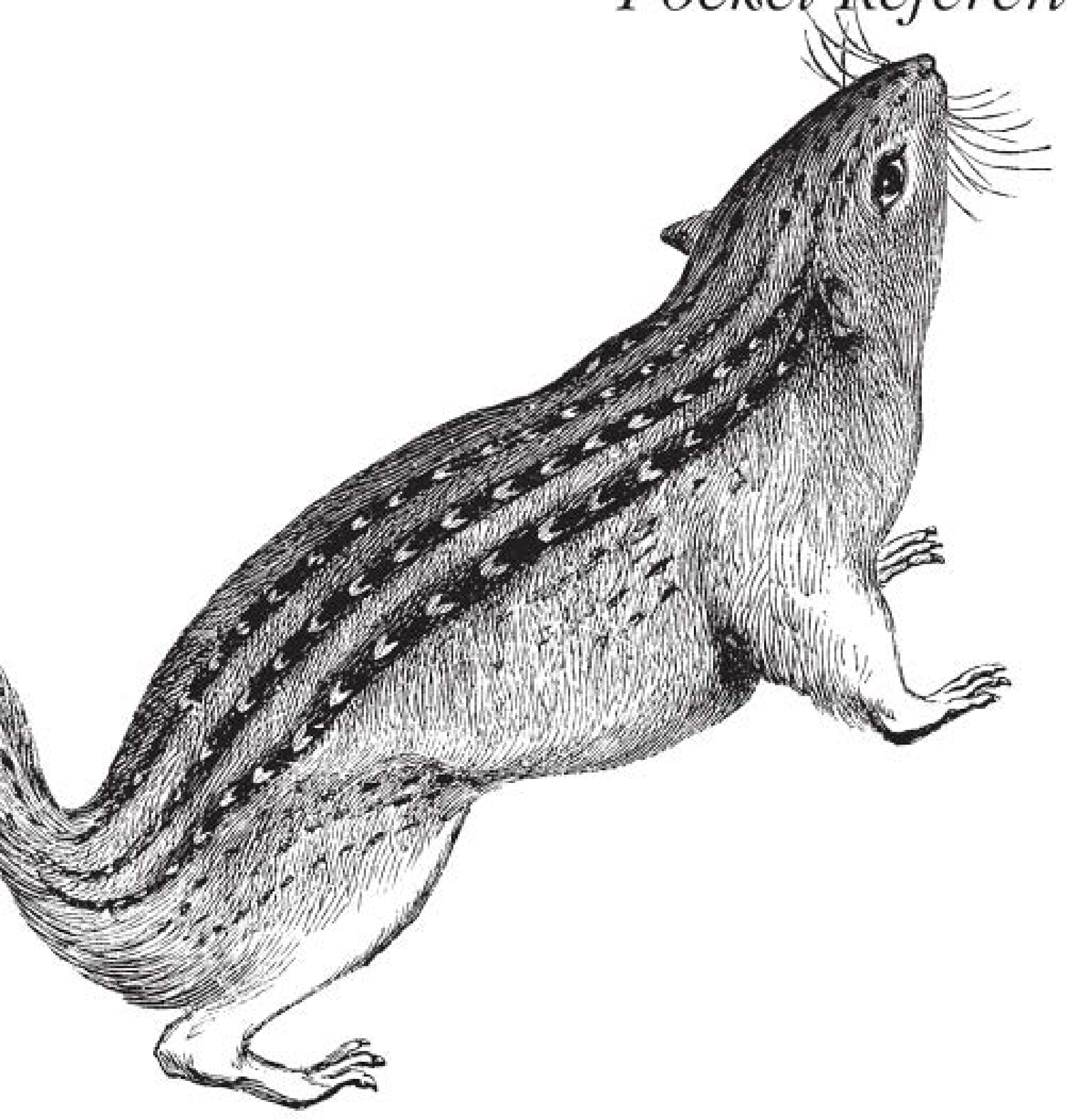
C++ Syntax and Fundamentals

C++

Pocket Reference



O'REILLY®

Kyle Loudon

C++
Pocket Reference

Kyle Loudon

O'REILLY®

Beijing · Cambridge · Farnham · Köln · Paris · Sebastopol · Taipei · Tokyo

C++ Pocket Reference
by Kyle Loudon

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly Media, Inc. books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*safari.oreilly.com*). For more information, contact our corporate/institutional sales department: (800) 998-9938 or *corporate@oreilly.com*.

Editor:	Jonathan Gennick
Production Editor:	Emily Quill
Cover Designer:	Ellie Volckhausen
Interior Designer:	David Futato

Printing History:

May 2003:	First Edition.

ISBN-10: 0-596-00496-6
ISBN-13: 978-0-596-00496-5
[T] [9/07]

Typographic Conventions

This book uses the following typographic conventions:

Italic
: This style is used for filenames and for items emphasized in the text.

`Constant width`
: This style is used for code, commands, keywords, and names for types, variables, functions, and classes.

`Constant width italic`
: This style is used for items that you need to replace.

Acknowledgments

I would like to thank Jonathan Gennick, my editor at O'Reilly, for his support and direction with this book. Thanks also to Uwe Schnitker, Danny Kalev, and Ron Passerini for taking the time to read and comment on an early draft of this book.

Compatibility with C

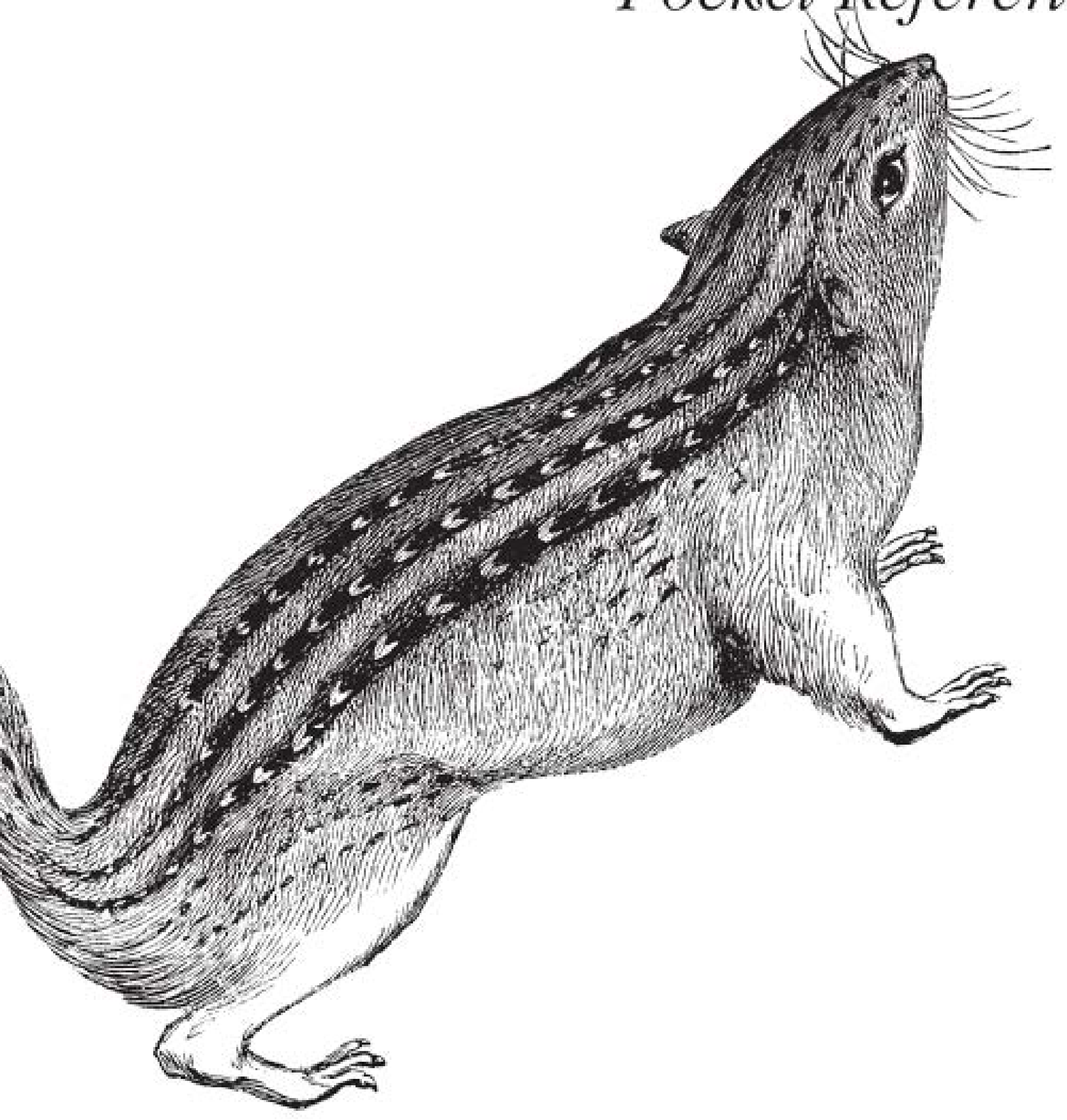
With some minor exceptions, C++ was developed as an extension, or superset, of C. This means that well-written C programs generally will compile and run as C++ programs. (Most incompatibilities stem from the stricter type checking that C++ provides.) So, C++ programs tend to look syntactically similar to C and use much of C's original functionality.

