C++ Quick Syntax Reference

Mikael Olsson

*For your convenience Apress has placed some of the front
matter material after the index. Please use the Bookmarks
and Contents at a Glance links to access them.*

friendsof

apress®

Contents at a Glance

Introduction

C++ is a general purpose multi-paradigm programming language. It is an extension of the C language and as such most C code can easily be made to compile in C++. Some of the major additions to C include object-orientated programming, operator overloading, multiple inheritance and exception handling.

The development of C++ began in 1979, seven years after C first made its appearance. Despite being what many consider legacy languages, C and C++ are still the most widely used languages in the software industry. They are used in creating everything from operating systems and embedded software to desktop applications, games and so on.

Compared with newer languages, C++ applications are often more complex and take longer to develop. In return, C++ gives the programmer a tremendous amount of control in that the language provides both high-level and low-level abstractions from the hardware. It is also designed to give the programmer a lot of freedom by supporting many different programming styles or paradigms, such as procedural, object-oriented or generic programming.

The compiler used in this book is the Microsoft C++ compiler. Some other common ones include Borland, Intel and GNU C++ compilers. Despite C++ being standardized in 1998, these compilers still support slightly different features. Therefore, when something applies specifically to the Microsoft compiler this will be pointed out.

CHAPTER 1

■ ■ ■

Hello World

Choosing an IDE

To begin developing in C++ you should download and install an Integrated Development Environment (IDE) that supports C++. A good choice is Microsoft's own Visual Studio.[1] If you do not have Visual Studio but would like to try out the examples in this book in a similar environment you can download Visual Studio Express[2] from Microsoft's website. This is a lightweight version of Visual Studio that is available for free. Alternatively, you can develop using a simple text editor – such as Notepad – although this is less convenient than using an IDE. If you choose to do so, just create an empty document with a .cpp file extension and open it in the editor of your choice.

Creating a project

After installing Visual Studio or Visual Studio Express, go ahead and launch the program. You then need to create a project, which will manage the C++ source files and other resources. Go to File ➤ New ➤ Project in Visual Studio, or File ➤ New Project in Visual Studio Express, to display the New Project window. From there select the Visual C++ template type in the left frame. Then select the Win32 Console Application template in the right frame. At the bottom of the window you can configure the name and location of the project. When you are finished, click the OK button and another dialog box will appear titled Win32 Application Wizard. Click next and a couple of application settings will be displayed. Leave the application type as Console application and check the Empty project checkbox. Then click Finish to let the wizard create your empty project.

Adding a source file

You have now created a C++ project. In the Solution Explorer pane (View ➤ Solution Explorer) you can see that the project consists of three empty folders: Header Files, Resource Files and Source Files. Right click on the Source Files folder and select Add ➤ New Item. From the Add New Item dialog box choose the C++ File (.cpp) template. Give this source file the name "MyApp" and click the Add button. An empty cpp file will now be added to your project and also opened for you.

---

[1] http://www.microsoft.com/visualstudio
[2] http://www.microsoft.com/express

1

Hello world

The first thing to add to the source file is the main function. This is the entry point of the program, and the code inside of the curly brackets is what will be executed when the program runs. The brackets, along with their content, is referred to as a code block, or just a block.

```
int main() {}
```

The first application will simply output the text "Hello World" to the screen. Before this can be done the iostream header needs to be included. This header provides input and output functionality for the program, and is one of the standard libraries that come with all C++ compilers. What the #include directive does is effectively to replace the line with everything in the specified header before the file is compiled into an executable.

```
#include <iostream>
int main() {}
```

With iostream included you gain access to several new functions. These are all located in the standard namespace called std, which you can examine by using a double colon, also called the scope resolution operator (::). After typing this in Visual Studio, the IntelliSense window will automatically open, displaying what the namespace contains. Among the members you find the cout stream, which is the standard output stream in C++ that will be used to print text to a console window. It uses two less-than signs known as the insertion operator (<<) to indicate what to output. The string can then be specified, delimited by double quotes, and followed by a semicolon. The semicolon is used in C++ to mark the end of all statements.

```
#include <iostream>

int main()
{
  std::cout << "Hello World";
}
```

Using namespace

To make things a bit easier you can add a line specifying that the code file uses the standard namespace. You then no longer have to prefix cout with the namespace (std::) since it is now used by default.

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

int main()
{
  cout << "Hello World";
}
```

2

■ ■ ■

Compile and Run

Visual Studio compilation

Continuing from the last chapter, the Hello World program is now complete and ready to be compiled and run. You can do this by going to the Debug menu and clicking on Start Without Debugging (Ctrl + F5). Visual Studio then compiles and runs the application which displays the text in a console window.

If you select Start Debugging (F5) from the Debug menu instead, the console window displaying Hello World will close as soon as the main function is finished. To prevent this you can add a call to the `cin::get` function at the end of main. This function, belonging to the console input stream, will read input from the keyboard until the return key is pressed.

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

int main()
{
  cout << "Hello World";
  cin.get();
}
```

Console compilation

As an alternative to using an IDE you can also compile source files from the command line as long as you have a C++ compiler.[1] For example, on a Linux machine you can use the GNU C++ compiler, which is available on virtually all Unix systems. You type the compiler name "g++" and give it the input and output filenames as arguments. It then produces an executable, which when run gives the same result as one compiled under Windows.

```
g++ MyApp.cpp -o MyApp.exe
./MyApp.exe
Hello World
```

---

[1] http://www2.research.att.com/~bs/compilers.html

3

Comments

C++ has two kinds of comment notations – single-line and multi-line. These are used to insert notes into the source code and will have no effect on the end program.

```
// single-line comment

/* multi-line
   comment */
```

■ ■ ■

Variables

Variables are used for storing data during program execution.

Data types

Depending on what data you need to store there are several kinds of built-in data types. These are often called fundamental data types or *primitives*. The integer (whole number) types are `short`, `int` and `long`. The `float`, `double` and `long  double` types are floating-point (real number) types. The `char` type holds a single character and the `bool` type contains either a true or false value.

Data Type	Size (byte)	Description
char	1	Integer or character
short	2	
int	4	Integer
long	4	
float	4	
double	8	Floating-point number
long double	8	
bool	1	Boolean value

In C++, the exact size and range of the data types are not fixed. Instead they are dependent on the system for which the program is compiled. The sizes shown in the table above are those found on most 32-bit systems and are given in C++ bytes. A byte in C++ is the minimum addressable unit of memory, which is guaranteed to be at least 8 bits, but might also be 16 or 32 bits depending on the system. By definition, a char in C++ is 1 byte in size. Furthermore, the int type will have the same size as the processor's word size, so for a 32-bit system the integers will be 32 bits in size. Each integer type in the table must also be at least as large as the one preceding it. The same applies to floating-point types where each one must provide at least as much precision as the preceding one.

5

Declaring variables

To *declare* (create) a variable you start with the data type you want the variable to hold followed by an *identifier*, which is the name of the variable. The name can consist of letters, numbers and underscores, but it cannot start with a number. It also cannot contain spaces or special characters and must not be a reserved keyword.

```
int myInt;    // correct
int _myInt32; // correct
int 32Int;    // incorrect (starts with number)
int Int 32;   // incorrect (contains space)
int Int@32;   // incorrect (contains special character)
int new;      // incorrect (reserved keyword)
```

Assigning variables

To assign a value to a declared variable the equal sign is used, which is called the assignment operator (=).

```
myInt = 50;
```

The declaration and assignment can be combined into a single statement. When a variable is assigned a value it then becomes *defined*.

```
int myInt = 50;
```

At the same time that the variable is declared there is an alternative way of assigning, or *initializing*, it by enclosing the value in parentheses. This is known as *constructor initialization* and is equivalent to the statement above.

```
int myAlt (50);
```

If you need to create more than one variable of the same type there is a shorthand way of doing it using the comma operator (,).

```
int x = 1, y = 2, z;
```

Octal and hexadecimal assignment

In addition to standard decimal notation, integers can also be assigned by using octal or hexadecimal notation. Both numbers below represent the same number, which in decimal notation is 50.

```
int myOct = 062;  // octal notation (0)
int myHex = 0x32; // hexadecimal notation (0x)
```

Using variables

Once a variable has been defined (declared and assigned) you can use it by simply referencing the variable's name, for example to print it.

```
std::cout << myInt << myAlt; // 5050
```

Variable scope

Variables in C++ may be declared both globally and locally. A global variable is declared outside of any code blocks and will be accessible from anywhere after it has been declared, even in other source files in the same project. A local variable, on the other hand, is declared inside of a function and will only be accessible within that function after it has been declared. The lifetime of a local variable is also limited. A global variable will remain allocated for the duration of the program, while a local variable will be destroyed when its function has finished executing.

```
int globalVar;              // global variable
int main() { int localVar; } // local variable
```

Default values

Global variables in C++ are automatically initialized to zero. Local variables however do not have this advantage. Instead, they will contain whatever garbage is left in that memory location. It is therefore a good idea to always give your local variables an initial value when they are declared.

```
int globalVar;              // initialized to 0
int main() { int localVar; } // uninitialized
```

Integer types

There are four integer types you can use depending on how large a number you need the variable to hold.

```
char  myChar  = 0; // -128   to +127
short myShort = 0; // -32768 to +32767
int   myInt   = 0; // -2^31  to +2^31-1
long  myLong  = 0; // -2^31  to +2^31-1
```

Many C++ compilers also support the long long data type, which is guaranteed to be at least 64-bits. This data type is included in the Microsoft C++ compiler.

```
long long myL2 = 0; // -2^63 to +2^63-1
```

To determine the exact size of a data type you can use the sizeof operator. This operator returns the number of bytes that a data type occupies in the system you are compiling for.

```
std::cout << sizeof(myChar)  // 1 byte (per definition)
         << sizeof(myShort) // 2
         << sizeof(myInt)   // 4
         << sizeof(myLong)  // 4
         << sizeof(myL2);   // 8
```

The Microsoft C++ compiler features five sized integer types. These types start with __int followed by the number of bits you want the integer to hold – either 8, 16, 32 or 64 bits.

```
__int8  myInt8  = 0; // 8 bits
__int16 myInt16 = 0; // 16 bits
__int32 myInt32 = 0; // 32 bits
__int64 myInt64 = 0; // 64 bits
```

Signed and unsigned integers

By default, all the number types in Microsoft C++ are signed and may therefore contain both positive and negative values. To explicitly declare a variable as signed the signed keyword can be used.

```
signed char  myChar  = 0; // -128   to +127
signed short myShort = 0; // -32768 to +32767
signed int   myInt   = 0; // -2^31  to +2^31-1
signed long  myLong  = 0; // -2^31  to +2^31-1
signed long long myL2= 0; // -2^63  to +2^63-1
```

If you only need to store positive values you can declare integer types as unsigned to double their upper range.

```
unsigned char  myChar  = 0; // 0 to 255
unsigned short myShort = 0; // 0 to 32767
unsigned int   myInt   = 0; // 0 to 2^32-1
unsigned long  myLong  = 0; // 0 to 2^32-1
unsigned long long myL2= 0; // 0 to 2^64-1
```

The signed and unsigned keywords may be used as standalone types, which are short for signed int and unsigned int.

```
unsigned uInt; // unsigned int
signed sInt;   // signed int
```

Similarly, the short and long data types are abbreviations of short int and long int.

```
short myShort; // short int
long myLong;   // long int
```

Floating-point types

The floating-point types can store real numbers with different levels of precision.

```
float  myFloat  = 3.14;        // 3.4E +/- 38 (7 digits)
double myDouble = 3.14;        // 1.7E +/- 308 (15 digits)
long double myLongDouble = 3.14; // same as double
```

The precision shown above refers to the total number of digits in the number. For example, trying to assign more than 7 digits to a `float` means that the least significant digits will get rounded off.

```
myFloat = 12345.678; // rounded to 12345.68
```

Floats and doubles can be assigned by using either decimal or exponential notation.

```
myFloat = 3e2; // 3*10^2 = 300
```

Char type

The `char` type is commonly used to represent ASCII characters.

```
char c = 'x'; // assigns 120 (ASCII for 'x')
```

The conversion between the number stored in the `char` and the character shown when the `char` is printed occurs automatically.

```
std::cout << c; // prints 'x'
```

For another integer type to be displayed as a character it has to be explicitly cast to `char`. An explicit cast is performed by placing the desired data type in parentheses before the variable or constant that is to be converted.

```
int i = c;              // assigns 120
std::cout << i;         // prints 120
std::cout << (char)i; // prints 'x'
```

Bool type

The `bool` type can store a Boolean value, which is a value that can only be either true or false. These values are specified with the `true` and `false` keywords.

```
bool b = false; // true or false value
```

■ ■ ■

Operators

The numerical operators in C++ can be grouped into five types: arithmetic, assignment, comparison, logical and bitwise operators.

