Writing C++

C++ is a very flexible language when it comes to formatting and writing code. It is also a strongly typed language, meaning there are rules about declaring the types of variables, which you can use to your advantage by making the compiler help you write better code. In this section, we will cover how to format C++ code and rules on declaring and scoping variables.

Using white space

Other than string literals, you have free usage of white space (spaces, tabs, newlines), and are able to use as much or as little as you like. C++ statements are delimited by semicolons, so in the following code there are three statements, which will compile and run:

```
int i = 4;
i = i / 2;
std::cout << "The result is" << i << std::endl;
```

The entire code could be written as follows:

```
int i=4;i=i/2; std::cout<<"The result is "<<i<<std::endl;
```

There are some cases where white space is needed (for example, when declaring a variable you must have white space between the type and the variable name), but the convention is to be as judicious as possible to make the code readable. And while it is perfectly correct, language-wise, to put all the statements on one line (like JavaScript), it makes the code almost completely unreadable.

If you are interested in some of the more creative ways of making code unreadable, have a look at the entries for the annual International Obfuscated C Code Contest (http://www.ioccc.org/). As the progenitor of C++, many of the lessons in C shown at IOCCC apply to C++ code too.

Bear in mind that, if the code you write is viable, it may be in use for decades, which means you may have to come back to the code years after you have written it, and it means that other people will support your code, too. Making your code readable is not only a courtesy to other developers, but unreadable code is always a likely target for replacement.

Formatting code

Inevitably, whoever you are writing code for will dictate how you format code. Sometimes it makes sense, for example, if you use some form of preprocessing to extract code and definitions to create documentation for the code. In many cases, the style that is imposed on you is the personal preference of someone else.

Visual C++ allows you to place XML comments in your code. To do this you use a three--slash comment (///) and then compile the source file with the /doc switch. This creates an intermediate XML file called an xdc file with a <doc> root element and containing all the three--slash comments. The Visual C++ documentation defines standard XML tags (for example, <param>, <returns> to document the parameters and return value of a function). The intermediate file is compiled to the final document XML file with the xdcmake utility.

There are two broad styles in C++: **K&R** and **Allman**.

Kernighan and Ritchie (K&R) wrote the first, and most influential, book about C (Dennis Ritchie was the author of the C language). The K&R style is used to describe the formatting style used in that book. In general, K&R places the opening brace of a code block on the same line of the last statement. If your code has nested statements (and typically, it will) then this style can get a bit confusing:

```
if (/* some test */) {
```

```
// the test is true
if (/* some other test */) {
// second test is true
} else {
// second test is false
}
} else {
// the test is false
}
```

This style is typically used in Unix (and Unix-like) code.

The Allman style (named after the developer Eric Allman) places the opening brace on a new line, so the nested example looks as follows:

```
if (/* some test */)
{
// the test is true
if (/* some other test */)
{
// second test is true
}
else
{
// second test is false
}
}
else
{
// the test is false
}
```

The Allman style is typically used by Microsoft.

Remember that your code is unlikely to be presented on paper, so the fact that K&R is more compact will save no trees. If you have the choice, you should choose the style that is the most readable; the decision of this author, for this book, is that Allman is the more readable.

If you have multiple nested blocks, the indents can give you an idea of which block the code resides in. However, comments can help. In particular, if a code block has a large amount of code, it is often helpful to comment the reason for the code block. For example, in an `if` statement, it is helpful to put the result of the test in the code block so you know what the variable values are in that block. It is also useful to put a comment on the closing brace of the test:

```
if (x < 0)
{
// x < 0
/* lots of code */
} // if (x < 0)
else
{
// x >= 0
/* lots of code */
} // if (x < 0)
```

If you put the test as a comment on a closing brace, it means that you have a search term that you can use to find the test that resulted in the code block. The preceding lines make this commenting redundant, but when you have code blocks with many tens of lines of code, and with many levels of nesting, comments like this can be very helpful.

Writing statements

A statement can be a declaration of a variable, an expression that evaluates to a value, or it can be a definition of a type. A statement may also be a control structure to affect the flow of

the execution through your code.
A statement ends with a semicolon. Other than that, there are few rules about how to format statements. You can even use a semicolon on its own, and this is called a null statement. A null statement does nothing, so having too many semicolons is usually benign.