This being said, don't let the similarities between C and C++ fool you into thinking that C++ is merely a trivial derivation of C. In fact, it is a rich language that extends C with some grand additions. These include support for object-oriented programming, generic programming using templates, namespaces, inline functions, operator and function overloading, better facilities for memory management, references, safer forms of casting, runtime type information, exception handling, and an extended standard library.

C++ Syntax and Fundamentals

C++

Pocket Reference



O'REILLY®

Kyle Loudon

C++
Pocket Reference

Kyle Loudon

O'REILLY®

Beijing · Cambridge · Farnham · Köln · Paris · Sebastopol · Taipei · Tokyo

C++ Pocket Reference
by Kyle Loudon

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly Media, Inc. books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*safari.oreilly.com*). For more information, contact our corporate/institutional sales department: (800) 998-9938 or *corporate@oreilly.com*.

Editor:	Jonathan Gennick
Production Editor:	Emily Quill
Cover Designer:	Ellie Volckhausen
Interior Designer:	David Futato

Printing History:

May 2003:	First Edition.

ISBN-10: 0-596-00496-6
ISBN-13: 978-0-596-00496-5
[T] [9/07]

Typographic Conventions

This book uses the following typographic conventions:

Italic
: This style is used for filenames and for items emphasized in the text.

`Constant width`
: This style is used for code, commands, keywords, and names for types, variables, functions, and classes.

`Constant width italic`
: This style is used for items that you need to replace.

Acknowledgments

I would like to thank Jonathan Gennick, my editor at O'Reilly, for his support and direction with this book. Thanks also to Uwe Schnitker, Danny Kalev, and Ron Passerini for taking the time to read and comment on an early draft of this book.

Compatibility with C

With some minor exceptions, C++ was developed as an extension, or superset, of C. This means that well-written C programs generally will compile and run as C++ programs. (Most incompatibilities stem from the stricter type checking that C++ provides.) So, C++ programs tend to look syntactically similar to C and use much of C's original functionality.

This being said, don't let the similarities between C and C++ fool you into thinking that C++ is merely a trivial derivation of C. In fact, it is a rich language that extends C with some grand additions. These include support for object-oriented programming, generic programming using templates, namespaces, inline functions, operator and function overloading, better facilities for memory management, references, safer forms of casting, runtime type information, exception handling, and an extended standard library.

Program Structure

At the highest level, a C++ program is composed of one or more *source files* that contain C++ source code. Together, these files define exactly one starting point, and perhaps various points at which to end.

C++ source files frequently import, or *include*, additional source code from *header files*. The C++ preprocessor is responsible for including code from these files before each file is compiled. At the same time, the preprocessor can also perform various other operations through the use of *preprocessor directives*. A source file after preprocessing has been completed is called a *translation unit*.

Startup

The function `main` is the designated start of a C++ program, which you as the developer must define. In its standard form, this function accepts zero or two arguments supplied by the operating system when the program starts, although many C++ implementations allow additional parameters. Its return type is `int`. For example:

```
int main( )
int main(int argc, char *argv[])
```

`argc` is the number of arguments specified on the command line; `argv` is an array of null-terminated (C-style) strings containing the arguments in the order they appear. The name of the executable is stored in `argv[0]`, and may or may not be prefixed by its path. The value of `argv[argc]` is 0.

The following shows the `main` function for a simple C++ program that prompts the user for actions to perform on an account:

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

```
#include "Account.h"

int main(int argc, char *argv[])
{
   Account       account(0.0);
   char          action;
   double        amount;

   if (argc > 1)
      account.deposit(atof(argv[1]));

   while (true)
   {
      cout << "Balance is "
           << account.getBalance( )
           << endl;

      cout << "Enter d, w, or q: ";
      cin  >> action;

      switch (action)
      {
         case 'd':
            cout << "Enter deposit: ";
            cin  >> amount;
            account.deposit(amount);
            break;

         case 'w':
            cout << "Enter withdrawal: ";
            cin  >> amount;
            account.withdraw(amount);
            break;

         case 'q':
            exit(0);

         default:
            cout << "Bad command" << endl;
      }
   }

   return 0;
}
```

The class for the account is defined in a later example. An initial deposit is made into the account using an amount specified on the command line when the program is started. The function `atof` (from the C++ Standard Library) is used to convert the command-line argument from a string to a `double`.

Termination

A C++ program terminates when you return from `main`. The value you return is passed back to the operating system and becomes the return value for the executable. If no return is present in `main`, an implicit return of 0 takes places after falling through the body of `main`. You can also terminate a program by calling the `exit` function (from the C++ Standard Library), which accepts the return value for the executable as an argument.

Header Files

Header files contain source code to be included in multiple files. They usually have a *.h* extension. Anything to be included in multiple places belongs in a header file. A header file should never contain the following:

- Definitions for variables and static data members (see "Declarations" for the difference between declarations and definitions).
- Definitions for functions, except those defined as template functions or inline functions.
- Namespaces that are unnamed.

NOTE

Header files in the C++ Standard Library do not use the *.h* extension; they have no extension.

Often you create one header file for each major class that you define. For example, Account is defined in the header file *Account.h*, shown below. Of course, header files are used for other purposes, and not all class definitions need to be in header files (e.g., helper classes are defined simply within the source file in which they will be used).

```
#ifndef ACCOUNT_H
#define ACCOUNT_H

class Account
{
public:
                  Account(double b);

   void           deposit(double amt);
   void           withdraw(double amt);
   double         getBalance( ) const;

private:
   double         balance;
};

#endif
```

The implementation of this class is in *Account.cpp*. You use the preprocessor directive #include to include a header file within another file (see "Preprocessor Directives").

Because header files are often included by other headers themselves, care must be taken not to include the same file more than once, which can lead to compilation errors. To avoid this, it is conventional to wrap the contents of header files with the preprocessor directives #ifndef, #define, and #endif, as done in the previous example.

The tactic of wrapping a header file forces the preprocessor to test an identifier. If that identifier is not defined, the preprocessor defines it and processes the file's contents. As an example, the contents of *Account.h* are processed only when ACCOUNT_H is undefined, and the first thing that processing does is to define ACCOUNT_H to ensure the header is not processed a second time. To ensure uniqueness, *X_H* is typically used as the identifier, where *X* is the name of the header file without its extension.

Source Files

C++ source files contain C++ source code. They usually have a *.cpp* extension. During compilation, the compiler typically translates source files into object files, which often have a *.obj* or *.o* extension. Object files are joined by the linker to produce a final executable or library.

Often you create one source file for each major class you implement. For example, the implementation of `Account` is in *Account.cpp*, shown below. Of course, there is no requirement about this; source files often contain more than just the implementation of a single class.

```
#include "Account.h"

Account::Account(double b)
{
   balance = b;
}

void Account::deposit(double amt)
{
   balance += amt;
}

void Account::withdraw(double amt)
{
   balance -= amt;
}

double Account::getBalance( ) const
{
   return balance;
}
```

Preprocessor Directives

The C++ preprocessor can be used to perform a number of useful operations controlled via several directives. Each directive begins with a pound sign (#) as the first character that is not whitespace on a line. Directives can span multiple lines by including a backslash (\) at the end of intermediate lines.