Arithmetic operators

There are the four basic arithmetic operators, as well as the modulus operator (%) which is used to obtain the division remainder.

```
x = 3 + 2; // 5 // addition
x = 3 - 2; // 1 // subtraction
x = 3 * 2; // 6 // multiplication
x = 3 / 2; // 1 // division
x = 3 % 2; // 1 // modulus (division remainder)
```

Notice that the division sign gives an incorrect result. This is because it operates on two integer values and will therefore truncate the result and return an integer. To get the correct value one of the numbers must be explicitly converted to a floating-point number.

```
x = 3 / (float)2; // 1.5
```

Assignment operators

The second group is the assignment operators. Most importantly, the assignment operator (=) itself, which assigns a value to a variable.

Combined assignment operators

A common use of the assignment and arithmetic operators is to operate on a variable and then to save the result back into that same variable. These operations can be shortened with the combined assignment operators.

```
x += 5; // x = x+5;
x -= 5; // x = x-5;
x *= 5; // x = x*5;
x /= 5; // x = x/5;
x %= 5; // x = x%5;
```

Increment and decrement operators

Another common operation is to increment or decrement a variable by one. This can be simplified with the increment (++) and decrement (--) operators.

```
x++; // x = x+1;
x--; // x = x-1;
```

Both of these can be used either before or after a variable.

```
x++; // post-increment
x--; // post-decrement
++x; // pre-increment
--x; // pre-decrement
```

The result on the variable is the same whichever is used. The difference is that the post-operator returns the original value before it changes the variable, while the pre-operator changes the variable first and then returns the value.

```
x = 5; y = x++; // y=5, x=6
x = 5; y = ++x; // y=6, x=6
```

Comparison operators

The comparison operators compare two values and return either true or false. They are mainly used to specify conditions, which are expressions that evaluate to either true or false.

```
bool x = (2 == 3); // false // equal to
     x = (2 != 3); // true  // not equal to
     x = (2 > 3);  // false // greater than
     x = (2 < 3);  // true  // less than
     x = (2 >= 3); // false // greater than or equal to
     x = (2 <= 3); // true  // less than or equal to
```

Logical operators

The logical operators are often used together with the comparison operators. Logical and (&&) evaluates to true if both the left and right sides are true, and logical or (||) is true if either the left or right side is true. For inverting a Boolean result there is the logical not (!)

operator. Note that for both "logical and" and "logical or" the right-hand side will not be evaluated if the result is already determined by the left-hand side.

```
bool x = (true && false); // false // logical and
     x = (true || false); // true  // logical or
     x = !(true);         // false // logical not
```

Bitwise operators

The bitwise operators can manipulate individual bits inside an integer. For example, the "bitwise or" operator (|) makes the resulting bit 1 if the bits are set on either side of the operator.

```
int x = 5 & 4;  // 101 & 100 = 100 (4)        // and
    x = 5 | 4;  // 101 | 100 = 101 (5)        // or
    x = 5 ^ 4;  // 101 ^ 100 = 001 (1)        // xor
    x = 4 << 1; // 100 << 1  =1000 (8)        // left shift
    x = 4 >> 1; // 100 >> 1  =  10 (2)        // right shift
    x = ~4;     // ~00000100 = 11111011 (-5) // invert
```

The bitwise operators also have combined assignment operators.

```
int x=5; x &= 4; // 101 & 100 = 100 (4) // and
    x=5; x |= 4; // 101 | 100 = 101 (5) // or
    x=5; x ^= 4; // 101 ^ 100 = 001 (1) // xor
    x=5; x <<= 1;// 101 << 1  =1010 (10)// left shift
    x=5; x >>= 1;// 101 >> 1  =  10 (2) // right shift
```

Operator precedence

In C++, expressions are normally evaluated from left to right. However, when an expression contains multiple operators, the precedence of those operators decides the order that they are evaluated in. The order of precedence can be seen in the table below. This same order also applies to many other languages, such as Java and C#.

Pre	Operator	Pre	Operator
1	++ -- ! ~	7	&
2	* / %	8	^
3	+ -	9	|
4	<< >>	10	&&
5	< <= > >=	11	||
6	== !=	12	= op=

For example, logical and (&&) binds weaker than relational operators, which in turn bind weaker than arithmetic operators.

```
bool x = 2+3 > 1*4 && 5/5 == 1; // true
```

To make things clearer, parentheses can be used to specify which part of the expression will be evaluated first. Parentheses have the highest precedence of all operators.

```
bool x = ((2+3) > (1*4)) && ((5/5) == 1); // true
```

CHAPTER 5

■ ■ ■

Pointers

A pointer is a variable that contains the memory address of another variable, called the *pointee*.

Creating pointers

Pointers are declared as any other variable, except that an asterisk (*) is placed between the data type and the pointer's name. The data type used determines what type of memory it will point to.

```
int* p; // pointer to an integer
int *q; // alternative syntax
```

A pointer can point to a variable of the same type by prefixing that variable with an ampersand, in order to retrieve its address and assign it to the pointer. The ampersand is known as the address-of operator (&).

```
int i = 10;
p = &; // address of i assigned to p
```

Dereferencing pointers

The pointer above now contains the memory address to the integer variable. Referencing the pointer will retrieve this address. To obtain the actual value stored in that address the pointer must be prefixed with an asterisk, known as the dereference operator (*).

```
std::cout << "Address of i: " <<  p; // ex. 0017FF1C
std::cout << "Value of i: "   << *p; // 10
```

When writing to the pointer, the same method is used. Without the asterisk the pointer is assigned a new memory address, and with the asterisk the actual value of the variable pointed to will be updated.

```
p = &i;  // address of i assigned to p
*p = 20; // value of i changed through p
```

If a second pointer is created and assigned the value of the first pointer it will then get a copy of the first pointer's memory address.

```
int* p2 = p; // copy of p (copies address stored in p)
```

Pointing to a pointer

Sometimes it can be useful to have a pointer that can point to another pointer. This is done by declaring a pointer with two asterisks and then assigning it the address of the pointer that it will reference. This way when the address stored in the first pointer changes, the second pointer can follow that change.

```
int** r = &p; // pointer to p (assigns address of p)
```

Referencing the second pointer now gives the address of the first pointer. Dereferencing the second pointer gives the address of the variable and dereferencing it again gives the value of the variable.

```
std::cout << "Address of p: " << r;   // ex. 0017FF28
std::cout << "Address of i: " << *r;  // ex. 0017FF1C
std::cout << "Value of i: "   << **r; // 20
```

Dynamic allocation

One of the main usages of pointers is to allocate memory during run-time – so called *dynamic allocation*. In the examples so far, the programs have only had as much memory available as has been declared for the variables at compile-time. This is referred to as *static allocation*. If any additional memory is needed at run-time, the new operator has to be used. This operator allows for dynamic allocation of memory, which can only be accessed through pointers. The new operator takes either a primitive data type or an object as its argument, and it will return a pointer to the allocated memory.

```
int* d = new int; // dynamic allocation
```

An important thing to know about dynamic allocation is that the allocated memory will not be released like the rest of the program memory when it is no longer required. Instead, it has to be manually released with the delete keyword. This allows you to control the lifetime of a dynamically allocated object, but it also means that you are responsible for deleting it once it is no longer needed. Forgetting to delete memory that has been allocated with the new keyword will give the program memory leaks, because that memory will stay allocated until the program shuts down.

```
delete d; // release allocated memory
```

Null pointer

A pointer should be set to zero when it is not assigned to a valid address. Such a pointer is called a *null pointer*. Doing this will allow you to check whether the pointer can be safely dereferenced, because a valid pointer will never be zero.

For example, although the previous pointer has had its memory released, its stored address still points to a now inaccessible memory location. Trying to dereference such a pointer will cause a run-time error. To help prevent this, the deleted pointer should be set to zero. Note that trying to delete an already deleted null pointer is safe. However, if the pointer has not been set to zero, attempting to delete it again will cause memory corruption and possibly crash the program.

```
delete d; d = 0; // mark as null pointer
delete d;        // safe
```

Since you may not always know whether a pointer is valid, a check should be made whenever a pointer is dereferenced to make sure that it is not zero.

```
if (d != 0) { *d = 10; } // check for null pointer
```

The constant NULL can also be used to signify a null pointer. NULL is typically defined as zero in C++, making the choice of which to use a matter of preference. The constant is defined in the stdio.h standard library file, which is included through iostream.

```
#include <iostream>
// ...
if (d != NULL) { *d = 10; } // check for null pointer
```

CHAPTER 6

■ ■ ■

References

References allow a programmer to create a new name for a variable. They provide a simpler, safer and less powerful alternative to pointers.

Creating references

A reference is declared in the same way as a regular variable, except that an ampersand is appended between the data type and the variable name. Furthermore, at the same time as the reference is declared it must be initialized with a variable of the specified type.

```
int x = 5;
int& r = x; // r is an alias to x
int &s = x; // alternative syntax
```

Once the reference has been assigned, or seated, it can never be reseated to another variable. The reference has in effect become an alias for the variable and can be used exactly as though it was the original variable.

```
r = 10; // assigns value to r/x
```

References and pointers

A reference is similar to a pointer that always points to the same thing. However, while a pointer is a variable that points to another variable, a reference is only an alias and does not have an address of its own.

```
int* ptr = &x; // ptr assigned address to x
```

Reference and pointer guideline

Generally, whenever a pointer does not need to be reassigned a reference should be used instead, because a reference is safer than a pointer since it must always refer to a variable. This means that there is no need to check if a reference refers to null, as should be done

with pointers. It is possible for a reference to be invalid – for example when a reference refers to a null pointer – but it is much easier to avoid this kind of mistake with references than it is with pointers.

```
int* ptr = 0; // null pointer
int& ref = *ptr;
ref = 10;     // segmentation fault (invalid memory access)
```

■ ■ ■

Arrays

An array is a data structure used for storing a collection of values that all have the same data type.