Working with expressions

An expression is a sequence of operators and operands (variables or literals) that results in some value. Consider the following:

```
int i;
i = 6 * 7;
```

On the right side 6 * 7 is an expression, and the assignment (from i on the left-hand side to the semicolon on the right) is a statement.
Every expression is either an **lvalue** or an **rvalue**. You are most likely to see these keywords used in error descriptions. In effect, an lvalue is an expression that refers to some memory location. Items on the left-hand side of an assignment must be lvalues. However, an lvalue can appear on the left- or right-hand side of an assignment. All variables are lvalues. An rvalue is a temporary item that does not exist longer than the expression that uses it; it will have a value, but cannot have a value assigned to it, so it can only exist on the right-hand side of an assignment. Literals are rvalues. The following shows a simple example of lvalues and rvalues:

```
int i;
i = 6 * 7;
```

In the second line, i is an lvalue, and the expression 6 * 7 results in an rvalue (42). The following will not compile because there is an rvalue on the left:

```
6 * 7 = i;
```

Broadly speaking, an expression becomes a statement by when you append a semicolon. For example, the following are both statements:

```
42;
std::sqrt(2);
```

The first line is an rvalue of 42, but since it is temporary it has no effect. A C++ compiler will optimize it away. The second line calls the standard library function to calculate the square root of 2. Again, the result is an rvalue and the value is not used, so the compiler will optimize this away. However, it illustrates that a function can be called without using its return value. Although it is not the case with std::sqrt, many functions have a lasting effect other than their return value. Indeed, the whole point of a function is usually to do something, and the return value is often used merely to indicate if the function was successful; often developers assume that a function will succeed and ignore the return value.

Using the comma operator

Operators will be covered later in this chapter; however, it is useful to introduce the comma operator here. You can have a sequence of expressions separated by a comma as a single statement. For example, the following code is legal in C++:

```
int a = 9;
int b = 4;
int c;
c = a + 8, b + 1;
```

The writer intended to type c = a + 8 / b + 1; and : they pressed comma instead of a /. The intention was for c to be assigned to 9 + 2 + 1, or 12. This code will compile and run, and the variable c will be assigned with a value of 17 (a + 8). The reason is that the comma separates the right-hand side of the assignment into two expressions, a + 8 and b + 1, and it uses the value of the first expression to assign c. Later in this chapter, we will look at operator precedence. However, it is worth saying here that the comma has the lowest precedence and + has a higher precedence than =, so the statement is executed in the order of the addition: the assignment and then the comma operator (with the result of b + 1 thrown away).

You can change the precedence using parentheses to group expressions. For example, the mistyped code could have been as follows:

```
c = (a + 8, b + 1);
```

The result of this statement is: variable c is assigned to 5 (or b + 1). The reason is that with the comma operator expressions are executed from left to right so the value of the group of expressions is the tight-most one. There are some cases, for example, in the initialization or loop expression of a for loop, where you will find the comma operator useful, but as you can see here, even used intentionally, the comma operator produces hard-to-read code.

Using types and variables

Types will be covered in more detail in the next chapter, but it is useful to give basic information here. C++ is a strongly typed language, which means that you have to declare the type of the variables that you use. The reason for this is that the compiler needs to know how much memory to allocate for the variable, and it can determine this by the type of the variable. In addition, the compiler needs to know how to initialize a variable, if it has not been explicitly initialized, and to perform this initialization the compiler needs to know the type of the variable.

C++11 provides the auto keyword, which relaxes this concept of strong typing, and it will be covered in the next chapter. However, the type checking of the compiler is so important that you should use type checking as much as possible.

C++ variables can be declared anywhere in your code as long as they are declared before they are used. *Where* you declare a variable determines *how* you use it (this is called the **scope** of the variable). In general, it is best to declare the variable as close as possible to where you will use it, and within the most restrictive scope. This prevents *name clashes*, where you will have to add additional information to disambiguate two or more variables.

You may, *and should*, give your variables descriptive names. This makes your code much more readable and easier to understand. C++ names must start with an alphabetic character, or an underscore. They can contain alphanumeric characters except spaces, but can contain underscores. So, the following are valid names:

```
numberOfCustomers
NumberOfCustomers
number_of_customers
```

C++ names are case-sensitive, and the first 2,048 characters are significant. You can start a variable name with an underscore, but you cannot use two underscores, nor can you use an underscore followed by a capital letter (these are reserved by C++). C++ also reserves keywords (for example, while and if), and clearly you cannot use type names as variable names, neither built in type names (int, long, and so on) nor your own custom types.

You declare a variable in a statement, ending with a semicolon. The basic syntax of declaring a variable is that you specify the type, then the name, and, optionally, any initialization of the variable.