#define

The `#define` directive replaces an identifier with the text that follows it wherever the identifier occurs in a source file. For example:

```
#define             BUFFER_SIZE 80

char                buffer[BUFFER_SIZE];
```

If you specify no text after the identifier, the preprocessor simply defines the identifier so that any check for its definition tests true and it expands to nothing in the source code. (You can see this in use earlier where `ACCOUNT_H` was defined.)

NOTE

In C++, it is preferable to use enumerations, and to a lesser degree, variables and data members declared using the keywords `const` or `static const` for constant data, rather than the `#define` directive.

The `#define` directive can accept arguments for macro substitution in the text. For example:

```
#define MIN(a, b) (((a) < (b)) ? (a):(b))

int                 x = 5, y = 10, z;

z = MIN(x, y);      // This sets z to 5.
```

In order to avoid unexpected problems with operator precedence, parameters should be fully parenthesized in the text, as shown above.

NOTE

In C++, it is preferable to use templates and inline functions in place of macros. Templates and inline functions eliminate unexpected results produced by macros, such as `MIN(x++, y)` incrementing `x` twice when `a` is less than `b`. (Macro substitution treats `x++`, not the result of `x++`, as the first parameter.)

#undef

The `#undef` directive undefines an identifier so that a check for its definition tests false. For example:

```
#undef LOGGING_ENABLED
```

#ifdef, #ifndef, #else, #endif

You use the `#ifdef`, `#ifndef`, `#else`, and `#endif` directives together. The `#ifdef` directive causes the preprocessor to include different code based on whether or not an identifier is defined. For example:

```
#ifdef LOGGING_ENABLED
cout << "Logging is enabled" << endl;
#else
cout << "Logging is disabled" << endl;
#endif
```

Using `#else` is optional. `#ifndef` works similarly but includes the code following the `#ifndef` directive only if the identifier is not defined.

#if, #elif, #else, #endif

The `#if`, `#elif`, `#else`, and `#endif` directives, like the directives of `#ifdef`, are used together. These cause the preprocessor to include or exclude code based on whether an expression is true. For example:

```
#if (LOGGING_LEVEL == LOGGING_MIN && \
    LOGGING_FLAG)
cout << "Logging is minimal" << endl;
```

```
#elif (LOGGING_LEVEL == LOGGING_MAX && \
    LOGGING_FLAG)
cout << "Logging is maximum" << endl;
#elif LOGGING_FLAG
cout << "Logging is standard" << endl;
#endif
```

The #elif (else-if) directive is used to chain a series of tests together, as shown above.

#include

The #include directive causes the preprocessor to include another file, usually a header file. You enclose standard header files with angle brackets, and user-defined header files with quotes. For example:

```
#include <iostream>
#include "Account.h"
```

The preprocessor searches different paths depending on the form of enclosure. The paths searched depend on the system.

#error

The #error directive causes compilation to stop and a specified string to be displayed. For example:

```
#ifdef LOGGING_ENABLED
#error Logging should not be enabled
#endif
```

#line

The #line directive causes the preprocessor to change the current line number stored internally by the compiler during compilation in the macro __LINE__. For example:

```
#line 100
```

A filename optionally can be specified in double quotes after the line number. This changes the name of the file stored internally by the compiler in the macro __FILE__. For example:

```
#line 100 "NewName.cpp"
```

#pragma

Some operations that the preprocessor can perform are implementation-specific. The `#pragma` directive allows you to control these operations by specifying the directive along with any parameters in a form that the directive requires. For example:

```
#ifdef LOGGING_ENABLED
#pragma message("Logging enabled")
#endif
```

Under Microsoft Visual C++ 6.0, the `message` directive informs the preprocessor to display a message during compilation at the point where this line is encountered. The directive requires one parameter: the message to display. This is enclosed in parentheses and quoted.

Preprocessor Macros

The C++ preprocessor defines several macros for inserting information into a source file during compilation. Each macro begins and ends with two underscores, except for `__cplusplus`, which has no terminating underscores.

`__LINE__`
: Expands to the current line number of the source file being compiled.

`__FILE__`
: Expands to the name of the source file being compiled.

`__DATE__`
: Expands to the date on which the compilation is taking place.

`__TIME__`
: Expands to the time at which the compilation is taking place.

`__TIMESTAMP__`
: Expands to the date and time at which the compilation is taking place.

`__STDC__`
: Will be defined if the compiler is in full compliance with the ANSI C standard.

`__cplusplus`
: Will be defined if the program being compiled is a C++ program. How a compiler determines whether a given program is a C++ program is compiler-specific. You may need to set a compiler option, or your compiler may look at the source file's extension.

Fundamental Types

The type for an identifier determines what you are allowed to do with it. You associate a type with an identifier when you declare it. When declaring an identifier, you also may have the opportunity to specify a storage class and one or more qualifiers (see "Declarations").

The fundamental types of C++ are its Boolean, character, integer, floating-point, and void types. The Boolean, character, and integer types of C++ are called *integral types*. Integral and floating-point types are collectively called *arithmetic types*.

bool

Booleans are of type `bool`. The `bool` type is used for values of truth. For example:

```
bool            flag;
...
if (flag)
{
   // Do something when the flag is true.
}
```

Boolean values

Booleans have only two possible values: `true` or `false`. The typical size of a `bool` is one byte.

Boolean literals

The only Boolean literals are the C++ keywords `true` and `false`. By convention, false is defined as 0; any other value is considered true.

char and wchar_t

Characters are of type `char` or `wchar_t`. The `char` type is used for integers that refer to characters in a character set (usually ASCII). For example:

```
char            c = 'a';

cout << "Letter a: " << c << endl;
```

The `wchar_t` type is a distinct type large enough to represent the character sets of all locales supported by the implementation. To use facilities related to the `wchar_t` type, you include the standard header file *<cwchar>*.

Character types may be specified either as `signed` or `unsigned` and are sometimes used simply to store small integers. For example:

```
signed char       small = -128;
unsigned char     flags = 0x7f;
```

A `signed char` represents both positive and negative values, typically by sacrificing one bit to store a sign. An `unsigned char` doesn't have a sign and therefore can hold larger positive values, typically twice as large. If neither `signed` nor `unsigned` is specified, characters are usually signed by default, but this is left up to the compiler.

Character values

The range of values that characters may represent is found in the standard header file *<climits>*. The size of a `char` is one byte. The size of a byte technically is implementation-defined, but it is rarely anything but eight bits. The size of the wchar_t type is also implementation-defined, but is typically two bytes.

Character literals

Character literals are enclosed by single quotes. For example:

```
char              c = 'A';
```

To specify literals for wide characters, you use the prefix L. For example:

```
wchar_t           c = L'A';
```

To allow special characters, such as newlines and single quotes, to be used within literals, C++ defines a number of *escape sequences*, each of which begins with a backslash. Table 1 presents these escape sequences. There is no limit to the number of hexadecimal digits that can appear after \x in a hexadecimal escape sequence. Octal escape sequences can be at most three digits.

Table 1. Character escape sequences

Escape sequence	Description
`\a`	Alert (system bell)
`\b`	Backspace
`\f`	Form feed
`\n`	Newline
`\r`	Carriage return
`\t`	Horizontal tab
`\v`	Vertical tab
`\\`	Backslash
`\'`	Single quote
`\"`	Double quote
`\?`	Question mark
\ooo	Octal number *ooo*
\\x*hhh*...	Hexadecimal number *hhh*...

short, int, long

Integers are of type `short`, `int`, or `long`. These types differ in size and the range of values they can represent. For example:

```
short          sval = 32767;
int            ival = 2147483647;
long           lval = 0x7fffffff;
```

Integers may be specified as either `signed` or `unsigned`. For example:

```
signed short      total;
unsigned short    flags = 0xf0f0;
```

Signed integers represent both positive and negative values, typically by sacrificing one bit to store a sign. Unsigned integers don't have a sign and therefore can hold larger positive values. If an integer is not specified as either `signed` or `unsigned`, it is signed by default.