Array declaration and allocation

To declare an array you start as you would a normal variable declaration, but in addition append a set of square brackets following the array's name. The brackets contain the number of elements in the array. The default values for these elements are the same as for variables – elements in global arrays are initialized to their default values and elements in local arrays remain uninitialized.

```
int myArray[3]; // integer array with 3 elements
```

Array assignment

To assign values to the elements you can reference them one at a time by placing the element's index inside the square brackets, starting with zero.

```
myArray[0] = 1;
myArray[1] = 2;
myArray[2] = 3;
```

You can also assign values at the same time as the array is declared by enclosing them in curly brackets. The specified array length may be left out to let the array size be decided by the number of values assigned.

```
int myArray[3] = { 1, 2, 3 };
int myArray[] = { 1, 2, 3 };
```

Once the array elements are initialized they can be accessed by referencing the index of the element you want.

```
std::cout << myArray[0]; // 1
```

Multi-dimensional arrays

Arrays can be made multi-dimensional by adding more sets of square brackets. As with single-dimensional arrays, they can either be filled in one at a time or all at once during the declaration.

```
int myArray[2][2] = { { 0, 1 }, { 2, 3 } };
myArray[0][0] = 0;
myArray[0][1] = 1;
```

Dynamic arrays

Because the arrays above are made up of static (non-dynamic) memory, their size must be determined before execution. Therefore, the size needs to be a constant value. In order to create an array with a size that is not known until run-time you need to use dynamic memory, which is allocated with the new keyword and must be assigned to a pointer or reference.

```
int* p = new int[3]; // dynamically allocated array
```

Any array in C++ is actually a pointer to the first element in the array. The referencing of array elements can be made just as well with pointer arithmetic. By incrementing the pointer by one you move to the next element in the array, because changes to a pointer's address are implicitly multiplied by the size of the pointer's data type.

```
*(p+1) = 10; // p[1] = 10;
```

Determining array size

Keep in mind that just as with any other pointer it is possible to exceed the valid range of an array. It is therefore important to keep track of the array's length. To determine the length of a statically allocated array you can use the sizeof operator.

```
int length = sizeof(myArray) / sizeof(int); // 3
```

This method cannot be used for dynamically allocated arrays. The only way to determine the size of such an array is through the variable used in its allocation.

```
int size = 3;
int* p = new int[size]; // dynamically allocated array
```

When you are done using a dynamic array you must remember to delete it. This is done using the delete keyword with an appended set of square brackets.

```
delete[] p; // release allocated array
```

■ ■ ■

String

The string class in C++ is used to store string values. Before a string can be declared the string header must first be included. The standard namespace can also be included since the string class is part of that namespace.

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

Strings can then be declared like any other data type. To assign a string value to a string variable, delimit the literals by double quotes and assign them to the variable. The initial value can also be assigned through constructor initialization at the same time as the string is declared.

```
string h = "Hello";
string w (" World");
```

String combining

The plus sign, known as the concatenation operator (+) in this context, is used to combine two strings. It has an accompanying assignment operator (+=) to append a string.

```
string a = h + w; // Hello World
h += w;           // Hello World
```

The concatenation operator will work as long as one of the strings it operates on is a C++ string.

```
string b = "Hello" + w; // ok
```

It is not able to concatenate two C strings or two string literals. To do this, one of the values has to be explicitly cast to a string.

```
char *c = "World";              // C-style string
b = (string)c + c;              // ok
b = "Hello" + (string)" World"; // ok
```

String literals will also be implicitly combined if the plus sign is left out.

```
b = "Hel" "lo"; // ok
```

Escape characters

A string literal can be extended to more than one line by putting a backslash sign (\)
at the end of each line.

```
string s = "Hello \
          World";
```

To add a new line to the string itself, the escape character "\n" is used.

```
s = "Hello \n World";
```

This backslash notation is used to write special characters, such as tab or form
feed characters.

Character	Meaning	Character	Meaning
\n	newline	\f	form feed
\t	horizontal tab	\a	alert sound
\v	vertical tab	\'	single quote
\b	Backspace	\"	double quote
\r	carriage return	\\	backslash
\0	null character		

Additionally, any one of the 128 ASCII characters can be expressed by writing a
backslash followed by the ASCII code for that character, represented as either an octal or
hexadecimal number.

```
"\07F"  // octal character (0-07F)
"\0x177" // hexadecimal character (0-0x177)
```

String compare

The way to compare two strings is simply by using the equal to operator (==). This will not
compare the memory addresses of the strings, as is the case of C strings.

```
string s = "Hello";
bool b = (s == "Hello"); // true
```

String functions

The string class has a lot of functions. Among the most useful ones are the length and size functions, which both return the number of characters in the string. Their return type is size_t, which is an unsigned data type used to hold the size of an object. This is simply an alias for one of the built-in data types, but which one it is defined as varies between compilers. The alias is defined in the crtdefs.h standard library file, which is included through iostream.

```
size_t i = s.length(); // 5, length of string
i = s.size();          // 5, same as length()
```

Another useful function is substr (substring), which requires two parameters. The second parameter is the number of characters to return starting from the position specified in the first parameter.

```
s.substr(0,2); // "He"
```

A single character can also be extracted or changed by using the array notation.

```
char c = s[0]; // 'H'
```

CHAPTER 9

■ ■ ■

Conditionals

Conditional statements are used to execute different code blocks based on different conditions.

If statement

The if statement will only execute if the expression inside the parentheses is evaluated to true. In C++, this does not have to be a Boolean expression. It can be any expression that evaluates to a number, in which case zero is false and all other numbers are true.

```
if (x < 1) {
  cout << x << " < 1";
}
```

To test for other conditions, the if statement can be extended by any number of else if clauses.

```
else if (x > 1) {
  cout << x << " > 1";
}
```

The if statement can have one else clause at the end, which will execute if all previous conditions are false.

```
else {
  cout << x << " == 1";
}
```

As for the curly brackets, they can be left out if only a single statement needs to be executed conditionally.

```
if (x < 1)
  cout << x << " < 1";
else if (x > 1)
  cout << x << " > 1";
else
  cout << x << " == 1";
```

Switch statement

The switch statement checks for equality between an integer and a series of case labels, and then passes execution to the matching case. It may contain any number of case clauses and it can end with a default label for handling all other cases.

```
switch (x)
{
  case 0: cout << x << " is 0"; break;
  case 1: cout << x << " is 1"; break;
  default: cout << x << " is not 1 or 2"; break;
}
```

Note that the statements after each case label end with the break keyword to skip the rest of the switch. If the break is left out, execution will fall through to the next case, which can be useful if several cases need to be evaluated in the same way.

Ternary operator

In addition to the if and switch statements there is the ternary operator (?:) that can replace a single if/else clause. This operator takes three expressions. If the first one is true then the second expression is evaluated and returned, and if it is false, the third one is evaluated and returned.

```
x = (x < 0.5) ? 0 : 1; // ternary operator (?:)
```

C++ allows expressions to be used as stand-alone code statements. Because of this the ternary operator cannot just be used as an expression, but also as a statement.

```
(x < 0.5) ? x = 0 : x = 1; // alternative syntax
```

The programming term *expression* refers to code that evaluates to a value, whereas a *statement* is a code segment that ends with a semicolon or a closing curly bracket.

28

■ ■ ■

Loops

There are three looping structures available in C++, all of which are used to execute a specific code block multiple times. Just as with the conditional if statement, the curly brackets for the loops can be left out if there is only one statement in the code block.

While loop

The while loop runs through the code block only if its condition is true, and will continue looping for as long as the condition remains true. Bear in mind that the condition is only checked at the start of each iteration (loop).

```
int i = 0;
while (i < 10) { cout << i++; } // 0-9
```

Do-while loop

The do-while loop works in the same way as the while loop, except that it checks the condition after the code block. It will therefore always run through the code block at least once. Notice that this loop ends with a semicolon.

```
int j = 0;
do { cout << j++; } while (j < 10); // 0-9
```

For loop

The for loop is used to run through a code block a specific number of times. It uses three parameters. The first one initializes a counter and is always executed once before the loop. The second parameter holds the condition for the loop and is checked before each iteration. The third parameter contains the increment of the counter and is executed at the end of each loop.

```
for (int k = 0; k < 10; k++) { cout << k; } // 0-9
```

The for loop has several variations. For starters, the first and third parameters can be split into several statements by using the comma operator.

```
for (int k = 0, m = 0; k < 10; k++, m--) {
  cout << k+m; // 0x10
}
```

There is also the option of leaving out any one of the parameters.

```
for (;;) {
  cout << "infinite loop";
}
```

Break and continue

There are two jump statements that can be used inside loops: break and continue. The break keyword ends the loop structure, and continue skips the rest of the current iteration and continues at the beginning of the next iteration.

```
for (int i = 0; i < 10; i++)
{
  break;    // end loop
  continue; // start next iteration
}
```

Goto statement

A third jump statement that may be useful to know of is goto, which performs an unconditional jump to a specified label. This instruction is generally never used since it tends to make the flow of execution difficult to follow.

```
goto myLabel; // jump to label
myLabel:      // label declaration
```

■ ■ ■

Functions

Functions are reusable code blocks that will only execute when called.

Defining functions

A function can be created by typing void followed by the function's name, a set of parentheses and a code block. The void keyword means that the function will not return a value. The naming convention for functions is the same as for variables – a descriptive name with each word initially capitalized, except for the first one.

```
void myFunction()
{
  cout << "Hello World";
}
```

Calling functions

The function above will simply print out a text message when it is called. To invoke it from the main function the function's name is specified followed by a set of parentheses.

```
int main()
{
  myFunction(); // "Hello World"
}
```

Function parameters

The parentheses that follow the function name are used to pass arguments to the function. To do this the corresponding parameters must first be added to the function declaration in the form of a comma separated list.

```
void myFunction(string a, string b)
{
  cout << a + " " + b;
}
```

A function can be defined to take any number of parameters, and they can have any data types. Just ensure the function is called with the same types and number of arguments.

```
myFunction("Hello", "World"); // "Hello World"
```

To be precise, *parameters* appear in function definitions, while *arguments* appear in function calls. However, the two terms are sometimes used interchangeably.

Default parameter values

It is possible to specify default values for parameters by assigning them a value inside the parameter list.

```
void myFunction(string a, string b = "Earth")
{
  cout << a + " " + b;
}
```

Then, if that argument is unspecified when the function is called the default value will be used instead. For this to work it is important that the parameters with default values are to the right of those without default values.

```
myFunction("Hello"); // "Hello Earth"
```

Function overloading

A function in C++ can be defined multiple times with different arguments. This is a powerful feature called function overloading that allows a function to handle a variety of parameters without the programmer using the function needing to be aware of it.

```
void myFunction(string a, string b) { cout << a+" "+b; }
void myFunction(string a)           { cout << a; }
void myFunction(int a)              { cout << a; }
```

Return statement

A function can return a value. The void keyword is then replaced with the data type the function will return, and the return keyword is added to the function's body followed by an argument of the specified return type.

```
int getSum(int a, int b)
{
  return a + b;
}
```

Return is a jump statement that causes the function to exit and return the specified value to the place where the function was called. For example, the function above can be passed as an argument to the output stream since the function evaluates to an integer.

```
cout << getSum(5, 10); // 15
```

The return statement can also be used in void functions to exit before the end block is reached.

```
void dummy() { return; }
```

Note that although the main function is set to return an integer type, it does not have to explicitly return a value. This is because the compiler will automatically add a return zero statement to the end of the main function.

```
int main() { return 0; }
```

Forward declaration

An important thing to keep in mind in C++ is that functions must be declared before they can be called. This does not mean that the function has to be implemented before it is called. It only means that the function's header needs to be specified at the beginning of the source file, so that the compiler knows that the function exists. This kind of forward declaration is known as a *prototype*.

```
void myFunction(int a); // prototype
int main()
{
  myFunction(0);
}
void myFunction(int a) {}
```

The parameter names in the prototype do not need to be included. Only the data types must be specified.

```
void myFunction(int);
```

Pass by value

In C++, variables of both primitive and object data types are by default passed by value. This means that only a copy of the value or object is passed to the function. Therefore, changing the parameter in any way will not affect the original, and passing a large object will be very slow.