Built-in types must be initialized before you use them:

```
int i;
i++; // C4700 uninitialized local variable 'i' used
std::cout << i;
```

There are essentially three ways to initialize variables. You can assign a value, you can call the type constructor (constructors for classes will be defined in Chapter 6, *Classes*) or you can initialize a variable using function syntax:

```
int i = 1;
int j = int(2);
int k(3);
```

These three are all legal C++, but stylistically the first is the better because it is more obvious: the variable is an integer, it is called i, and it is assigned a value of 1. The third looks confusing; it looks like the declaration of a function when it is actually declaring a variable. The next chapter will show a variation of assigning a value using the initialization list syntax. The reasons why you will want to do this will be left to that chapter.
Chapter 6, *Classes* will cover classes, your own custom types. A custom type may be defined to have a default value, which means that you may decide not to initialize a variable of a custom type before using it. However, this will result in poorer performance, because the compiler will initialize the variable with the default value and subsequently your code will assign a value, resulting in an assignment being performed twice.

Using constants and literals

Each type will have a literal representation. An integer will be a numeric represented without a decimal point and, if it is a signed integer, the literal can also use the plus or minus symbol to indicate the sign. Similarly, a real number can have a literal value that contains a decimal point, and you may even use the scientific (or engineering) format including an exponent. C++ has various rules to use when specifying literals in code, and these will be covered in the next chapter. Some examples of literals are shown here:
```
int pos = +1;
int neg = -1;
double micro = 1e-6;
double unit = 1.;
std::string name = "Richard";
```
Note that for the unit variable, the compiler knows that the literal is a real number because the value has a decimal point. For integers, you can provide a hexadecimal literal in your code by prefixing the number with 0x, so 0x100 is 256 in decimal. By default, the output stream will print numeric values in base 10; however, you can insert a **manipulator** into an output stream to tell it to use a different number base. The default behavior is std::dec, which means the numbers should be displayed as base 10, std::oct means display as octal (base 8), and std::hex means display as hexadecimal (base 16). If you prefer to see the prefix printed, then you use the stream manipulator std::showbase (more details will be given in Chapter 8, *Using the Standard Library Containers*).
C++ defines some literals. For bool, the logic type, there are true and false constants, where false is zero and true is 1. There is also the nullptr constant, again, zero, which is used as an invalid value for any pointer type.

Defining constants

In some cases, you will want to provide constant values that can be used throughout your code. For example, you may decide to declare a constant for π. You should not allow this value to be changed because it will change the underlying logic in your code. This means that you should mark the variable as being constant. When you do this, the compiler will check the use of the variable and if it is used in code that changes the value of the variable the compiler will issue an error:
```
const double pi = 3.1415;
double radius = 5.0;
double circumference = 2 * pi * radius;
```
In this case the symbol pi is declared as being constant, so it cannot change. If you subsequently decide to change the constant, the compiler will issue an error:
```
// add more precision, generates error C3892
pi += 0.00009265359;
```

Once you have declared a constant, you can be assured that the compiler will make sure it remains so. You can assign a constant with an expression as follows:

```
#include <cmath>
const double sqrtOf2 = std::sqrt(2);
```

In this code, a global constant called sqrtOf2 is declared and assigned with a value using the std::sqrt function. Since this constant is declared outside a function, it is global to the file and can be used throughout the file.
In the last chapter, you learned that one way to declare a constant is to use #define symbols. The problem with this approach is that the preprocessor does a simple replacement. With constants declared with const, the C++ compiler will perform type checking to ensure that the constant is being used appropriately.
You can also use const to declare a constant that will be used as a **constant expression**. For example, you can declare an array using the square bracket syntax (more details will be given in Chapter 4, *Working with Memory, Arrays, and Pointers*):

```
int values[5];
```

This declares an array of five integers on the stack and these items are accessed through the values array variable. The 5 here is a constant expression. When you declare an array on the stack, you have to provide the compiler with a constant expression so it knows how much memory to allocate and this means the size of the array must be known at compile time. (You can allocate an array with a size known only at runtime, but this requires dynamic memory allocation, explained in Chapter 4, *Working with Memory, Arrays, and Pointers.*) In C++, you can declare a constant to do the following:

```
const int size = 5;
int values[size];
```

Elsewhere in your code, when you access the values array, you can use the size constant to make sure that you do not access items past the end of the array. Since the size variable is declared in just one place, if you need to change the size of the array at a later stage, you have just one place to make this change.
The const keyword can also be used on pointers and references (see Chapter 4, *Working with Memory, Arrays, and Pointers*) and on objects (see Chapter 6, *Classes*); often, you'll see it used on parameters to functions (see Chapter 5, *Using Functions*). This is used to get the compiler to help ensure that pointers, references, and objects are used appropriately, as you intended.