Integer values

The range of values that each of the integer types may represent is found in the standard header file *<climits>*. The exact size of a `short`, `int`, or `long` is left up to the compiler, but is typically two, four, or four bytes respectively. Although the size of each type can vary, the compiler guarantees that the size of a `short` is less than or equal to the size of an `int`, and the size of an `int` is less than or equal to the size of a `long`.

Integer literals

Literals for integers have several forms, as shown in Table 2. If `U`, `u`, `L`, or `l` is not used as a suffix, the compiler assigns a type appropriate for the magnitude of the literal.

Table 2. Integer literals

Examples	Description

`12`
`-5`
	The most common form of integer literals.

`012`
`0377`
	Literals that begin with `0` are octal values (e.g., `012` is the octal literal for the decimal number `10`).

`0x2a`
`0xffff`
	Literals that begin with `0x` are hexadecimal values (e.g., `0x2a` is the hexadecimal literal for the decimal number `42`).

`256L`
`0x7fL`
	Literals with `L` (or `l`) in the suffix are treated as `long`.

`0x80U`
`0xffffUL`
	Literals with `U` (or `u`) in the suffix are treated as `unsigned`.

float, double, long double

Floating points are of type `float`, `double`, or `long double`. These types differ in size and in the range and precision of values they can represent. For example:

```
float             fval = 3.4e+38F;
double            dval = 1.7e+308;
```

Floating-point values

The range and precision of values that each of the floating-point types may represent is found in the standard header file *<cfloat>*. The exact size, range, and precision of a `float`, `double`, or `long double` is left up to the compiler, but is typically four, eight, or ten bytes respectively. Although the size of each type can vary, the compiler guarantees that the size of a `float` is less than or equal to the size of a `double`, and the size of a double is less than or equal to the size of a `long double`.

Floating-point literals

Literals for floating points can take on several forms, as shown in Table 3. If `F`, `f`, `L`, or `l` is not used as a suffix, the compiler assigns a type of `double`.

Table 3. Floating-point literals

Examples	Description

`1.2345`
`-57.0`
`0.4567`
	The most common form of literal floating points.

`1.992e+2`
`1.71e-25`
	Literals expressed in scientific notation.

`8.00275F`
`3.4e+38L`
	Literals with the suffix `F` (or `f`) are given the type `float`; literals with the suffix `L` (or `l`) are given the type `long double`.

void

The `void` type indicates the absence of a value. One use is in declaring functions that do not return a value. For example:

```
void sayHello( )
{
   cout << "Hello" << endl;
}
```

Another use is in declaring a pointer that can point to any type of data. For example:

```
int                  i = 200;
void                 *p = &i;
```

The variable `p` points to an `int`. Variables that are not pointers cannot be declared as `void`.

Compound Types

Arithmetic types are the building blocks for more complex types, called *compound types*. These include enumerations, arrays, strings, pointers, pointers to members, references, and the various class types of C++, as well as functions. Arithmetic types, enumerations, pointers, and pointers to members are collectively called *scalar types*.

Enumerations

An enumeration, specified by the keyword enum, is a set of integer constants associated with identifiers, called *enumerators*, that you define. In general, enumerations provide a way to use meaningful names where you might otherwise use integer constants, perhaps defined using the preprocessor directive #define. Enumerations are preferred over the preprocessor for this in C++ because they obey the language's rules of scope. The following defines an enumeration for the colors of the rainbow:

```
enum SpectrumColor
{
                  Red,   Orange, Yellow,
                  Green, Blue,   Indigo,
                  Violet
};
```

If you plan to instantiate variables to store values of an enumeration, you can give the enumeration a name (here, SpectrumColor); however, a name is not required. With this enumeration, you can write a loop to cycle through the colors of the rainbow, for example:

```
for SpectrumColor operator++(SpectrumColor &s, int dummy)
{
   return s = (s >= Violet) ? Red : SpectrumColor(s + 1);
}
```

Following are some additional points to keep in mind about enumerations:

- You can specify values for enumerators within an enumeration, which you can then use in place of integer constants.
- When you let the compiler assign values to enumerators, it assigns the next integer after the one assigned to the preceding enumerator.
- Values start at 0 if you do not provide a value for the first enumerator.

- You can use enumerators anywhere that you would use an `int`.
- You cannot assign arbitrary integers to a variable of an enumeration type.
- The size of integers for enumerations is no larger than the size of an `int`, unless a larger integer is needed for explicit values.

The following example illustrates these points:

```
enum
{
                 ASCII_NUL,      // 0
                 ASCII_SOH,      // 1
                 ASCII_STX,      // 2

                 ASCII_A = 65,   // 65
                 ASCII_B,        // 66

                 BufferSize = 8  // 8
};

char             buffer[BufferSize];
```

Arrays

Arrays contain a specific number of elements of a particular type. So that the compiler can reserve the required amount of space when the program is compiled, you must specify the type and number of elements that the array will contain when it is defined. The compiler must be able to determine this value when the program is compiled. For example:

```
enum
{
                  HandleCount = 100
};

int               handles[HandleCount];
```

Once an array has been defined, you use the identifier for the array along with an index to access specific elements within the array. The following sets each element in the previous array to an initial value of –1:

```
for (int i = 0; i < HandleCount; i++)
{
   handles[i] = -1;
}
```

Arrays are zero-indexed; that is, the first element is at index 0. This indexing scheme is indicative of the close relationship in C++ between pointers and arrays and the rules that the language defines for pointer arithmetic. In short, the assignment in the example above is equivalent to the following:

```
*(handles + i) = -1;
```

It is important to remember that no bounds-checking is performed for arrays.

Multidimensional arrays

C++ supports multidimensional arrays, which are arrays defined using more than one index, as follows:

```
enum
{
                  Size1 = 4,
                  Size2 = 4
};

double            matrix[Size1][Size2];
```

Arrays can be defined with more than two indices in a similar manner. Once a multidimensional array is defined, you use multiple indices to access a specific element, as follows:

```
for (int i = 0; i < Size1; i++)
   for (int j = 0; j < Size2; j++)
      matrix[i][j] = 0.0;
```

The relationship between pointers and arrays extends to multidimensional arrays as well. In short, the assignment in the example above is equivalent to the following:

```
*(*(matrix + i) + j) = 0.0;
```

Passing arrays to functions

When defining a function that has an array as a parameter, all but the first dimension must be specified for the parameter. This ensures that the proper pointer arithmetic can be performed. In the case of an array with a single dimension, this means that no dimension is required:

```
void f(int handles[])
{
   handle[0] = 0;
}
```

In the case of an array with two dimensions, for example, the second dimension must be specified:

```
void g(double matrix[][Size2])
{
   matrix[0][Size2 - 1] = 1.0;
}
```

You can also define equivalent functions that use pointer parameters:

```
void f(int *handles)
{
   handles[0] = 0;
}

void g(double (*matrix)[Size2])
{
   matrix[0][Size2 - 1] = 1.0;
}
```

The parentheses are needed in the second case so that the array is a multidimensional array of `double` values, not a one-dimensional array of `double` pointers.

Initializer lists for arrays

An initializer list for an array is a comma-delimited list of values by which to initialize the array's elements. The list is enclosed by braces ({}). Each value's type must be acceptable for the type of elements that the array has been declared to contain. For example:

```
enum SwitchState
{
                     On, Off
};

SwitchState          switches[] =
                     {
                        On, Off, On, Off
                     };
```

When you initialize an array with an initializer list, you may omit the array size in the declaration; enough space will be allocated for the array to accommodate the values specified. If you provide a size but specify values for fewer elements than the size indicates, the missing elements are default-initialized. The rules for default initialization are complicated; you should not rely on them.