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

void change(int i)    { i = 10; }
void change(string s) { s = "Hello World"; }

int main()
{
  int x = 0;     // value type
  change(x);     // value is passed
  cout << x;     // 0

  string y = ""; // reference type
  change(y);     // object copy is passed
  cout << y;     // ""
}
```

Pass by reference

Alternatively, to instead pass a variable by reference you just need to add an ampersand before the parameter's name in the function's definition. When arguments are passed by reference, both primitive and object data types can be changed or replaced and the changes will affect the original.

```
void change(int& i) { i = 10; }

int main()
{
  int x = 0; // value type
  change(x); // reference is passed
  cout << x; // 10
}
```

Pass by address

As an alternative to passing by reference, arguments may also be passed by address using pointers. This passing technique serves the same purpose as passing by reference, but uses pointer syntax instead.

```
void change(int* i) { *i = 10; }

int main()
{
  int x = 0;  // value type
  change(&x); // address is passed
  cout << x;  // 10
}
```

Return by value, reference or address

In addition to passing variables by value, reference or address, a variable may also be returned in one of these ways. Most commonly, a function returns by value, in which case a copy of the value is returned to the caller.

```
int byVal(int i) { return i + 1; }

int main()
{
  int a = 10;
  cout << byVal(a); // 11
}
```

To return by reference instead, an ampersand is placed after the function's return type. The function must then return a variable and may not return an expression or literal, as can be done when using return by value. The variable returned should never be a local variable, since the memory to these variables is released when the function ends. Instead, return by reference is commonly used to return an argument that has also been passed to the function by reference.

```
int& byRef(int& i) { return i; }

int main()
{
  int a = 10;
  cout << byRef(a); // 10
}
```

To return by address the dereference operator is appended to the function's return type. This return technique has the same two restrictions as when returning by reference – the address of a variable must be returned and that returned variable must not be local to the function.

```
int* byAdr(int* i) { return i; }

int main()
{
  int a = 10;
  cout << *byAdr(&a); // 10
}
```

Inline functions

A thing to keep in mind when using functions is that every time a function is called, a performance overhead occurs. To potentially remove this overhead you can recommend that the compiler inlines the calls to a specific function by using the `inline` function modifier. This keyword is best suited for small functions that are called inside loops. It should not be used on larger functions since inlining these can severely increase the size of the code, which will instead decrease performance.

```
inline int myInc(int i) { return i++; }
```

Note that the `inline` keyword is only a recommendation. The compiler may in its attempts to optimize the code choose to ignore this recommendation and it may also inline functions that do not have the `inline` modifier.

■ ■ ■

Class

A class is a template used to create objects. To define one the `class` keyword is used followed by a name, a code block and a semicolon. The naming convention for classes is mixed case, meaning that each word should be initially capitalized.

```
class MyRectangle {};
```

Class members can be declared inside the class; the two main kinds are fields and methods. Fields are variables and they hold the state of the object. Methods are functions and they define what the object can do.

```
class MyRectangle
{
  int x, y;
};
```

Class methods

A method belonging to a class is normally declared as a prototype inside of the class, and the actual implementation is placed after the class's definition. The method's name outside the class then needs to be prefixed with the class name and the scope resolution operator in order to designate to which class the method definition belongs.

```
class MyRectangle
{
  int x, y;
  int getArea();
};

int MyRectangle::getArea() { return x * y; }
```

Inline methods

If the method is short and you want to recommend to the compiler that the function's code should be inserted (inlined) into the caller's code, one way to do this would be to use the `inline` keyword in the method's definition.

```
inline int MyRectangle::getArea() { return x * y; }
```

A more convenient way is to simply define the method inside of the class. This will implicitly recommend to the compiler that the method should be inlined.

```
class MyRectangle
{
  int x, y;
  int getArea() { return x * y; }
};
```

Object creation

The class definition is now complete. In order to use it you first have to create an object of the class, also called an instance. This can be done in the same way as variables are declared.

```
int main()
{
  MyRectangle r; // object creation
}
```

Accessing object members

Before the members that this object contains can be accessed, they first need to be declared as public in the class definition, by using the `public` keyword followed by a colon.

```
class MyRectangle
{
 public:
  int x, y;
  int getArea() { return x * y; }
};
```

38

The members of this object can now be reached using the dot operator (`.`) after the instance name.

```
r.x = 10;
r.y = 5;
int z = r.getArea(); // 50 (5*10)
```

Any number of objects can be created based on a class, and each one of them will have its own set of fields and methods.

```
MyRectangle r2; // another instance of MyRectangle
r2.x = 25;      // not same as r.x
```

When using an object pointer, the arrow operator (`->`) allows access to the object's members. This operator behaves like the dot operator, except that it dereferences the pointer first. It is used exclusively with pointers to objects.

```
MyRectangle r;
MyRectangle *p = &r; // object pointer

p->getArea();
(*p).getArea();      // alternative syntax
```

Forward declaration

Classes, just like functions, must be declared before they can be referenced. If there is a circular reference then changing the order in which the classes are defined will not solve the problem. Instead, what needs to be done in this situation is to declare a class prototype. This prototype declaration allows the class to be referenced before it is defined, as is done in the example below by using a pointer. Note that even with a prototype you still cannot create an object of a class before it has been defined.

```
class MyClass; // class prototype

class OtherClass
{
  MyClass* m;
};

class MyClass
{
  OtherClass* o;
};
```

■ ■ ■

Constructor

In addition to fields and methods, a class can contain a *constructor*. This is a special kind of method used to construct, or *instantiate*, the object. It always has the same name as the class and does not have a return type. To be accessible from another class the constructor needs to be declared in a section marked with the public access modifier.

```
class MyRectangle
{
 public:
  int x, y;
  MyRectangle();
};

MyRectangle::MyRectangle() { x = 10; y = 5; }
```

When a new instance of this class is created the constructor method will be called, which in this case assigns default values to the fields.

```
int main()
{
  MyRectangle s;
}
```

Constructor overloading

As with any other method the constructor can be overloaded. This will allow an object to be created with different parameter lists.

```
class MyRectangle
{
 public:
  int x, y;
  MyRectangle();
  MyRectangle(int, int);
};
```

```
MyRectangle::MyRectangle() { x = 10; y = 5; }
MyRectangle::MyRectangle(int a, int b) { x = a; y = b; }
```

For example, with the two constructors defined above the object can be initialized either with no arguments or with two arguments, which will be used to assign the fields.

```
// Calls parameterless constructor
MyRectangle r;

// Calls constructor accepting two integers
MyRectangle t(2,3);
```

This keyword

Inside the constructor, as well as in other methods belonging to the object – so called *instance methods* – a special keyword called this can be used. This is a pointer to the current instance of the class. It can be useful if, for example, the constructor's parameter names are the same as the field names. The fields can then still be accessed by using the this pointer, even though they are overshadowed by the parameters.

```
MyRectangle::MyRectangle(int x, int y)
{
  this->x = x; this->y = y;
}
```

Constructor initialization list

As an alternative to assigning fields inside the constructor, they may also be assigned by using the *constructor initialization list*. This list starts with a colon after the constructor parameters, followed by calls to the field's own constructors. This is actually the recommended way of assigning fields, because it gives better performance than assigning the fields inside the constructor.

```
MyRectangle::MyRectangle(int a, int b) : x(a), y(b) {}
```

Default constructor

If no constructors are defined for a class the compiler will automatically create a default parameterless constructor when the program compiles. Because of this, a class can be instantiated even if no constructor has been implemented. The default constructor will only allocate memory for the object. It will not initialize the fields. In contrast to global variables, fields in C++ are not automatically initialized to their default values. The fields will contain whatever garbage is left in their memory locations until they are explicitly assigned values.

Destructor

In addition to constructors, a class can also have a *destructor*. The destructor is used to release any resources allocated by the object. It is called automatically before an object is destroyed, either when the object passes out of scope or when it is explicitly deleted for objects created with the new operator. The name of the destructor is the same as the class name, but preceded by a tilde (~). A class may only have one destructor and it never takes any arguments or returns anything.

```
class Semaphore
{
 public:
  bool *sem;

  Semaphore()  { sem = new bool; }
  ~Semaphore() { delete sem; }
};
```

Direct initialization

The object creation syntax that has been used so far is called *direct initialization*. This syntax can include a set of parentheses which are used to pass arguments to a constructor in the class. If the parameterless constructor is used, the parentheses are left out.

```
// Direct initialization
MyClass a(5);
MyClass b;
```

Value initialization

An object can also be *value initialized*. The object is then created by using the class name followed by a set of parentheses. The parentheses can supply constructor arguments, or remain empty to construct the object using the parameterless constructor. A value initialization creates only a temporary object, which is destroyed at the end of the statement. To preserve the object it must either be copied to another object or assigned to a constant reference. Assigning the temporary object to a constant reference will maintain the object until that reference goes out of scope.

```
// Value initialization
const MyClass& a = MyClass();
```

A value initialized object is almost identical to one created by using default initialization. A minor difference is that non-static fields will in some cases be initialized to their default values when using value initialization.

Copy initialization

If an existing object is assigned to an object of the same type when it is declared, the new object will be *copy initialized*. This means that each member of the existing object will be copied to the new object.

```
// Copy initialization
MyClass a = MyClass();
MyClass b(a);
```

This occurs because the compiler will automatically create a *copy constructor* for each class, which is called for these kinds of assignments. The copy constructor takes a single argument of its own type, and then constructs a copy of the specified object. Note that this behavior is different from many other languages, such as Java and C#. In those languages initializing an object with another object will only copy the object's reference, and not create a new object copy.

New initialization

Finally, an object can be initialized through dynamic memory allocation by using the new keyword. Dynamically allocated memory must be used through a pointer or reference. The new operator returns a pointer, so to assign it to a reference it needs to be dereferenced first. Keep in mind that dynamically allocated memory must be explicitly freed once it is no longer needed.

```
// New initialization
MyClass* a = new MyClass();
MyClass& b = *new MyClass();
// ...
delete a, b;
```

■ ■ ■

Inheritance

Inheritance allows a class to acquire the members of another class. In the example below, Square inherits from Rectangle. This is specified after the class name by using a colon followed by the public keyword, and the name of the class to inherit from. Rectangle then becomes a base class of Square, which in turn becomes a derived class of Rectangle. In addition to its own members, Square gains all accessible members in Rectangle, except for its constructors and destructor.

```
class Rectangle
{
 public:
  int x, y;
  int getArea() { return x * y; }
};

class Square : public Rectangle {};
```

Upcasting

An object can be upcast to its base class, because it contains everything that the base class contains. An upcast is performed by assigning the object to either a reference or a pointer of its base class type. In the example below, a Square object is upcast to Rectangle. When using Rectangle's interface the Square object will be viewed as a Rectangle, so only Rectangle's members can be accessed.

```
Square s;
Rectangle& r = s;  // reference upcast
Rectangle* p = &s; // pointer upcast
```

A derived class can be used anywhere a base class is expected. For example, a Square object can be passed as an argument to a function that expects a Rectangle object. The derived object will then implicitly be upcast to its base type.

```
void setXY(Rectangle& r) { r.x = 2; r.y = 3; }

int main() {
  Square s;
  setXY(s);
}
```

Downcasting

A Rectangle reference that points to a Square object can be downcast back to a Square object. This downcast has to be made explicit since downcasting an actual Rectangle to a Square is not allowed.

```
Square& a = (Square&) r;  // reference downcast
Square& b = (Square&) *p; // pointer downcast
```

Multiple inheritance

C++ allows a derived class to inherit from more than one base class. This is called *multiple inheritance*. The base classes are specified in a comma-separated list.

```
class Person {}
class Employee {}

class Teacher: public Person, public Employee {}
```

Multiple inheritance is not commonly used since most real-world relationships can be better described by single inheritance. It also tends to significantly increase the complexity of the code.

CHAPTER 15

Overriding

A new method in a derived class can redefine a method in a base class in order to give it a new implementation.

Hiding derived members

In the example below, Rectangle's getArea method is redeclared in Triangle with the same signature. The signature includes the name, parameter list and return type of the method.