Using constant expressions

C++11 introduces a keyword called constexpr. This is applied to an expression, and indicates that the expression should be evaluated at compile type rather than at runtime:

```
constexpr double pi = 3.1415;
constexpr double twopi = 2 * pi;
```

This is similar to initializing a constant declared with the const keyword. However, the constexpr keyword can also be applied to functions that return a value that can be evaluated at compile time, and so this allows the compiler to optimize the code:

```
constexpr int triang(int i)
{
return (i == 0) ? 0 : triang(i - 1) + i;
}
```

In this example, the function triang calculates triangular numbers recursively. The code uses the conditional operator. In the parentheses, the function parameter is tested to see if it is zero, and if so the function returns zero, in effect ending the recursion and returning the function to the original caller. If the parameter is not zero, then the return value is the sum of the parameter and the return value of triang called with the parameter is decremented.

This function, when called with a literal in your code, can be evaluated at compile time. The constexpr is an indication to the compiler to check the usage of the function to see if it can determine the parameter at compile time. If this is the case, the compiler can evaluate the return value and produce code more efficiently than by calling the function at runtime. If the compiler cannot determine the parameter at compile-time, the function will be called as **normal**. A function marked with the constexpr keyword must only have one expression (hence the use of the conditional operator ?: in the triang function).

Using enumerations

A final way to provide constants is to use an enum variable. In effect, an enum is a group of named constants, which means that you can use an enum as a parameter to a function. For example:

```
enum suits {clubs, diamonds, hearts, spades};
```

This defines an enumeration called suits, with named values for the suits in a deck of cards. An enumeration is an integer type and by default the compiler will assume an int, but you can change this by specifying the integer type in the declaration. Since there are just four possible values for card suits, it is a waste of memory to use int (usually 4 bytes) and instead we can use char (a single byte):

```
enum suits : char {clubs, diamonds, hearts, spades};
```

When you use an enumerated value, you can use just the name; however, it is usual to scope it with the name of the enumeration, making the code more readable:

```
suits card1 = diamonds;
suits card2 = suits::diamonds;
```

Both forms are allowed, but the latter makes it more explicit that the value is taken from an enumeration. To force developers to specify the scope, you can apply the keyword class:

```
enum class suits : char {clubs, diamonds, hearts, spades};
```

With this definition and the preceding code, the line declaring card2 will compile, but the line declaring card1 will not. With a scoped enum, the compiler treats the enumeration as a new type and has no inbuilt conversion from your new type to an integer variable. For example:

```
suits card = suits::diamonds;
char c = card + 10; // errors C2784 and C2676
```

The enum type is based on char but when you define the suits variable as being scoped (with class) the second line will not compile. If the enumeration is defined as not being scoped (without class) then there is an inbuilt conversion between the enumerated value and char.

By default, the compiler will give the first enumerator a value of 0 and then increment the value for the subsequent enumerators. Thus suits::diamonds will have a value of 1 because it is the second value in suits. You can assign values yourself:

```
enum ports {ftp=21, ssh, telnet, smtp=25, http=80};
```

In this case, ports::ftp has a value of 21, ports::ssh has a value of 22 (21 incremented), ports::telnet is 22, ports::smtp is 25, and ports::http is 80.

Often the point of enumerations is to provide named symbols within your code and their values are unimportant. Does it matter what value is assigned to suits::hearts? The intention is usually to ensure that it is different from the other values. In other cases, the values are important because they are a way to provide values to other functions.

Enumerations are useful in a switch statement (see later) because the named value makes it clearer than using just an integer. You can also use an enumeration as a parameter to a function and hence restrict the values passed via that parameter:

```
void stack(suits card)
{
// we know that card is only one of four values
}
```

Declaring pointers

Since we are covering the use of variables, it is worth explaining the syntax used to define pointers and arrays because there are some potential pitfalls. Chapter 4, *Working with Memory, Arrays, and Pointers*, covers this in more detail, so we will just introduce the syntax so that you are familiar with it.

In C++, you will access memory using a typed pointer. The type indicates the type of the data that is held in the memory that is pointed to. So, if the pointer is an (4 byte) integer pointer, it will point to four bytes that can be used as an integer. If the integer pointer is incremented, then it will point to the next four bytes, which can be used as an integer.

Don't worry if you find pointers confusing at this point. Chapter 4, *Working with Memory, Arrays, and Pointers*, will explain this in more detail. The purpose of introducing pointers at this time is to make you aware of the syntax.

In C++, pointers are declared using the * symbol and you access a memory address with the & operator:

```
int *p;
int i = 42;
p = &i;
```

The first line declares a variable, p, which will be used to hold the memory address of an integer. The second line declares an integer and assigns it a value. The third line assigns a value to the pointer p to be the address of the integer variable just declared. It is important to stress that the value of p *is not* 42; it will be a memory address where the value of 42 is stored.