Initializer lists can also be used to initialize arrays that are multidimensional. The rules are essentially the same as for arrays of one dimension, except that an initializer list for a multidimensional array uses nested braces to align its values in a manner consistent with the size of each dimension.

```
char                 tictactoe[3][3] =
                     {
                        {'_', '_', '_',},
                        {'_', '_', '_',},
                        {'_', '_', '_',}
                     };
```

Strings

Character (C-style) strings are arrays of characters terminated with a null character (\0). The characters of the string are of type char, or type wchar_t for wide-character strings. For example:

```
enum
{
                     NameLength = 81
};
```

```
    char            name[NameLength];
    wchar_t         wide[NameLength];
```

You must allocate one extra character for the null terminator in arrays of characters to be used for strings. Functions that return a string's length, such as `strlen` (from the C++ Standard Library), do not include a string's null terminator in the length returned. Wide-character versions of standard facilities typically have the prefix `w` or use `wcs` instead of `str` (e.g., `wostream`, `wcsncpy`, etc.).

NOTE

Although many facilities in the C++ Standard Library work with character (C-style) strings, the preferred way to work with strings in C++ is to use the `string` class from the C++ Standard Library. The wide-character version is `wstring`.

String literals

String literals are enclosed in double quotes. For example:

```
    char            name[] = "Margot";
```

Long string literals can be broken into quoted strings separated by whitespace for style, when needed. For example:

```
    char            s[] = "This string is "
                          "on two lines.";
```

To specify literals for wide-character strings, you use the prefix `L`. For example:

```
    wchar_t         wide[] = L"Margot";
```

The compiler allocates enough space for a string, including its null terminator. An empty string (`""`) actually has space reserved for one character: the null terminator. The storage for a string literal is guaranteed to exist for the life of the program, even for a string literal defined locally within a block. The type of a string literal is an array of `const char` or `wchar_t` elements of static duration.

Pointers

For any type *T*, there is a corresponding type *pointer to T* for variables that contain addresses in memory of where data of type *T* resides. *T* is the *base type* of a pointer to *T*. Pointers are declared by placing an asterisk (*) before the variable name in a declaration (see "Declaring Variables"). In the following example, `i` is an `int` while `*iptr` is a pointer to `i`:

```
int                  i = 20;
int                  *iptr = &i;
```

Normally you can set a pointer of a specific type only to the address of data of that same type, as just shown. However, in the case of a pointer to a class, the pointer can also be assigned the address of an object of some type derived from that class. This is essential for polymorphism (see "Virtual Member Functions"). For example, if `Circle` were derived from `Shape` (see "Inheritance"), we could do the following:

```
Circle               c;
Shape                *s = &c;
```

Pointer dereferencing

Dereferencing a pointer yields what the pointer points to. To dereference a pointer, you precede it with an asterisk in an expression, as shown in the commented lines below:

```
int                  i = 20;
int                  *iptr = &i;
int                  j;
int                  k = 50;

j = *iptr;           // This sets j to i.
*iptr = k;           // This sets i to k;
```

Pointer arithmetic

Pointers in expressions are evaluated using the rules of *pointer arithmetic*. When an operator for addition, subtraction, increment, or decrement is applied to a pointer *p* of type *T*, *p* is treated as an array of type *T*. As a result, *p* + *n* points to the *n*th successive element in the array, and *p* – *n* points to the *n*th previous element. If *n* is 0, *p* + *n* points to the first element in the array. So, if *T* were a type with a size of 24 bytes, *p* += 2 would actually increase the address stored in *p* by 48 bytes.

Pointer arithmetic illustrates the close relationship between pointers and arrays in C++. However, pointers and arrays do have a fundamental difference: whereas a pointer can be modified to point to something else, an array cannot be changed to point away from the data it was created to reference.

Void pointers

Pointers of type `void` are permitted to point to data of any type. For example:

```
Circle          c(2.0);
void            *p;

p = &c;         // c is a circle.
```

When assigning a void pointer to a pointer of some other type, an explicit cast is required. For example:

```
Circle          *c;

c = static_cast<Circle *>(p);
```

Void pointers cannot be dereferenced or used with pointer arithmetic.

Null pointers

Pointers of any type can be assigned the value 0, which indicates that the pointer points to nothing at all. A pointer with the value 0 is called a *null pointer*. You should never dereference a null pointer.

Function pointers

A function pointer is a pointer that points to a function. Its type is related to the signature of the function to which it points. For example, the following defines a function named addOne, then defines inc as a pointer to a function that takes a reference to an int as a parameter and returns void. inc is then set to addOne, which has that same signature:

```
void addOne(int &x)
{
   x += 1;
}

void                (*inc)(int &x) = addOne;
```

The last line could also be written as shown below (using the address-of operator, &, before addOne):

```
void                (*inc)(int &x) = &addOne;
```

Parentheses are needed around inc so that the asterisk is associated with the name of the pointer, not the type. Once a function pointer points to a function, it can be used to invoke the function, as follows:

```
int                 a = 10;

inc(a);             // This adds 1 to a.
```

The last line could also be written as shown below (using the indirection operator, *, before the pointer):

```
(*inc)(a);
```

Pointers to Members

Pointers to members are like alternative names for class members (see "Classes, Structs, and Unions"). For example, assume that class X has a member of type int called data:

```
int                 X::*p = &X::data;
X                   object;
X                   *objptr = new X;

int                 i = object.*p;
int                 j = objptr->*p;
```

This sets i to the value of data in object, and j to the value of data in the object addressed by objptr.

References

References are used to provide alternative names for variables. They are declared by placing an ampersand (&) before the variable name in a declaration. For example:

```
int                 i = 20;
int                 &r = i;
```

Because a reference always has to refer to something, references must be initialized where they are defined. Therefore, a reasonable way to think of a reference is as a constant pointer. Once initialized, the reference itself cannot be made to refer to anything else; however, the variable or object to which it refers can be modified. Operations applied to the reference affect the variable or object to which the reference refers. For example:

```
int                 i = 20;
int                 &r = i;

r++;                // This increments i.
```

Normally you can set a reference of a specific type to a variable of that same type, as just shown. However, in the case of a reference to a class, the reference can also refer to an object of some type derived from that class. Therefore, like pointers, references support polymorphic behavior (see "Virtual Member Functions"). For example, if Circle were derived from Shape (see "Inheritance"), you could do the following:

```
Circle              c;
Shape               &s = c;
```

Reference parameters

A common use of references is with parameters for functions. References allow changes to parameters to be reflected in the caller's environment. For example:

```
void xchg(int &x, int &y)
{
   int t = x;
   x = y;
   y = t;
}
```

Using the definition above, you could swap two integers `a` and `b` by doing the following:

```
xchg(a, b);
```

If `x` and `y` were not references in the definition of `xchg`, the contents of `x` and `y` would be swapped within the function, but the contents of `a` and `b` would be unchanged when the function returned.

References as l-values

References are also often used in C++ as return values for functions. This allows the return value of a function to be used as an *l-value*, which is a value that can appear on the left side of an assignment.

Class Types

The class types of C++ are classes, structs, and unions (see "Classes, Structs, and Unions").

Type Conversions and Definitions

In C++ you can convert a value of one type into a value of another type. Such an action is called a *type conversion*. You can also define your own type names using the `typedef` keyword.

Type Conversions

Type conversions are performed when you use a cast explicitly (see "Casts and Runtime Type Information"), and at times implicitly by the compiler. For example, the compiler converts a type implicitly when the types in a binary operation are not the same. A compilation error occurs if no conversion is possible.

Implicit conversions

Implicit conversions occur between C++'s arithmetic types, between certain pointer types (see "Pointers"), and between user-defined types and others. The implicit conversion of arithmetic types and pointer types in binary operations is carried out by converting the smaller or less precise type to the larger or more precise one. Booleans, characters, and integers smaller than an `int` are first converted to an `int` using *integral promotion*. When an integer and a floating point appear in the same operation, the integer is converted to the floating-point type.

Preservation of values

The implicit conversion of arithmetic types is performed in such a way as to preserve the original values of the entities being converted whenever possible. However, there are many situations in which surprising results can occur. For example, a compiler may not warn about conversions from wider or more precise types to smaller or less precise ones (e.g., from `long` to `short`, or `double` to `float`), in which a wider value may not be representable in the smaller type. In addition, the conversion from an unsigned type to a signed one can result in a loss of information.