```
class Rectangle
{
 public:
  int x, y;
  int getArea() { return x * y; }
};

class Triangle : public Rectangle
{
 public:
  Triangle(int a, int b) { x = a; y = b; }
  int getArea() { return x * y / 2; }
};
```

If a Triangle object is created and the getArea method is invoked, then Triangle's version of the method will get called.

```
Triangle t = Triangle(2,3);
t.getArea(); // 3 (2*3/2) calls Triangle's version
```

However, if the Triangle is upcast to a Rectangle then Rectangle's version will get called instead.

```
Rectangle& r = t;
r.getArea(); // 6 (2*3) calls Rectangle's version
```

That is because the redefined method has only hidden the inherited method. This means that Triangle's implementation is redefined downwards in the class hierarchy to any child classes of Triangle, but not upwards to the base class.

Overriding derived members

In order to redefine a method upwards in the class hierarchy – what is called *overriding* – the method needs to be declared with the virtual modifier in the base class. This modifier allows the method to be overridden in derived classes.

```
class Rectangle
{
 public:
  int x, y;
  virtual int getArea() { return x * y; }
};
```

Calling the getArea method from Rectangle's interface will now invoke Triangle's implementation.

```
Rectangle& r = t;
r.getArea(); // 3 (2*3/2) calls Triangle's version
```

Base class scoping

It is still possible to access a redefined method from a derived class by typing the class name followed by the scope resolution operator. This is called *base class scoping* and can be used to allow access to redefined methods that are any number of levels deep in the class hierarchy.

```
class Triangle : public Rectangle
{
 public:
  Triangle(int a, int b) { x = a; y = b; }
  int getArea() { return Rectangle::getArea() / 2; }
};
```

Calling base class constructor

Another place where base class scoping is important is in the constructors of derived classes. In C++, constructors in derived classes will implicitly call the parameterless base class constructor. To call another base class constructor it will need to be explicitly invoked at the beginning of the derived constructor's initialization list.

```
class Rectangle
{
 public:
  int x, y;
  Rectangle(int a, int b) { x = a; y = b; }
};

class Triangle : public Rectangle
{
 public:
  Triangle(int a, int b) : Rectangle(a,b) {}
};
```

Note that the base class constructor does not have to be qualified with the class name.

49

■ ■ ■

Access Levels

Every class member has an accessibility level that determines where the member will be visible. There are three of them available in C++: public, protected and private. The default access level for class members is private. To change the access level for a section of code, an access modifier is used followed by a colon. Every field or method that comes after this label will have the specified access level, until another access level is set or the class declaration ends.

```
class MyClass
{
  int myPrivate;

 public:
  int myPublic;
  void publicMethod();
};
```

Private access

All members regardless of their access level are accessible in the class in which they are declared, the enclosing class. This is the only place where private members can be accessed.

```
class MyClass
{
  // Unrestricted access
  public: int myPublic;

  // Defining or derived class only
  protected: int myProtected;

  // Defining class only
  private: int myPrivate;
```

```
  void test()
  {
    myPublic    = 0; // allowed
    myProtected = 0; // allowed
    myPrivate   = 0; // allowed
  }
};
```

Protected access

A protected member can also be accessed from inside a derived class, but it cannot be reached from an unrelated class.

```
class MyChild : public MyClass
{
  void test()
  {
    myPublic    = 0; // allowed
    myProtected = 0; // allowed
    myPrivate   = 0; // inaccessible
  }
};
```

Public access

Public access gives unrestricted access from anywhere in the code.

```
class OtherClass
{
  void test(MyClass& c)
  {
    c.myPublic    = 0; // allowed
    c.myProtected = 0; // inaccessible
    c.myPrivate   = 0; // inaccessible
  }
};
```

Access level guideline

As a guideline, when choosing an access level it is generally best to use the most restrictive level possible. This is because the more places a member can be accessed, the more places it can be accessed incorrectly, which makes the code harder to debug. Using restrictive access levels will also make it easier to modify the class without breaking the code for any other programmers using that class.

Friend classes and functions

A class can be allowed to access the private and protected members of another class by declaring the class a friend. This is done by using the friend modifier. The friend is allowed to access all members in the class where the friend is defined, but not the other way around.

```
class MyClass
{
  int myPrivate;

  // Give OtherClass access
  friend class OtherClass;
};

class OtherClass
{
  void test(MyClass c) { c.myPrivate = 0; } // allowed
};
```

A global function can also be declared as a friend to a class in order to gain the same level of access.

```
class MyClass
{
  int myPrivate;

  // Give myFriend access
  friend void myFriend(MyClass c);
};

void myFriend(MyClass c) { c.myPrivate = 0; } // allowed
```

Public, protected and private inheritance

When a class is inherited in C++ it is possible to change the access level of the inherited members. Public inheritance allows all members to keep their original access level. Protected inheritance reduces the access of public members to protected. Private inheritance restricts all inherited members to private access.

```
class MyChild : private MyClass
{
  // myPublic is private
  // myProtected is private
  // myPrivate is private
};
```

Private is the default inheritance level, although public inheritance is the one that is nearly always used.

■ ■ ■

Static

The static keyword is used to create class members that exist in only one copy, which belongs to the class itself. These members are shared among all instances of the class. This is different from instance (non-static) members, which are created as new copies for each new object.

Static fields

A static field (class field) cannot be initialized inside the class like an instance field. Instead it must be defined outside of the class declaration. This initialization will only take place once, and the static field will then remain initialized throughout the life of the application.

```
class MyCircle
{
 public:
  double r;         // instance field (one per object)
  static double pi; // static field (only one copy)
};

double MyCircle::pi = 3.14;
```

To access a static member from outside the class, the name of the class is used followed by the scope resolution operator and the static member. This means that there is no need to create an instance of a class in order to access its static members.

```
int main()
{
  double p = MyCircle::pi;
}
```

Static methods

In addition to fields, methods can also be declared as static, in which case they can also be called without having to define an instance of the class. However, because a static method is not part of any instance it cannot use instance members. Methods should therefore only be declared static if they perform a generic function that is independent of any instance variables. Instance methods on the other hand, in contrast to static methods, can use both static and instance members.

```
class MyCircle
{
 public:
  double r;         // instance variable (one per object)
  static double pi; // static variable (only one copy)

  double getArea() { return pi * r * r; }
  static double newArea(double a) { return pi * a * a; }
};

int main()
{
  double a = MyCircle::newArea(1);
}
```

Static local variables

Local variables inside a function can be declared as static to make the function remember the variable. A static local variable is only initialized once when execution first reaches the declaration, and that declaration is then ignored every subsequent time the execution passes through.

```
int myFunc()
{
  static int count = 0; // holds # of calls to function
  count++;
}
```

Static global variables

One last place where the static keyword can be applied is to global variables. This will limit the accessibility of the variable to only the current source file, and can therefore be used to help avoid naming conflicts.

```
// Only visible within this source file
static int myGlobal;
```

■ ■ ■

Enum

Enum is a user-defined type consisting of a fixed list of named constants. In the example below, the enumeration type is called Color and contains three constants: Red, Green and Blue.

```
enum Color { Red, Green, Blue };
```

The Color type can be used to create variables that may hold one of these constant values.

```
int main()
{
  Color c = Red;
}
```

In order to avoid naming conflicts the constant can be qualified with the name of the enum.

```
Color c = Color.Red;
```

Enum example

The switch statement provides a good example of when enumerations can be useful. Compared to using ordinary constants, the enumeration has the advantage that it allows the programmer to clearly specify what values a variable should contain.

```
switch(c)
{
  case Red:   break;
  case Green: break;
  case Blue:  break;
}
```

Enum constant values

Usually there is no need to know the underlying values that the constants represent, but in some cases it can be useful. By default, the first constant in the enum list has the value zero and each successive constant is one value higher.

```
enum Color
{
  Red   // 0
  Green // 1
  Blue  // 2
};
```

These default values can be overridden by assigning values to the constants. The values can be computed and do not have to be unique.

```
enum Color
{
  Red   = 5,        // 5
  Green = Red,      // 5
  Blue  = Green + 2 // 7
};
```

Enum conversions

The compiler can implicitly convert an enumeration constant to an integer. However, converting an integer back into an enum variable requires an explicit cast, since this conversion makes it possible to assign a value that is not included in the enum's list of constants.

```
int i = Red;
Color c = (Color)i;
```

Enum scope

An enum does not have to be declared globally. It can also be placed within a class as a class member, or locally within a function.

```
class MyClass
{
  enum Color { Red, Green, Blue };
};

void myFunction()
{
  enum Color { Red, Green, Blue };
}
```

■ ■ ■

Struct and Union

Struct

A struct in C++ is equivalent to a class, except that members of a struct default to public access, instead of private access as in classes. By convention, structs are used instead of classes to represent simple data structures that mainly contain public fields.

```
struct Point
{
  int x, y; // public
};

class Point
{
  int x, y; // private
};
```

Declarator list

To declare objects of a struct the normal declaration syntax can be used.

```
Point p, q; // object declarations
```

Another alternative syntax commonly used with structs is to declare the objects when the struct is defined by placing the object names before the final semicolon. This position is known as the *declarator list* and can contain a comma-separated sequence of declarators.

```
struct Point
{
  int x, y;
} r, s; // object declarations
```

Aggregate initialization

There is a syntactical shortcut available when initializing an object called *aggregate initialization*. This allows programmers to set fields by using a brace-enclosed comma-separated list of initializer-clauses. Aggregate initialization can only be used when the class type does not include any constructors, virtual functions or base classes. The fields must also be public, unless they are declared as static. This is the reason why it is more common to use aggregate initialization with structs rather than with classes.

```
struct Point {
  int x, y;
} r = { 2, 3 }; // set values of x and y

int main() {
  Point p = { 2, 3 };
}
```

Union

Although similar to struct, the union type is different in that all fields share the same memory position. Therefore, the size of a union is the size of the largest field it contains. For example, in the case below this is the integer field which is 4 bytes large.

```
union Mix
{
  char c;  // 1 byte
  short s; // 2 bytes
  int i;   // 4 bytes
} m;
```

This means that the union type can only be used to store one value at a time, because changing one field will overwrite the value of the others.

```
int main()
{
  m.c = 0xFF; // set first 8 bits
  m.s = 0;    // reset first 16 bits
}
```

The benefit of a union, in addition to efficient memory usage, is that it provides multiple ways of viewing the same memory location. For example, the union below has three data members that allow access to the same group of 4 bytes in multiple ways.

```
union Mix
{
  char c[4];                 // 4 bytes
  struct { short hi, lo; } s; // 4 bytes
  int i;                     // 4 bytes
} m;
```

The integer field will access all 4 bytes at once. With the struct 2 bytes can be viewed at a time, and by using the char array each byte can be referenced individually.

```
int main()
{
  m.i=0xFF00F00F; // 11111111 00000000 11110000 00001111
  m.s.lo;         // 11111111 00000000
  m.s.hi;         //                   11110000 00001111
  m.c[3];         // 11111111
  m.c[2];         //          00000000
  m.c[1];         //                   11110000
  m.c[0];         //                            00001111}
```

Anonymous union

A union type can be declared without a name. This is called an *anonymous union* and defines an unnamed object whose members can be accessed directly from the scope where it is declared. An anonymous union cannot contain methods or non-public members.

```
int main()
{
  union { short s; }; // defines an unnamed union object
  s = 15;
}
```

An anonymous union that is declared globally must be made static.