Note how the declaration has the * on the variable name. This is common convention. The reason is that if you declare several variables in one statement, the * applies only to the immediate variable. So, for example:

```
int* p1, p2;
```

Initially this looks like you are declaring two integer pointers. However, this line does not do this; it declares just one pointer to integer called p1. The second variable is an integer called p2. The preceding line is equivalent to the following:

```
int *p1;
int p2;
```

If you wish to declare two integers in one statement, then you should do it as follows:

```
int *p1, *p2;
```

Using namespaces

Namespaces give you one mechanism to modularize code. A namespace allows you to label your types, functions, and variables with a unique name so that, using the scope resolution operator, you can give a *fully qualified name*. The advantage is that you know exactly which item will be called. The disadvantage is that using a fully qualified name you are in effect switching off C++'s *argument-dependent lookup* mechanism for overloaded functions where the compiler will choose the function that has the best fit according to the arguments passed to the function.

Defining a namespace is simple: you decorate the types, functions, and global variables with the namespace keyword and the name you give to it. In the following example, two functions are defined in the utilities namespace:

```
namespace utilities
{
bool poll_data()
{
```

```
// code that returns a bool
}
int get_data()
{
// code that returns an integer
}
}
```
Do not use semicolon after the closing bracket.
Now when you use these symbols, you need to qualify the name with the namespace:
```
if (utilities::poll_data())
{
int i = utilities::get_data();
// use i here...
}
```
The namespace declaration may just declare the functions, in which case the actual functions would have to be defined elsewhere, and you will need to use a qualified name:
```
namespace utilities
{
// declare the functions
bool poll_data();
```

```
int get_data();
}
//define the functions
bool utilities::poll_data()
{
// code that returns a bool
}
int utilities::get_data()
{
// code that returns an integer
}
```
One use of namespaces is to version your code. The first version of your code may have a side-effect that is not in your functional specification and is technically a bug, but some callers will use it and depend on it. When you update your code to fix the bug, you may decide to allow your callers the option to use the old version so that their code does not break. You can do this with a namespace:
```
namespace utilities
{
bool poll_data();
int get_data();
namespace V2
{
bool poll_data();
int get_data();
int new_feature();
}
}
```
Now callers who want a specific version can call the fully qualified names, for example, callers could use utilities::V2::poll_data to use the newer version and utilities::poll_data to use the older version. When an item in a specific namespace calls an item in the same namespace, it does not have to use a qualified name. So, if the new_feature function calls get_data, it will be utilities::V2::get_data that is called. It is important to note that, to declare a nested namespace, you have to do the nesting manually (as shown here); you cannot simply declare a namespace called utilities::V2.

The preceding example has been written so that the first version of the code will call it using the namespace utilities. C++11 provides a facility called an **inline** namespace that allows you to define a nested namespace, but allows the compiler to treat the items as being in the parent namespace when it performs an argument-dependent lookup:

```
namespace utilities
{
inline namespace V1
{
bool poll_data();
int get_data();
}
namespace V2
{
bool poll_data();
int get_data();
int new_feature();
}
}
```

Now to call the first version of get_data, you can use utilities::get_data or utilities::V1::get_data.

Fully qualified names can make the code difficult to read, especially if your code will only use one namespace. To help here you have several options. You can place a using statement to indicate that symbols declared in the specified namespace can be used without a fully qualified name:

```
using namespace utilities;
int i = get_data();
int j = V2::get_data();
```

You can still use fully qualified names, but this statement allows you to ease the requirement. Note that a nested namespace is a member of a namespace, so the preceding using statement means that you can call the second version of get_data with either utilities::V2::get_data or V2::get_data. If you use the unqualified name, then it means that you will call utilities::get_data.

A namespace can contain many items, and you may decide that you only want to relax the use of fully qualified names with just a few of them. To do this, use using and give the name of the item:

```
using std::cout;
using std::endl;
cout << "Hello, World!" << endl;
```

This code says that, whenever cout is used, it refers to std::cout. You can use using within a function, or you can put it as file scope and make the intention global to the file.

You do not have to declare a namespace in one place, you can declare it over several files. The following could be in a different file to the previous declaration of utilities:

```
namespace utilities
{
namespace V2
{
void print_data();
}
}
```

The print_data function is still part of the utilities::V2 namespace.

You can also put an #include in a namespace, in which case the items declared in the header file will now be part of the namespace. The standard library header files that have a prefix of c (for example, cmath, cstdlib, and ctime) give access to the C runtime functions by including the appropriate C header in the std namespace.