User-defined conversions

You can specify explicit conversions for user-defined types by defining user-defined conversion operators (see "Overloading Operators"). For example, the following user-defined conversion operator, `operator double`, converts an `Account` object to a `double`:

```
class Account
{
public:
```

```
                    Account(double b)
                    {
                       balance = b;
                    }

                    operator double( )
                    {
                       return balance;
                    }
   ...
   private:
      double        balance;
   };
```

This user-defined conversion operator allows you to use a value of type `Account` where you might otherwise use a `double`:

```
   Account          account(100.0);
   double           balance = account;
```

When C++ sees the assignment of an `Account` value to a `double` variable, it invokes `operator double` to perform the conversion.

Converting constructors

A constructor that has a single parameter and is not declared using `explicit` can be used by the compiler to perform implicit conversions between the type of the parameter and the class type. For example:

```
   class Account
   {
   public:
                    Account(double b)
                    {
                       balance = b;
                    }
   ...
   private:
      double        balance;
   };
```

The constructor in this class allows you to do the following, for example:

```
   Account          account = 100.0;
```

Type Definitions

Frequently it is useful to provide an alternative name for types that have long or otherwise unwieldy names. This is accomplished using `typedef`.

To define a new name for a type, you use the keyword `typedef` followed by the old type, then the new type. The following example defines `uint32` to mean `unsigned long`:

```
typedef unsigned long uint32;

uint32          value32bit;
```

This illustrates using `typedef` to define your own sized-integer type (e.g., `int8`, `int16`, `int32`, etc.). Some compilers define `__int8`, `__int16`, and so forth; `typedef` provides a way to use types like these with any compiler. Another common use of `typedef` is in providing alternative names for parameterized types, which tend to be long, when working with the Standard Template Library. For example:

```
typedef map<int, string> IntStringMap;

IntStringMap      m;
```

Lexical Elements

At the most fundamental level, a C++ program consists of individual lexical elements called *tokens*. Tokens are delineated by whitespace (spaces, newlines, tabs, etc.), or can be formed once the start of another token is recognized, as shown below:

```
ival+3
```

This stream actually consists of three tokens even though there is no whitespace. The tokens are `ival`, `+`, and `3`. In the absence of whitespace, the compiler forms tokens by consuming the longest possible token as it scans from left to right.

Tokens are passed to the parser, which determines if a stream of tokens has the correct syntax. Tokens together form more complex semantic constructs, such as declarations, expressions, and statements that affect the flow of execution.

Comments

Comments are notes written in the source code for developers; they are ignored completely by the compiler. The preprocessor converts each comment to a single space before the compiler ever gets the chance to see it.

A comment is any block of text enclosed between `/*` and `*/`, or following `//` on a single line. Comments of the first form cannot be nested within one another. They usually span multiple lines. For example:

```
/* This comment has more than one line.
   Here is another part of the comment.*/
```

Comments of the second form are useful for short explanations that do not occupy more than a single line. For example:

```
z = MIN(x, y);    // z is the smallest.
```

Once a single-line comment begins, it occupies the remainder of the line. There is no way to end the comment before this.

Identifiers

Identifiers in C++ are sequences of characters that are used for names of variables, functions, parameters, types, labels, namespaces, and preprocessor macros. Identifiers may consist of letters, digits, and underscores, but they must not begin with a digit. For example, the following are all legal C++ identifiers:

```
i       addressBook     Mgr       item_count
ptr2    NAME_LENGTH     class_    showWindow
```

The following rules apply to identifiers:

- Identifiers are case-sensitive, and they must not be one of the C++ reserved words (see "Reserved Words").
- Identifiers that begin with an underscore are reserved for implementations of the language.
- Although C++ imposes no limit on the size of identifiers, your compiler and linker will have size limits that you should consider in practice.

NOTE

There is no one stylistic convention for identifiers upon which everyone agrees. One common convention, however, is to use lowercase characters to begin names for local variables, data members, and functions. Uppercase characters are then used to begin the names of types, namespaces, and global variables. Names processed by the preprocessor are written entirely in uppercase. Names of parameters in macros are written entirely in lowercase.

Reserved Words

C++ defines a number of keywords and alternative tokens, which are sequences of characters that have special meaning in the language. These are reserved words and cannot be used for identifiers. The reserved words of C++ are listed below:

```
and                 and_eq              asm
auto                bitand              bitor
bool                break               case
catch               char                class
compl               const               const_cast
continue            default             delete
do                  double              dynamic_cast
else                enum                explicit
export              extern              false
float               for                 friend
goto                if                  inline
```

```
int                long               mutable
namespace          new                not
not_eq             operator           or
or_eq              private            protected
public             register           reinterpret_cast
return             short              signed
sizeof             static             static_cast
struct             switch             template
this               throw              true
try                typedef            typeid
typename           union              unsigned
using              virtual            void
volatile           wchar_t            while
xor                xor_eq
```

Literals

Literals are lexical elements that represent explicit values in a program. C++ defines many types of literals. Each is described under its respective type in "Fundamental Types."

Operators

An operator is used to perform a specific operation on a set of operands in an expression. Operators in C++ work with anywhere from one to three operands, depending on the operator.

Associativity

Operators may associate to the left or right. For example, assignment operators (=, +=, <<=, etc.) associate to the right. Therefore, the following:

```
i = j = k
```

actually implies:

```
i = (j = k)
```

On the other hand, the operator for addition (+) associates to the left. Therefore, the following:

```
i + j + k
```

actually implies:

```
    (i + j) + k
```

Precedence

Operators also have an order, or *precedence*, by which expressions that contain them are evaluated. Expressions containing operators with a higher precedence are evaluated before those containing operators with a lower precedence.

You can use parentheses around an expression to force grouping. Even when not essential, it's best to use parentheses in expressions to document your intentions. The number of operators in C++ often makes their precedence difficult to remember.

Table 4 lists the operators of C++ from highest precedence to lowest and describes how each operator associates. Each section contains operators of equal precedence. The table also describes the behavior of each operator when used with the intrinsic types of C++. For most operators, C++ lets you define additional behaviors for your own types (see "Overloading Operators").

Table 4. Operators

Operator	Description	Associates
`::`	Scope resolution	No
`[]`	Array subscript	Left
`.`	Member selection	Left
`->`	Member selection	Left
`()`	Function call	Left
`()`	Value construction	No
`++`	Postfix increment	No
`--`	Postfix decrement	No
`typeid`	Type information	No
`*_cast`	C++ cast	No

Table 4. Operators (continued)

Operator	Description	Associates
`sizeof`	Size information	No
`++`	Prefix increment	No
`--`	Prefix decrement	No
`~`	Bitwise NOT	No
`!`	Logical NOT	No
`-`	Unary minus	No
`+`	Unary plus	No
`&`	Address-of	No
`*`	Indirection	No
`new`	Allocate	No
`new[]`	Allocate	No
`delete`	Deallocate	No
`delete[]`	Deallocate	No
`()`	C-style cast	Right
`.*`	Pointer-to-member selection	Left
`->*`	Pointer-to-member selection	Left
`*`	Multiply	Left
`/`	Divide	Left
`%`	Modulo (remainder)	Left
`+`	Add	Left
`-`	Subtract	Left
`<<`	Shift left	Left
`>>`	Shift right	Left
`<`	Less than	Left
`<=`	Less than or equal to	Left
`>`	Greater than	Left
`>=`	Greater than or equal to	Left
`==`	Equal to	Left
`!=`	Not equal to	Left

Table 4. Operators (continued)

Operator	Description	Associates
&	Bitwise AND	Left
^	Bitwise XOR	Left
|	Bitwise OR	Left
&&	Logical AND	Left
||	Logical OR	Left
?:	Conditional expression	Right
=	Simple assignment	Right
*=	Multiply and assign	Right
/=	Divide and assign	Right
%=	Modulo and assign	Right
+=	Add and assign	Right
-=	Subtract and assign	Right
<<=	Shift left and assign	Right
>>=	Shift right and assign	Right
&=	AND and assign	Right
^=	XOR and assign	Right
|=	OR and assign	Right
throw	Throw exception	Right
,	Sequence	Left

Additional information about the behaviors of the operators in C++ is summarized in the following sections.