```
static union {};
```

■ ■ ■

Operator Overloading

Operator overloading allows operators to be redefined and used where one or both of the operands are of a user-defined class. When done correctly, this can simplify the code and make user-defined types as easy to use as the primitive types.

Operator overloading example

In the example below there is a class called MyNum with an integer field and a constructor for setting that field. The class also has an addition method that adds two MyNum objects together and returns the result as a new object.

```
class MyNum
{
 public:
  int val;
  MyNum(int i) : val(i) {}

  MyNum add(MyNum &a)
  { return MyNum( val + a.val ); }
}
```

Two MyNum instances can be added together using this method.

```
MyNum a = MyNum(10), b = MyNum(5);
MyNum c = a.add(b);
```

Binary operator overloading

What operator overloading does is simplify this syntax and thereby provide a more intuitive interface for the class. To convert the add method to an overload for the addition sign, replace the name of the method with the operator keyword followed by the operator that is to be overloaded. The whitespace between the keyword and the operator can optionally be left out.

```
MyNum operator + (MyNum &a)
{ return MyNum( val + a.val ); }
```

Since the class now overloads the addition sign, this operator can be used to perform the calculation needed.

```
MyNum c = a + b;
```

Keep in mind that the operator is only an alternative syntax for calling the actual method.

```
MyNum d = a.operator + (b);
```

Unary operator overloading

Addition is a binary operator, because it takes two operands. The first operand is the object from which the method is called, and the second operand is that which is passed to the method. When overloading a unary operator, such as prefix increment (++), there is no need for a method parameter since these operators only affect the object from which they are called.

With unary operators, a reference of the same type as the object should always be returned. This is because when using a unary operator on an object, programmers expect the result to return the same object and not just a copy. On the other hand, when using a binary operator, programmers expect a copy of the result to be returned and therefore return by value should be used.

```
MyNum& operator++() // ++ prefix
{ ++val; return *this; }
```

Not all unary operators should return by reference. The two postfix operators – post-increment and post-decrement – should instead return by value, because the postfix operations are expected to return the state of the object before the increment or decrement occurs. Note that the postfix operators have an unused int parameter specified. This parameter is used to distinguish them from the prefix operators.

```
MyNum operator++(int) // postfix ++
{
  MyNum t = MyNum(val);
  ++val;
  return t;
}
```

Overloadable operators

C++ allows overloading of almost all operators in the language. As can be seen in the table below, most operators are of the binary type. Only a few of them are unary, and some special operators cannot be categorized as either. There are also some operators that cannot be overloaded at all.

Binary operators	Unary operators
+ - * / %	+ - ! ~ & * ++ --
= + = - = * = / = % =	**Special operators**
& = ^ = | = << = >> =	( ) [ ] delete new
== != > < > = < =	**Not overloadable**
& | ^ << >> && ||	. .* :: ?: # ## sizeof
–> – >* ,	

■ ■ ■

Custom Conversions

This chapter covers how custom type conversions for an object can be defined. In the example below, there is a class called MyNum with a single integer field. With custom type conversions it is possible to allow integer types to be implicitly converted to this object's type.

```
class MyNum
{
 public:
  int value;
};
```

Implicit conversion methods

For this to work a constructor needs to be added that takes a single parameter of the desired type, in this case an int.

```
class MyNum
{
 public:
  int value;
  MyNum(int i) { value = i; }
};
```

When an integer is assigned to an object of MyNum this constructor will implicitly be called to perform the type conversion.

```
MyNum A = 5; // implicit conversion
```

This means that any constructor that takes exactly one argument can be used both for constructing objects and for performing implicit type conversions to that object type.

```
MyNum B = MyNum(5); // object construction
MyNum C(5);         // object construction
```

These conversions will work not only for the specific parameter type, but also for any type that can be implicitly converted to it. For example, a char can be implicitly converted to an int and can therefore be implicitly changed into a MyNum object as well.

```
MyNum D = 'H'; // implicit conversion (char->int->MyNum)
```

Explicit conversion methods

To help prevent potentially unintended object type conversions it is possible to disable the second use of the single parameter constructor. The explicit constructor modifier is then applied, which specifies that the constructor may only be used for object construction, and not for type conversion.

```
class MyNum
{
 public:
  int value;
  explicit MyNum(int i) { value = i; }
};
```

The explicit constructor syntax must therefore be used to create a new object.

```
MyNum A = 5;        // error
MyNum B(5);         // allowed
MyNum C = MyNum(5); // allowed
```

■ ■ ■

Namespaces

Namespaces are used to avoid naming conflicts by allowing entities, such as classes and functions, to be grouped under a separate scope. In the example below there are two classes that belong to the global scope. Since both classes share the same name and scope the code will not compile.

```
class Table {};
class Table {}; // error: class type redefinition
```

One way to solve this problem would be to rename one of the conflicting classes. Another solution is to group one or both of them under a different namespace by enclosing each in a namespace block. The classes then belong to different scopes and so will no longer cause a naming conflict.

```
namespace furniture
{
  class Table {};
}

namespace html
{
  class Table {};
}
```

Accessing namespace members

To access a member of a namespace from outside that namespace the member's fully qualified name needs to be specified. This means that the member name has to be prefixed with the namespace it belongs to, followed by the scope resolution operator.

```
int main()
{
  furniture::Table fTable;
  html::Table hTable;
}
```

Nesting namespaces

It is possible to nest namespaces any number of levels deep to further structure the program entities.

```
namespace furniture
{
  namespace wood { class Table {}; }
}
```

Ensure that the nested namespace members are qualified with the full namespace hierarchy when using them from another namespace.

```
furniture::wood::Table fTable;
```

Importing namespaces

To avoid having to specify the namespace every time one of its members is used, the namespace can be imported into the global or local scope with the help of a using declaration. This declaration includes the using namespace keywords followed by the namespace to be imported. It can be placed either locally or globally. Locally, the declaration will only be in scope until the end of the code block, while at the global scope it will apply to the whole source file following its declaration.

```
using namespace html;   // global namespace import

int main()
{
  using namespace html; // local namespace import
}
```

Keep in mind that importing a namespace into the global scope defeats the main purpose of using namespaces, which is to avoid naming conflicts. Such conflicts however are mainly an issue in projects that use several independently developed code libraries.

Namespace member import

If you want to avoid both typing the fully qualified name and importing the whole namespace there is a third alternative available. That is to only import the specific members that are needed from the namespace. This is done by declaring one member at a time with the using keyword followed by the fully qualified namespace member to be imported.

```
using html::Table; // import a single namespace member
```

Namespace alias

Another way to shorten the fully qualified name is to create a namespace alias. The namespace keyword is then used followed by an alias name, to which the fully qualified namespace is assigned.

```
namespace myAlias = furniture::wood; // namespace alias
```

This alias can then be used instead of the namespace qualifier that it represents.

```
myAlias::Table fTable;
```

Note that both the namespace member imports and the namespace aliases may be declared both globally and locally.

Type alias

Aliases can also be created for types. A type alias is defined using the typedef keyword followed by the type and the alias.

```
typedef my::name::MyClass MyType;
```

The alias can then be used as a synonym for the specified type.

```
MyType t;
```

Typedef does not only work for existing types, but can also include a definition of a user-defined type – such as a class, struct, union or enum.

```
typedef struct { int len; } Length;
Length a, b, c;
```

Typedef aliases are not commonly used since they tend to obfuscate the code. However, if used properly a type alias can simplify a long or confusing type name. Another function they provide is the ability to change the definition of a type from a single location.

Including namespace members

Keep in mind that in C++ merely importing a namespace does not provide access to the members included in that namespace. In order to access the namespace members the prototypes also have to be made available, for example by using the appropriate #include directives.

```
// Include input/output prototypes
#include <iostream>

// Import standard library namespace to global scope
using namespace std;
```

■ ■ ■

Constants

A constant is a variable that has a value which cannot be changed once the constant has been assigned.

Constant variables

A variable can be made into a constant by adding the const keyword either before or after the data type. This modifier means that the variable becomes read-only, and it must therefore be assigned a value at the same time as it is declared. Attempting to change the value will result in a compile-time error.

```
const int var = 5;
int const var2 = 10; // alternative order
```

Constant pointers

When it comes to pointers, const can be used in two ways. First, the pointer can be made constant, which means that it cannot be changed to point to another location.

```
int myPointee;
int *const p = &myPointee; // pointer constant
```

Second, the pointee can be declared constant. This means that the variable pointed to cannot be modified through this pointer.

```
const int *q = &var; // pointee constant
```

It is possible to declare both the pointer and the pointee as constant to make them both read-only.

```
const int *const r = &var; // pointer & pointee constant
```

Note that constant variables may not be pointed to by a non-constant pointer. This prevents programmers from accidentally rewriting a constant variable using a pointer.

```
int *s = &var; // error: const to non-const assignment
```

Constant references

References can be declared constant in the same way as pointers. However, since reseating a reference is never allowed, declaring the reference as const would be redundant. It only makes sense to protect the referee from change.

```
const int& y = var; // referee constant
```

Constant objects

Just as with variables, pointers and references, objects can also be declared constant. Take the following class as an example.

```
class MyClass
{
  public: int x;
  void setX(int a) { x = a; }
};
```

A constant object of this class cannot be reassigned to another instance. The constness of an object also affects its fields and prevent them from being changed.

```
const MyClass a, b;
a = b;        // error: object is const
a.x = 10;     // error: object field is const
```

Constant methods

Because of this last restriction, a constant object may not call a non-constant method since such methods are allowed to change the object's fields.

```
a.setX(2); // error: cannot call non-const method
```

They may only call constant methods, which are methods that are marked with the const modifier before the method body.

```
int getX() const { return x; } // constant method
```

This const modifier means that the method is not allowed to modify the state of the object and can therefore safely be called by a constant object of the class. More specifically, the const modifier applies to the this pointer that is implicitly passed to the method. This effectively restricts the method from modifying the object's fields or calling any non-constant methods in the class.

74

Constant return type and parameters

In addition to making a method constant, the return type and method parameters may also be made read-only. For example, if a field is returned by reference instead of by value from a constant method it is important that it is returned as a constant in order to maintain the constness of the object. Not all C++ compilers will be able to catch this subtle mistake.

```
const int& getX() const { return x; }
```

Constant fields

Both static and instance fields in a class can be declared constant. A constant instance field must be assigned its value using the constructor initialization list. This is the same as the preferred way of initializing regular (non-constant, non-static) fields.

```
class MyClass
{
 public:
  int i;
  const int c;
  MyClass() : c(5), i(5) {}
}
```

A constant static field on the other hand must be defined outside of the class declaration, in the same way as non-constant static fields. The exception to this is when the constant static field is of an integer data type. Such a field may also be initialized within the class at the same time as the field is declared.

```
class MyClass
{
 public:
  static int si;
  const static double csd;
  const static int csi = 5;
};
int MyClass::si = 1.23;
const double MyClass::csd = 1.23;
```

Constant guideline

In general, it is a good idea to always declare variables as constants if they do not need to be modified. This ensures that the variables are not changed anywhere in the program by mistake, which in turn will help to prevent bugs.

■ ■ ■

Preprocessor

The preprocessor is a program invoked by the compiler that modifies the source code before the actual compilation takes place. This modification is done according to the preprocessor directives that are included in the source files. The directives are easily distinguished from normal programming code in that they all start with a hash sign (#). They must always appear as the first non-whitespace character on a line. Below is a table of the preprocessor directives available in C++ and their function.

Directive	Description
#include	File include
#define	Macro definition
#undef	Macro undefine
#ifdef	If macro defined
#ifndef	If macro not defined
#if	If
#elif	Else if
#else	Else
#endif	End if
#line	Set line number
#error	Abort compilation
#pragma	Set compiler option

Including source files

The #include directive inserts the contents of a file into the current source file. If the filename is enclosed between angle brackets the compiler will search for the file in the default directory where it is configured to look for the standard header files.