The great advantage of a namespace is to be able to define your items with names that may be common, but are hidden from other code that does not know the namespace name of. The namespace means that the items are still available to your code via the fully qualified name. However, this only works if you use a unique namespace name, and the likelihood is that, the longer the namespace name, the more unique it is likely to be. Java developers often name their classes using a URI, and you could decide to do the same thing:

```
namespace com_packtpub_richard_grimes
{
int get_data();
}
```

The problem is that the fully qualified name becomes quite long:

```
int i = com_packtpub_richard_grimes::get_data();
```

You can get around this issue using an alias:

```
namespace packtRG = com_packtpub_richard_grimes;
int i = packtRG::get_data();
```

C++ allows you to define a namespace without a name, an **anonymous** namespace. As mentioned previously, namespaces allow you to prevent name clashes between code defined in several files. If you intend to use such a name in only one file you could define a unique namespace name. However, this could get tedious if you had to do it for several files. A namespace without a name has the special meaning that it has **internal linkage**, that is, the items can only be used in the current translation unit, the current file, and not in any other file.

Code that is not declared in a namespace will be a member of the global namespace. You can call the code without a namespace name, but you may want to explicitly indicate that the item is in the global namespace using the scope resolution operator without a namespace name:

```
int version = 42;
void print_version()
{
std::cout << "Version = " << ::version << std::endl;
}
```

C++ scoping of variables

As you saw in the previous chapter the compiler will compile your source files as individual items called **translation units**. The compiler will determine the objects and variables you declare and the types and functions you define, and once declared you can use any of these in the subsequent code within the scope of the declaration. At its very broadest, you can declare an item at the global scope by declaring it in a header file that will be used by all of the source files in your project. If you do not use a namespace it is often wise when you use such global variables to name them as being part of the global namespace:

```
// in version.h
extern int version;
// in version.cpp
#include "version.h"
version = 17;
// print.cpp
#include "version.h"
void print_version()
{
std::cout << "Version = " << ::version << std::endl;
}
```

This code has the C++ for two source files (version.cpp and print.cpp) and a header file (version.h) included by both source files. The header file declares the global variable

version, which can be used by both source files; it declares the variable, but does not define it. The actual variable is defined and initialized in version.cpp; it is here that the compiler will allocate memory for the variable. The extern keyword used on the declaration in the header indicates to the compiler that version has **external linkage**, that is, the name is visible in files other than where the variable is defined. The version variable is used in the print.cpp source file. In this file, the scope resolution operator (::) is used without a namespace name and hence indicates that the variable version is in the global namespace.
You can also declare items that will only be used within the current translation unit, by declaring them within the source file before they are used (usually at the top of the file). This produces a level of modularity and allows you to hide implementation details from code in other source files. For example:

```
// in print.h
void usage();
// print.cpp
#include "version.h"
std::string app_name = "My Utility";
void print_version()
{
std::cout << "Version = " << ::version << std::endl;
}
void usage()
{
std::cout << app_name << " ";
print_version();
}
```

The print.h header contains the interface for the code in the file print.cpp. Only those functions declared in the header will be callable by other source files. The caller does not need to know about the implementation of the usage function, and as you can see here it is implemented using a call to a function called print_version that is only available to code in print.cpp. The variable app_name is declared at file scope, so it will only be accessible to code in print.cpp.
If another source file declares a variable at file scope, that is called app_name, and is also a std::string the file will compile, but the linker will complain when it tries to link the object files. The reason is that the linker will see the same variable defined in two places and it will not know which one to use.

A function also defines a scope; variables defined within the function can only be accessed through that name. The parameters of the function are also included as variables within the function, so when you declare other variables, you have to use different names. If a parameter is not marked as const then you can alter the value of the parameter in your function.
You can declare variables anywhere within a function as long as you declare them before you use them. Curly braces ({}) are used to define code blocks, and they also define local scope; if you declare a variable within a code block then you can only use it there. This means that you can declare variables with the same name outside the code block and the compiler will use the variable closest to the scope it is accessed.
Before finishing this section, it is important to mention one aspect of the C++ **storage class**. A variable declared in a function means that the compiler will allocate memory for the variable on the stack frame created for the function. When the function finishes, the stack frame is torn down and the memory recycled. This means that, after a function returns, the values in any local variables are lost; when the function is called again, the variable is created anew and initialized again.
C++ provides the static keyword to change this behavior. The static keyword means

that the variable is allocated when the program starts just like variables declared at global scope. Applying static to a variable declared in a function means that the variable has internal linkage, that is, the compiler restricts access to that variable to that function:

```
int inc(int i)
{
static int value;
value += i;
return value;
}
int main()
{
std::cout << inc(10) << std::endl;
std::cout << inc(5) << std::endl;
}
```

By default, the compiler will initialize a static variable to 0, but you can provide an initialization value, and this will be used when the variable is first allocated. When this program starts, the value variable will be initialized to 0 before the main function is called. The first time the inc function is called, the value variable is incremented to 10, which is returned by the function and printed to the console. When the inc function returns the value variable is retained, so that when the inc function is called again, the value variable is incremented by 5 to a value of 15.