Scope resolution (::)

The scope resolution operator is used to specify a scope (see "Scope"). For example, the following invokes a static member function of a class called `Dialog`:

```
    dialog = Dialog::createDialog( );
```

The scope operator can also be used without a scope name to specify file (global) scope. For example:

```
::serialize(i);
```

This ensures that the global function `serialize` is invoked, even if `serialize` has been declared within the local scope.

Array subscript ([ ])

The array subscript operator is used to access individual elements of arrays or memory referenced by pointers. For example:

```
tmp = table[0];
```

This assigns the first element in an array called `table` to `tmp`. The expression between the brackets indicates the element.

Member selection (. and ->)

Member selection operators are used to specify members of objects (see "Classes, Structs, and Unions"). You use the dot form with objects and the arrow form with pointers to objects. For example:

```
object.f( );
```

This invokes member function `f` of an object called `object`. The following illustrates the arrow form:

```
objptr->f( );
```

This invokes member function `f` for an object that is addressed by the pointer `objptr`.

Function call (())

The function call operator, which is `( )`, is used to invoke a function. For example:

```
f(a, b);
```

This invokes a function called `f` with two arguments, `a` and `b`.

Value construction (())

The value construction operator, which is also ( ), is used to create an instance of a type. For example:

```
g(Circle(5.0));
```

This constructs a temporary object that is an instance of the `Circle` class, which is passed to g.

Postfix increment and decrement (++, --)

The postfix increment and decrement operators increment or decrement an operand, but the value of the operand within its expression is the value prior to modification. For example:

```
void count( )
{
   static int     i = 0;

   if (i++ == 0)
   {
      // This is the first time called.
   }
}
```

The value of `i` prior to being incremented is tested for equality with 0. Because `i` is initialized to 0, the test is true during the first invocation of the function.

typeid

The `typeid` operator gets runtime type information for an operand. See "Casts and Runtime Type Information" for a complete description of this operator.

C++ cast

Type cast operators specific to C++ are `dynamic_cast`, `static_cast`, `const_cast`, and `reinterpret_cast`. See "Casts and Runtime Type Information" for a complete description of these operators.

sizeof

The `sizeof` operator gets the size of its operand. For example:

```
size_t          s = sizeof(c);
```

This initializes `s` to the size of `c`. The operand may be an expression or type. The result is an integer of type `size_t`.

Prefix increment and decrement (++, −−)

The prefix increment and decrement operators increment or decrement an operand. The value of the operand within its expression is the value after modification. For example:

```
void count( )
{
   static int    i = 0;

   if (++i == 1)
   {
      // This is the first time called.
   }
}
```

The value of `i` after being incremented is tested for equality with 1. Because `i` is initialized to 0, the test is true during the first invocation of the function.

Bitwise NOT (~)

The bitwise NOT operator computes the bitwise complement of its operand. For example:

```
unsigned char   bits = 0x0;
bits = ~bits;
```

This assigns 0xFF back into `bits`, assuming a character is eight bits. The operand must be one of the Boolean, character, or integer types of C++.

Logical NOT (!)

The logical NOT operator reverses the truth of its operand; it yields false if its operand is true (nonzero) and true if its operand is false. For example:

```
bool             done = false;

while (!done)
{
   // Set done to true when finished.
}
```

This loop is repeated until something in the loop sets done to true. The result of the logical NOT operator is a bool.

Unary minus and plus (–, +)

The unary minus and plus operators compute the negative and positive values their operands. For example:

```
i = -125;
j = +273;
```

Because the unary plus operator simply returns the value of its operand (promoted to an int), it is seldom used.

Address-of (&)

The address-of operator gets the address at which its operand resides in memory. For example:

```
Circle           c;
Circle           *p = &c;
```

This assigns the address of c to the Circle pointer p. The address is a pointer derived from the type of the operand.

Indirection (*)

The indirection operator dereferences a pointer and gets the value that it addresses. For example:

```
int              i;
int              *p = new int;

*p = 5;
i = *p;
```

This assigns the value 5 to i. The type of the result is the type from which the pointer is derived. The operand must be a pointer.

Allocate and deallocate

The C++ memory management operators are new, new[], delete, and delete[]. They allocate and reclaim memory on the heap. See "Memory Management" for a complete description of these operators.

C-style cast (())

The C-style cast operator converts the type of its operand to a new type (see "C-Style Casts"). For example:

```
void                 *p = new int;
*p = 10;
int                  *q = (int *)p;
```

This casts p from a void pointer to an int pointer. No run-time checking is performed to ensure that the cast is legal.

Pointer-to-member selection (.* and –>*)

The .* and ->* operators access a class member via a pointer to the member. For example:

```
int                    X::*p = &X::data;
X                      object;
X                      *objptr = new X;

int                    i = object.*p;
int                    j = objptr->*p;
```

This sets i to the value of data in object, and j to the value of data in the object addressed by objptr. You use the dot form with objects and the arrow form with pointers to objects.

Arithmetic (*, /, %, +, –)

Arithmetic operators perform multiplication (*), division (/), modulus (%), addition (+), and subtraction (–) using two operands. For example:

```
if (x % 2 == 0)
{
   // The integer x is an even number.
}
```

Either condition can be true for the block containing the comment to be executed.

For both operators, if a result can be determined from the first operand alone, the second operand is not evaluated. When these operators are overloaded, both operands are always evaluated.

Conditional expression (?:)

The conditional expression operator uses the value of one operand to determine whether to evaluate the second or third operand. For example:

```
i = (p != NULL) ? *p : -1;
```

If the first operand is true, the result is the second operand; otherwise, the result is the third. The first operand appears before the question mark (?); the second and third operands are separated by a colon (:).

Simple and compound assignments (=, *=, /=, %=, +=, −=, <<=, >>=, &=, |=, ^=)

Assignment operators assign the value of one operand to another. For example:

```
i = (j + 10) * 5;
```

This is the simplest form of assignment; the second operand is simply evaluated and stored into the first. The other assignment operators perform compound assignments. For example:

```
i += 5;
```

This adds 5 to `i` and assigns the result back to `i`. Therefore, it has the same effect as the following but avoids the need for `i` to be evaluated twice:

```
i = i + 5;
```

After any assignment, the value of the expression is the value that was assigned. This allows assignments to be chained together, as follows:

```
a = b = c;
```

Exception (throw)

The throw operator is used to throw an exception. See "Exception Handling" for a complete description of this operator.

Sequence (,)

The sequence operator, which is a comma, evaluates two operands from left to right. The value of the expression becomes the value of the last operand. For example:

```
for (i = 0, j = 10; i < 10; i++, j--)
{
   // Increase i while making j smaller.
}
```

In this case, the result of i=0, j=10 is 10. However, both assignments are performed; both variables are initialized.

Expressions

An expression is something that yields a value. Nearly every type of statement uses an expression in some way. For example, the declaration below uses an expression for its initializer:

```
int                  t = (100 + 50) / 2;
```

The simplest expressions in C++ are just literals or variables by themselves. For example:

```
1.23     false    "string"    total
```

More interesting expressions are formed by combining literals, variables, and the return values of functions with various operators to produce new values. These can then be used in expressions themselves. For example, the following are all C++ expressions:

```
i->getValue( ) + 10
p * pow(1.0 + rate,(double)mos))
new char[20]
sizeof(int) + sizeof(double) + 1
```

Scope

A name can be used only within certain regions of a program. These regions define its *scope*. The scope of a name is based on where, and to some extent how, you declare it. Most names have one of four scopes. Labels and prototype parameters have their own special scopes.

Local Scope

A name has local scope when it is declared inside of a block. A block is a compound statement that begins with a left brace ({) and ends with a right brace (}). For example:

```
void f( )
{
   int            i = 10;
   ...
}
```

In this example, `i` has local scope. A name with local scope is visible only within its block.