```
#include <iostream> // search default directory
```

If instead the filename is specified between double-quotes, the compiler will first search for the file in the same directory as the source file and in case it is not there it will then search among the standard header files.

```
#include "MyFile" // search current, then default
```

Double quotes can also be used to specify an absolute or relative path to the file.

```
#include "C:\MyFile" // absolute path
#include "..\MyFile" // relative path
```

The angle bracket form of #include is normally used for programmer defined headers, while the quoted form is used for standard library headers.

Defining macros

Another important directive is #define, which is used to create macros. After the directive, the name of the macro is specified followed by what it will be replaced by.

```
#define MACRO 0 // macro definition
```

The preprocessor will go through and change any occurrences of the macro with whatever comes after it in its definition until the end of the line.

```
int x = MACRO; // x = 0
```

By convention, macros should be named in uppercase letters with each word separated by an underscore. That way they are easy to spot when reading the source code.

Undefining macros

A #define directive should not be used to directly override a previously defined macro. Doing so will give a compiler warning. In order to change a macro it first needs to be undefined using the #undef directive before it is redefined. Note that attempting to undefine a macro that is not currently defined will not generate a warning.

```
#undef MACRO // macro undefine
#undef MACRO // allowed
```

Macro functions

A macro can be made to take arguments. This allows them to define compile-time functions. For example, the macro function below evaluates to the larger of its two arguments.

```
#define MAX(a,b) a>b ? a:b
```

The macro function is called just as if it was a regular C++ function. Keep in mind that for this kind of function to work the arguments must be known at compile-time.

```
int x = MAX(MACRO, 1); // evaluates to 1
```

Line-break

To break a macro function across several lines the backslash character can be used. This will escape the newline character that marks the end of a preprocessor directive. There must not be any whitespace after the backslash.

```
#define MAX(a,b)  \
          a>b ? \
          a:b
```

Avoid using macros

Although #define directives can be powerful, they tend to make the code more difficult to read and debug. Macros should therefore only be used when they are absolutely necessary. C++ code such as constant variables, enums and inline functions can often accomplish the same goal more efficiently and safely than #define directives can.

```
#define DEBUG 0
const bool DEBUG = 0;

#define FORWARD 1
#define STOP 0
#define BACKWARD -1
enum DIR { FORWARD = 1, STOP = 0, BACKWARD = -1 };

#define MAX(a,b) a>b ? a:b
inline int MAX(int a, int b) { return a>b ? a:b; }
```

Conditional compilation directives

The directives used for conditional compilation can include or exclude part of the source code if a certain condition is met. First, there is the #if and #endif directives, which specifies a section of code that will only be included if the condition after the #if directive is true. Note that this condition must evaluate to a constant expression.

```
#define DEBUG_LEVEL 3

#if DEBUG_LEVEL > 2
 // ...
#endif
```

Just as with the C++ if statement, any number of #elif (else if) directives and one final #else directive can be included.

```
#if DEBUG_LEVEL > 2
 // ...
#elif DEBUG_LEVEL == 2
 // ...
#else
 // ...
#endif
```

Compile if defined

Sometimes, a section of code should only be compiled if a certain macro has been defined, irrespective of its value. For this purpose two special operators can be used: defined and !defined (not defined).

```
#define DEBUG

#if defined DEBUG
 // ...
#elif !defined DEBUG
 // ...
#endif
```

The same effect can also be achieved using the directives #ifdef and #ifndef respectively. For instance, the #ifdef section is only compiled if the specified macro has been previously defined. Note that a macro is considered defined even if it has not been given a value.

```
#ifdef DEBUG
 // ...
#endif

#ifndef DEBUG
 // ...
#endif
```

Error directive

When the #error directive is encountered the compilation is aborted. This directive can be useful for example to determine whether or not a certain line of code is being compiled. It can optionally take a parameter that specifies the description of the generated compilation error.

```
#error Compilation aborted
```

Line directive

Another standard directive is #line, which can be used to change the line number that is displayed when an error occurs during compilation. Following this directive the line number will as usual be increased by one for each successive line. The directive can take an optional string parameter that sets the filename that will be shown when an error occurs.

```
#line 5 "My MyApp Error"
```

Pragma directive

The last standard directive is #pragma, or pragmatic information. This directive is used to specify options to the compiler. For example, #pragma message can be used to have the compiler output a string to the build window. Another argument for this directive is warning, which changes how the compiler handles warnings.

```
// Show compiler message
#pragma message( "Hello Compiler" )

// Disable warning 4507
#pragma warning(disable : 4507)
```

CHAPTER 25

Exception Handling

Exception handling allows programmers to deal with unexpected situations that may occur in a program.

Throwing exceptions

When a function encounters a situation that it cannot recover from it can generate an exception to signal the caller that the function has failed. This is done using the `throw` keyword followed by whatever it is the method wants to signal. When this statement is reached, the method will stop executing and the exception will propagate up to the caller where it can be caught, using a try-catch statement.

```
int divide(int x, int y)
{
  if (y == 0) throw 0;
  return x / y;
}
```

Try-catch statement

The try-catch statement consists of a try block containing code that may cause exceptions and one or more catch clauses to handle them. In the above case an integer is thrown and so a catch block needs to be included that handles this type of exception. The thrown expression will get passed as an argument to this exception handler, where it can be used to determine what has gone wrong with the function. Note that when the exception has been handled, the execution will then continue running after the try-catch blocks and not after the throw statement.

```
try {
  divide(10,0);
}
catch(int& e) {
  std::cout << "Error code: " << e;
}
```

An exception handler can catch a thrown expression by either value, reference or pointer. However, catching by value should be avoided since this causes an extra copy to be made. Catching by reference is generally preferable. If the code in the try block can throw more types of exceptions then more catch clauses need to be added to handle them as well. Keep in mind that only the handler that matches the thrown expression will be executed.

```
catch(char& e) {
  std::cout << "Error char: " << e;
}
```

To catch all types of exceptions an ellipsis (...) can be used as the parameter of catch. This default handler must be placed as the last catch statement since no handler placed after it will ever be executed.

```
catch(...) { std::cout << "Error"; }
```

Re-throwing exceptions

If an exception handler is not able to recover from an exception it can be re-thrown by using the throw keyword with no argument specified. This will pass the exception up the caller stack until another try-catch block is encountered. Be careful however, because if an exception is never caught the program will terminate with a run-time error.

```
int main()
{
  try {
    try { throw 0; }
    catch(...) { throw; } // re-throw exception
  }
  catch(...) { throw; } // run-time error
}
```

Exception specification

Functions are by default allowed to throw exceptions of any type. To specify the exception types that a function may throw the throw keyword can be appended to the function declaration. The throw keyword is followed by a comma separated list of the allowed types, if any, enclosed in parentheses.

```
void error1() {}          // may throw any exceptions
void error2() throw(...) {} // may throw any exceptions
void error3() throw(int) {} // may only throw int
void error4() throw() {}    // may not throw exceptions
```

This kind of exception specification is very different from the one used in for example Java, and overall there is very little reason to specify exceptions in C++. The compiler will not enforce the specified exceptions in any way and it will not be able to make any optimizations because of them.

Exception class

As previously mentioned, any data type can be thrown in C++. However, the standard library does provide a base class called exception which is specifically designed to declare objects to be thrown. It is defined in the exception header file and is located under the std namespace. As seen below, the class can be constructed with a string that becomes the exception's description.

```
#include <exception>
void make_error()
{
  throw std::exception("My Error Description");
}
```

When catching this exception the object's function what can be used to retrieve the description.

```
try { make_error(); }
catch (std::exception e) {
  std::cout << e.what();
}
```

■ ■ ■

Type Conversions

Converting an expression from one type to another is known as type-conversion. This can be done either implicitly or explicitly.

Implicit conversions

An implicit conversion is performed automatically by the compiler when an expression needs to be converted into one of its compatible types. For example, any conversions between the primitive data types can be done implicitly.

```
long a = 5;   // int implicitly converted to long
double b = a; // long implicitly converted to double
```

These implicit primitive conversions can be further grouped into two kinds: *promotion* and *demotion*. Promotion occurs when an expression gets implicitly converted into a larger type and demotion occurs when converting an expression to a smaller type. Because a demotion can result in the loss of information, these conversions will generate a warning on most compilers. If the potential information loss is intentional, the warning can be suppressed by using an explicit cast.

```
// Promotion
long  a = 5;  // int promoted to long
double b = a;  // long promoted to double

// Demotion
int  c = 10.5; // warning: possible loss of data
bool d = c;    // warning: possible loss of data
```

Explicit conversions

The first explicit cast is the one inherited from C, commonly called the *C-style cast*. The desired data type is simply placed in parentheses to the left of the expression that needs to be converted.

```
int  c = (int)10.5; // double demoted to int
char d = (char)c;   // int demoted to char
```

C++ casts

The C-style cast is suitable for most conversions between the primitive data types. However, when it comes to conversions between classes and pointers it can be too powerful. In order to get greater control over the different types of conversions possible C++ introduced four new casts, called *named casts* or *new-style casts*. These casts are: static, reinterpret, const and dynamic cast.

```
static_cast<new_type> (expression)
reinterpret_cast<new_type> (expression)
const_cast<new_type> (expression)
dynamic_cast<new_type> (expression)
```

As seen above, their format is to follow the cast's name with the new type enclosed in angle brackets and thereafter the expression to be converted in parentheses. These casts allow more precise control over how a conversion should be performed, which in turn makes it easier for the compiler to catch conversion errors. In contrast, the C-style cast includes the static, reinterpret and const cast in one operation. That cast is therefore more likely to execute subtle conversion errors if used incorrectly.

Static cast

The static cast performs conversions between compatible types. It is similar to the C-style cast, but is more restrictive. For example, the C-style cast would allow an integer pointer to point to a char.

```
char c = 10;       // 1 byte
int *p = (int*)&c; // 4 bytes
```

Since this results in a 4-byte pointer pointing to 1 byte of allocated memory, writing to this pointer will either cause a run-time error or will overwrite some adjacent memory.

```
*p = 5; // run-time error: stack corruption
```

In contrast to the C-style cast, the static cast will allow the compiler to check that the pointer and pointee data types are compatible, which allows the programmer to catch this incorrect pointer assignment during compilation.

```
int *q = static_cast<int*>(&c); // compile-time error
```

Reinterpret cast

To force the pointer conversion, in the same way as the C-style cast does in the background, the reinterpret cast would be used instead.

```
int *r = reinterpret_cast<int*>(&c); // forced conversion
```

This cast handles conversions between certain unrelated types, such as from one pointer type to another incompatible pointer type. It will simply perform a binary copy of the data without altering the underlying bit pattern. Note that the result of such a low-level operation is system-specific and therefore not portable. It should be used with caution if it cannot be avoided altogether.

Const cast

The third C++ cast is the const cast. This one is primarily used to add or remove the const modifier of a variable.

```
const int myConst = 5;
int *nonConst = const_cast<int*>(&a); // removes const
```

Although const cast allows the value of a constant to be changed, doing so is still invalid code that may cause a run-time error. This could occur for example if the constant was located in a section of read-only memory.

```
*nonConst = 10; // potential run-time error
```

Const cast is instead used mainly when there is a function that takes a non-constant pointer argument, even though it does not modify the pointee.

```
void print(int *p) { std::cout << *p; }
```

The function can then be passed a constant variable by using a const cast.