Using operators

Operators are used to compute a value from one or more operands. The following table groups all of the operators with equal *precedence* and lists their *associativity*. The higher in the table, the higher precedence of execution the operator has in an expression. If you have several operators in an expression, the compiler will perform the higher--precedence operators before the lower--precedence operators. If an expression contains operators of equal precedence, then the compiler will use the associativity to decide whether an operand is grouped with the operator to its left or right.

There are some ambiguities in this table. A pair of parentheses can mean a function call or a cast and in the table these are listed as function() and cast(); in your code you will simply use (). The + and - symbols are either used to indicate sign (unary plus and unary minus, given in the table as +x and -x), or addition and subtraction (given in the table as + and -). The & symbol means either "take the address of" (listed in the table as &x) or bitwise AND (listed in the table as &). Finally, the postfix increment and decrement operators (listed in the table as x++ and x--) have a higher precedence than the prefix equivalents (listed as ++x and --x).

Precedence and Associativity Operators

1: No associativity ::
2: Left to right associativity . or -> [] function() {} x++ x-- typeid const_cast dynamic_cast reinterpret_cast static_cast
3: Right to left associativity sizeof ++x --x ~ ! -x +x &x * new delete cast()
4: Left to right associativity .* or ->*
5: Left to right associativity * / %
6: Left to right associativity + -
7: Left to right associativity << >>
8: Left to right associativity < > <= >=
9: Left to right associativity == !=

10: Left to right associativity &

11: Left to right associativity ^
12: Left to right associativity |
13: Left to right associativity &&
14: Left to right associativity ||
15: Right to left associativity ? :
16: Right to left associativity = *= /= %= += -= <<= >>= &= |= ^=
17: Right to left associativity throw
18: Left to right associativity ,

For example, take a look at the following code:

```
int a = b + c * d;
```

This is interpreted as the multiplication being performed first, and then the addition. A clearer way to write the same code is:

```
int a = b + (c * d);
```

The reason is that * has a higher precedence than + so that the multiplication is carried out first, and then the addition is performed

```
int a = b + c + d;
```

In this case, the + operators have the same precedence, which is higher than the precedence of assignment. Since + has left to right associativity the statement is interpreted as follows:

```
int a = ((b + c) + d);
```

That is, the first action is the addition of b and c, and the result is added to d and it is this result that is used to assign a. This may not seem important, but bear in mind that the addition could be between function calls (a function call has a higher precedence than +):

```
int a = b() + c() + d();
```

This means that the three functions are called in the order b, c, d, and then their return values are summed according to the left-to-right associativity. This may be important because d may depend on global data altered by the other two functions.

It makes your code more readable and easier to understand if you explicitly specify the precedence by grouping expressions with parentheses. Writing b + (c * d) makes it immediately clear which expression is executed first, whereas b + c * d means you have to know the precedence of each operator.

The built-in operators are overloaded, that is, the same syntax is used regardless of which built-in type is used for the operands. The operands must be the same type; if different types are used, the compiler will perform some default conversions, but in other cases (in particular, when operating on types of different sizes), you will have to perform a cast to indicate explicitly what you mean. The next chapter will explain this in more detail.

Exploring the built-in operators

C++ comes with a wide range of built-in operators; most are arithmetic or logic operators, which will be covered in this section. The casting operators will be covered in the next chapter; memory operators will be covered in Chapter 4, *Working with Memory, Arrays, and Pointers,* and the object-related operators in Chapter 6, *Classes*.

Arithmetic operators

The arithmetic operators +, -, /, *, and % need little explanation other than perhaps the division and modulus operators. All of these operators act upon integer and real numeric types except for %, which can only be used with integer types. If you mix the types (say, add an integer to a floating-point number) then the compiler will perform an automatic conversion, as explained in the next chapter. The division operator / behaves as you expect for floating point variables: it produces the result of the division of the two operands. When you perform the division between two integers a / b, the result is the whole number of the divisor (b) in the dividend (a). The remainder of the division is obtained by the modulus %. So, for any integer, b (other than zero), one could say that, an integer a can be expressed as follows:

```
(a / b) * b + (a % b)
```

Note that the modulus operator can only be used with integers. If you want to get the remainder of a floating-point division, use the standard function, std:;remainder.