Class Scope

A name has class scope when it is declared within the confines of a class and does not have local scope. For example:

```
class Event
{
public:
   enum Type
   {
                  keyDown,
                  ...
   };
   ...
   Type           getType( ) const
                  {
                      return type;
                  }
   ...
private:
```

Declaring Variables

Declarations for variables introduce names that refer to data. They contain the following, in order: an optional storage class, optional qualifiers, a type, and a comma-delimited list of one or more names to declare. For example:

```
int                 i, j, k;
char                buffer[80];
static int          counter, a;
volatile float      x;
```

Data members of classes are declared in a similar manner (see "Classes, Structs, and Unions"); however, they can only have the storage classes `static` (see "Static data members") and `mutable` (see "Mutable data members").

Declarations for variables may appear anywhere within a block, not just at the start. This makes code like the following common in C++:

```
void spin(int n)
{
   cout << "Spinning" << endl;

   for (int i = 0; i < n; i++)
      ;
}
```

In this example, the variable `i` is declared within the `for` loop, as opposed to at the start of the function in which the loop appears.

Pointer variables

Declarations for pointers follow the same rules as for other types of variables, except you must be sure to precede each name with an asterisk (*). For example:

```
int                 *p, *q, *r;
```

Special situations arise when the qualifier `const` is used in the declaration of pointer variables (see "Qualifiers").

Function definitions

Declarations for functions are called *prototypes*. They do not define a function; they simply inform the compiler of your intention to define and use it. To define a function, you specify a body for it, as follows:

```
void xchg(int &x, int &y)
{
   int t = x;
   x = y;
   y = t;
}
```

This function has a return type of `void`. If the return type is anything other than `void`, the function must use a `return` statement (see "Jump Statements") to return a value suitable for the function's return type. Functions that return `void` can use a `return` statement without a value.

Default arguments

Default arguments can be specified for the parameters of functions. You do this by setting a parameter equal to its default value in the function declaration, as shown below:

```
void            isTempOK(const int t,
                   const int low = 20,
                   const int high = 50);
```

A default argument is used for a parameter when nothing is specified for it in an invocation of the function. For example, assuming a temperature declared as `temp`, the following uses the default arguments 20 for `low` and 50 for `high`:

```
if (!isTempOK(temp))
{
   // Do something if too low or high.
}
```

If a function is declared with default arguments, the parameters with defaults must appear last in the parameter list. Values are assigned to parameters from left to right. Remaining parameters are then assigned their default values. Therefore,

Forward Declarations

You can declare a class without providing a definition for it. You do that using what is called a *forward declaration*, which declares the class name without specifying any other details about the class. For example:

```
class Account;
```

This informs the compiler that you plan to define the class later but are going to use its name now without referring to any of the class's members. For example, forward declarations are needed when two classes refer to each other:

```
class Account;

class Bank
{
...
private:
   Account       *accounts;
};

class Account
{
...
private:
   Bank          *bank;
};
```

Alternatively, you could define `Account` first and provide a forward declaration for `Bank`.

Structs

Structs are functionally identical to classes except that the default access level for their members is public, not private. To define a struct, you use the keyword `struct` in place of the keyword `class`.

Unions

Unions are similar to classes; however, they can hold a value for only one data member at a time. As a result, a union occupies only as much space as its largest data member requires. Other differences between unions and classes are:

- The default access level for unions is public; the default access level for classes is private.
- Unions cannot have member functions that are declared using the keyword `virtual`.
- Unions cannot inherit from anything, nor can anything inherit from them.
- The members of unions cannot be objects that define constructors or destructors, or that overload the assignment operator.

Unions can be anonymous (unnamed). This form is used when nesting a union inside of a struct or class that contains an extra data member to indicate what the union contains. For example:

```
struct AccountInfo
{
   enum
   {
                NameInfo,
                BalanceInfo
   };

   int          type;

   union
   {
      char      name[20];
      double    balance;
   };
};
```

When setting a value in the union, you record how the union is being used. For example:

```
AccountInfo     info;

info.type = AccountInfo::BalanceInfo;
info.balance = 100.0;
```

Whenever you need to access the data member, you check what the union contains. For example:

```
if (info.type == AccountInfo::BalanceInfo)
{
   // Use the balance.
}
```

Inheritance

When you derive one class from another, the derived class inherits the data members and member functions that the other class defines (subject to access controls) while adding its own. Aside from the benefits that inheritance offers stemming from the reuse of functionality provided by the base class, inheritance is fundamental to supporting polymorphism (see "Virtual Member Functions"), an essential part of object-oriented programming. Consider the version of `Account` below:

```
class Account
{
public:
                  Account(double b);

   void           deposit(double amt);
   void           withdraw(double amt);
   double         getBalance( ) const;

protected:
   double         balance;
};
```

To derive a new class called `BankAccount` from `Account`, you do the following:

```
class BankAccount : public Account
{
public:
                  BankAccount(double r);

   void           addInterest( );
   void           chargeFee(double c);
private:
   double          interestRate;
};
```

Account is called the *base class* (or *superclass*). BankAccount is called the *derived class* (or *subclass*). A BankAccount object gives you the functionality of both BankAccount and Account:

```
BankAccount       bankAccount(2.25);

bankAccount.deposit(50.0);
bankAccount.addInterest( );
```

Access to members of the base class depends on two criteria: the access level of the member in the base class (see "Access Levels for Members") and the access level for inheritance (see "Access Levels for Inheritance").

Constructors and Inheritance

Whenever you instantiate an object of a class derived from another class, multiple constructors are called so that each class in the derivation chain can initialize itself (see "Constructors").

Order of construction

The constructor for each class in the derivation chain is called beginning with the base class at the top of the derivation chain and ending with the most derived class.

Base class initializers

Base class initializers stipulate the data to pass to the constructors of base classes. They are specified with a constructor's definition. Base class initializers are placed in a comma-delimited list between the constructor's signature and its body. The list begins with a colon (:). For example:

```
class BankAccount : public Account
{
public:
                  BankAccount(double r) :
                        {
                           balance -= amt;
                        }

   protected:
      double            balance;
   };
```

The member function in the derived class is declared like any other member function, although it is common to declare the member function using the keyword `virtual` in the derived class as well for purposes of documentation. For example:

```
   class BankAccount : public Account
   {
   public:
   ...
      virtual void      withdraw(double amt)
                        {
                           if (balance - amt < 0.0)
                           {
                              // Do nothing.
                           }
                           else
                           {
                              // Balance OK.
                              balance -= amt;
                           }
                        }
   ...
   };
```

When you invoke a virtual member function via a base class pointer or reference to an object of a derived class, the member function of the derived class is called instead of the member function of the base class. For example:

```
   BankAccount          bankAccount(2.25);
   Account              *aptr = &bankAccount;

   aptr->withdraw(50.0);
```

The last line in this example calls `withdraw` of the `BankAccount` class. To determine which member function to call, C++ uses *polymorphism*, or *dynamic binding*. This allows the

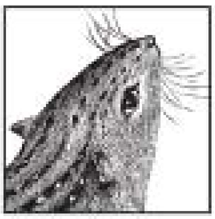
Index

We'd like to hear your suggestions for improving our indexes. Send email to *index@oreilly.com*.

C++ Programming

O'REILLY®

C++ Pocket Reference



C++ is a complex language with many subtle facets. The *C++ Pocket Reference* allows C++ programmers to quickly look up usage and syntax for the most commonly used features of the language. As much information as possible has been crammed onto its pages, and its small size makes it easy to take anywhere. The *C++ Pocket Reference* covers:

- C++ statements and preprocessor directives
- C++ namespaces and scope
- Template programming and exception handling
- Classes and inheritance
- C++ types and type conversion, including C++ casts

In addition to serving as a ready-reference for C++ programmers, the *C++ Pocket Reference* is useful to Java and C programmers who are making the transition to C++ or who find themselves occasionally programming in C++. The three languages are often confusingly similar. This book enables programmers familiar with C or Java to quickly come up to speed on how a particular construct or concept is implemented in C++.

The *C++ Pocket Reference* is one of the most concise, portable quick references to the C++ language available.

www.oreilly.com

US $9.95 **CAN $14.95**
ISBN-10: 0-596-00496-6
ISBN-13: 978-0-596-00496-5

50995

9 780596 004965