```
print(&myConst); // error: cannot convert
                 // const int* to int*

print(nonConst); // allowed
```

C-style and new-style casts

Keep in mind that the C-style cast can also remove the const modifier, but again since it does this conversion behind the scenes the C++ casts are preferable. Another reason to use the C++ casts is that they are easier to find in the source code then the C-style cast. This is important because casting errors can be difficult to discover. A third reason for using the C++ casts is that they are unpleasant to write. Since explicit conversion in many cases can be avoided, this was done intentionally so that programmers would look for a different solution.

Dynamic cast

The fourth and final C++ cast is the dynamic cast. This one is only used to convert object pointers and object references into other pointer or reference types in the inheritance hierarchy. It is the only cast that makes sure that the object pointed to can be converted, by performing a run-time check that the pointer refers to a complete object of the destination type. For this run-time check to be possible the object must be *polymorphic*. That is, the class must define or inherit at least one virtual function. This is because the compiler will only generate the needed run-time type information for such objects.

Dynamic cast examples

In the example below, a MyChild pointer is converted into a MyBase pointer using a dynamic cast. This derived-to-base conversion succeeds, because the Child object includes a complete Base object.

```
class MyBase { public: virtual void test() {} };
class MyChild : public MyBase {};

int main()
{
  MyChild *child = new MyChild();
  MyBase  *base = dynamic_cast<MyBase*>(child); // ok
}
```

The next example attempts to convert a MyBase pointer to a MyChild pointer. Since the Base object does not contain a complete Child object this pointer conversion will fail. To indicate this, the dynamic cast returns a null pointer. This gives a convenient way to check whether or not a conversion has succeeded during run-time.

```
MyBase  *base = new MyBase();
MyChild *child = dynamic_cast<MyChild*>(base);

if (child == 0) std::cout << "Null pointer returned";
```

If a reference is converted instead of a pointer, the dynamic cast will then fail by throwing a bad_cast exception. This needs to be handled using a try-catch statement.

```
#include <exception>
// ...
try { MyChild &child = dynamic_cast<MyChild&>(*base); }
catch(std::bad_cast &e)
{
  std::cout << e.what(); // bad dynamic_cast
}
```

Dynamic or static cast

The advantage of using a dynamic cast is that it allows the programmer to check whether or not a conversion has succeeded during run-time. The disadvantage is that there is a performance overhead associated with doing this check. For this reason using a static cast would have been preferable in the first example, because a derived-to-base conversion will never fail.

```
MyBase *base = static_cast<MyBase*>(child); // ok
```

However, in the second example the conversion may either succeed or fail. It will fail if the MyBase object contains a MyBase instance and it will succeed if it contains a MyChild instance. In some situations this may not be known until run-time. When this is the case dynamic cast is a better choice than static cast.

```
// Succeeds for a MyChild object
MyChild *child = dynamic_cast<MyChild*>(base);
```

If the base-to-derived conversion had been performed using a static cast instead of a dynamic cast the conversion would not have failed. It would have returned a pointer that referred to an incomplete object. Dereferencing such a pointer can lead to run-time errors.

```
// Allowed, but invalid
MyChild *child = static_cast<MyChild*>(base);

// Incomplete MyChild object dereferenced
(*child);
```

CHAPTER 27

■ ■ ■

Templates

Templates provide a way to make a class or function operate with different data types without having to rewrite the code for each type.

Function templates

The example below shows a function that swaps two integer arguments.

```
void swap(int& a, int& b)
{
  int tmp = a;
  a = b;
  b = tmp;
}
```

To convert this method into a function template that can work with any type the first step is to add a *template parameter declaration* before the function. This declaration includes the template keyword followed by the keyword class and the name of the *template parameter*, both enclosed between angle brackets. The name of the template parameter may be anything, but it is common to name it with a capital T.

```
template<class T>
```

Alternatively, the keyword typename can be used instead of class. They are both equivalent in this context.

```
template<typename T>
```

The second step in creating a function template is to replace the data type that will be made generic with the template parameter.

```
template<class T>
void swap(T& a, T& b)
{
  T tmp = a;
  a = b;
  b = tmp;
}
```

Calling function templates

The function template is now complete. To use it `swap` can be called as if it was a regular function, but with the desired template argument specified in angle brackets before the function arguments. Behind the scenes, the compiler will instantiate a new function with this template parameter filled in, and it is this generated function that will be called from this line.

```
int a = 1, b = 2;
swap<int>(a,b); // calls int version of swap
```

Every time the function template is called with a new type, the compiler will instantiate another function using the template.

```
bool c = true, d = false;
swap<bool>(c,d); // calls bool version of swap
```

In this example, the `swap` function template may also be called without specifying the template parameter. This is because the compiler can automatically determine the type, because the function template's arguments use the template type. However, if this is not the case, or if there is a need to force the compiler to select a specific instantiation of the function template, the template parameter would then need to be explicitly specified within angle brackets.

```
int e = 1, f = 2;
swap(e,f); // calls int version of swap
```

Multiple template parameters

Templates can be defined to accept more than one template parameter by adding them between the angle brackets.

```
template<class T, class U>
void swap(T& a, U& b)
{
  T tmp = a;
  a = b;
  b = tmp;
}
```

The second template parameter in the example above allows `swap` to be called with two arguments of different types.

```
int main()
{
  int a = 1;
  long b = 2;
  swap<int, long>(a,b);
}
```

Class templates

Class templates allow class members to use template parameters as types. They are created in the same way as function templates.

```
template<class T>
class myBox
{
 public:
  T a, b;
};
```

Unlike function templates, a class template must always be instantiated with explicitly specified template parameters.

```
myBox<int> box;
```

Another thing to remember when using class templates is that if a method is defined outside of the class template that definition must also be preceded by the template declaration.

```
template<class T>
class myBox
{
 public:
  T a, b;
  void swap();
};

template<class T>
void myBox<T>::swap()
{
  T tmp = a;
  a = b;
  b = tmp;
}
```

Notice that the template parameter is included in the swap template function definition after the class name qualifier. This specifies that the function's template parameter is the same as the template parameter of the class.

Non-type parameters

In addition to type parameters, templates can also have regular function-like parameters. As an example, the `int` template parameter below is used to specify the size of an array.

```
template<class T, int N>
class myBox
{
 public:
  T store[N];
};
```

When this class template is instantiated, both a type and an integer have to be included.

```
myBox<int, 5> box;
```

Default types and values

Class template parameters can be given default values and types.

```
template<class T = int, int N = 5>
```

To use these defaults the angle brackets just need to be left empty when instantiating the class template.

```
myBox<> box;
```

Note that default template parameters may not be used in function templates.

Class template specialization

If there is a need to define a different implementation for a template when a specific type is passed as the template parameter, a *template specialization* can be declared. For example, in the following class template there is a `print` method that outputs the value of a template variable.

```
#include <iostream>

template<class T>
class myBox
{
 public:
  T a;
  void print() { std::cout << a; }
};
```

When the template parameter is a `bool` the method should print out "true" or "false" instead of "1" or "0". One way to do this would be to create a *class template specialization*. A reimplementation of the class template is then created where the template parameter list is empty. Instead, a `bool` specialization parameter is placed after the class template's name and this data type is used instead of the template parameter throughout the implementation.

```
template<>
class myBox<bool>
{
 public:
  bool a;
  void print() { std::cout << (a ? "true" : "false"); }
};
```

When this class template is instantiated with a `bool` template type, this template specialization will be used instead of the standard one.

```
int main()
{
  myBox<bool> box = { true };
  box.print(); // true
}
```

Note that there is no inheritance of members from the standard template to the specialized template. The whole class will have to be redefined.

Function template specialization

Since there is only one function that is different between the templates in the example above, a better alternative would be to create a *function template specialization*. This kind of specialization looks very similar to the class template specialization, but is only applied to a single function instead of the whole class.

```
#include <iostream>

template<class T>
class myBox
{
 public:
  T a;

  template<class T> void print() {
    std::cout << a;
  }
```

```
  template<> void print<bool>() {
    std::cout << (a ? "true" : "false");
  }
};
```

This way only the print method has to be redefined and not the whole class.

```
int main()
{
  myBox<bool> box = { true };
  box.print<bool>(); // true
}
```

Notice that the template parameter has to be specified when the specialized function is invoked. This is not the case with the class template specialization.

CHAPTER 28

■ ■ ■

Headers

When a project grows it is common to split the code up into different source files. When this happens the interface and implementation are generally separated. The interface is placed in a header file, which commonly has the same name as the source file and a .h file extension. This header file contains forward declarations for the source file entities that need to be accessible to other files in the project.

Why to use headers

C++ requires everything to be declared before it can be used. It is not enough to simply compile the source files in the same project. For example, if a function is placed in MyFunc.c, and a second file named MyApp.c in the same project tries to call it, the compiler will report that it cannot find the function.

```
// MyFunc.c
void myFunc()
{
  // ...
}

// MyApp.c
int main()
{
  myFunc(); // error: myFunc identifier not found
}
```

To make this work the function's prototype has to be included in MyApp.c.

```
// MyApp.c
void myFunc(); // prototype

int main()
{
  myFunc();     // ok
}
```

Using headers

This can be made more convenient if the prototype is placed in a header file named MyFunc.h and this header is included in MyApp.c through the use of the `#include` directive. This way if any changes are made to MyFunc there is no need to update the prototypes in MyApp.c. Furthermore, any source file that wants to use the shared code in MyFunc can just include this one header.

```
// MyFunc.h
void myFunc(); // prototype

// MyApp.c
#include "MyFunc.h"
```

What to include in headers

As far as the compiler is concerned there is no difference between a header file and a source file. The distinction is only conceptual. The key idea is that the header should contain the interface of the implementation file – that is, the code that other source files will need to use. This may include shared constants and macros.

```
// MyApp.h - Interface
#define PI 3.14
const double E = 2.72;
```

The header can also contain prototypes of the shared functions defined in the source file.

```
void myFunc();
```

Additionally, shared classes are typically defined in the header, while their methods are implemented in the source file.

```
// MyApp.h
class MyClass
{
 public:
  void myMethod();
};

// MyApp.c
void MyClass::myMethod() {}
```

The header should not include any executable statements, with two important exceptions. First, if a shared class member function or global function is declared as `inline`, that function must be defined in the header. Otherwise, calling the inline function from another source file will give an unresolved external error.

```
// MyApp.h
inline void myFunc() {}

class MyClass
{
 public:
  void myMethod() {}
};
```

The second exception is shared templates. Since templates are not compiled until they are required, the implementation of a shared class or function template must be in the header file together with its declaration.

```
template<class T>
void templateFunction()
{
  // ...
}
```

Include guards

An important thing to bear in mind when using header files is that a code entity may only be defined once in every source file. Consequently, including the same header file more than once will most likely result in compilation errors. To avoid this problem of multiple definitions, a couple of preprocessor directives can be used to create a so called *include guard*. An include guard is created by enclosing the start of the header in a `#ifndef` section that checks for a macro specific to that header file. Only when the macro is not defined is the file included. The macro is then defined, which effectively prevents the file from being included again.

```
// MyApp.h
#ifndef MYAPP
#define MYAPP
// ...
#endif
```

Index

C++ Quick Syntax Reference

Reference

Mikael Olsson

Apress®

C++ Quick Syntax Reference

Contents

About the Author

Mikael Olsson is a professional web entrepreneur, programmer, and author. He works for an R&D company in Finland where he specializes in software development.
In his spare time he writes books and creates websites that summarize various fields of interest. The books he writes are focused on teaching their subject in the most efficient way possible, by explaining only what is relevant and practical without any unnecessary repetition or theory.

About the Technical Reviewer

Michael Thomas has worked in software development for over 20 years as an individual contributor, team lead, program manager, and Vice President of Engineering. Michael has over 10 years experience working with mobile devices. His current focus is in the medical sector using mobile devices to accelerate information transfer between patients and health care providers.