Be careful when using division with integers, since fractional parts are discarded. If you need the fractional parts, then you may need to explicitly convert the numbers into real numbers. For example:

```
int height = 480;
int width = 640;
float aspect_ratio = width / height;
```

This gives an aspect ratio of 1 when it should be 1.3333 (or 4 : 3). To ensure that floating-point division is performed, rather than integer division, you can cast either (or both) the dividend or divisor to a floating-point number as explained in the next chapter.

Increment and decrement operators

There are two versions of these operators, prefix and postfix. As the name suggests, prefix means that the operator is placed on the left of the operand (for example, ++i), and a postfix operator is placed to the right (i++). The ++ operator will increment the operand and the -- operator will decrement it. The prefix operator means "return the value *after* the operation," and the postfix operator means "return the value *before* the operation." So the following code will increment one variable and use it to assign another:

```
a = ++b;
```

Here, the prefix operator is used so the variable b is incremented and the variable a is assigned to the value after b has been incremented. Another way of expressing this is:

```
a = (b = b + 1);
```

The following code assigns a value using the postfix operator:

```
a = b++;
```

This means that the variable b is incremented, but the variable a is assigned to the value before b has been incremented. Another way of expressing this is:

```
int t;
a = (t = b, b = b + 1, t);
```

Note that this statement uses the comma operator, so a is assigned to the temporary variable t in the right-most expression.

The increment and decrement operators can be applied to both integer and floating point numbers. The operators can also be applied to pointers, where they have a special meaning. When you increment a pointer variable it means *increment the pointer by the size of the type pointed to by the operator*.

Bitwise operators

Integers can be regarded as a series of bits, 0 or 1. Bitwise operators act upon these bits compared to the bit in the same position in the other operand. Signed integers use a bit to indicate the sign, but bitwise operators act on every bit in an integer, so it is usually only sensible to use them on unsigned integers. In the following, all the types are marked as unsigned, so they are treated as not having a sign bit.

The & operator is bitwise AND, which means that each bit in the left-hand operand is compared with the bit in the right-hand operand in the same position. If both are 1, the resultant bit in the same position will be 1; otherwise, the resultant bit is zero:

```
unsigned int a = 0x0a0a; // this is the binary 0000101000001010
unsigned int b = 0x00ff; // this is the binary 0000000000001111
unsigned int c = a & b; // this is the binary 0000000000001010
std::cout << std::hex << std::showbase << c << std::endl;
```

In this example, using bitwise & with 0x00ff has the same effect as providing a mask that masks out all but the lowest byte.

The bitwise OR operator | will return a value of 1 if either or both bits in the same position are 1, and a value of 0 only if both are 0:

```
unsigned int a = 0x0a0a; // this is the binary 0000101000001010
unsigned int b = 0x00ff; // this is the binary 0000000000001111
unsigned int c = a & b; // this is the binary 0000101000001111
std::cout << std::hex << std::showbase << c << std::endl;
```

One use of the & operator is to find if a particular bit (or a specific collection of bits) is set:

```
unsigned int flags = 0x0a0a; // 0000101000001010
unsigned int test = 0x00ff; // 0000000000001111
// 0000101000001111 is (flags & test)
if ((flags & test) == flags)
{
// code for when all the flags bits are set in test
}
if ((flags & test) != 0)
{
```

```
// code for when some or all the flag bits are set in test
}
```

The flags variable has the bits we require, and the test variable is a value that we are examining. The value (flags & test) will have only those bits in the test variables that are also set in flags. Thus, if the result is non-zero, it means that at least one bit in test is also set in flags; if the result is exactly the same as the flags variable then all the bits in flags are set in test.

The exclusive OR operator ^ is used to test when the bits are different; the resultant bit is 1 if the bits in the operands are different, and 0 if they are the same. Exclusive OR can be used to flip specific bits:

```
int value = 0xf1;
int flags = 0x02;
int result = value ^ flags; // 0xf3
std::cout << std::hex << result << std::endl;
```

The final bitwise operator is the bitwise complement ~. This operator is applied to a single integer operand and returns a value where every bit is the complement of the corresponding bit in the operand; so if the operand bit is 1, the bit in the result is 0, and if the bit in the operand is 0, the bit in the result is 1. Note that all bits are examined, so you need to be aware of the size of the integer.

Boolean operators

The == operator tests whether two values are exactly the same. If you test two integers then the test is obvious; for example, if x is 2 and y is 3, then x == y is obviously false.

However, two real numbers may not be the same even when you think so:

```
double x = 1.000001 * 1000000000000;
double y = 1000001000000;
if (x == y) std::cout << "numbers are the same";
```

The double type is a floating-point type held in 8 bytes, but this is not enough for the precision being used here; the value stored in the x variable is 1000000999999.9999 (to four decimal places).

The != operator tests if two values are not true. The operators > and <, test two values to see if the left-hand operand is greater than, or less than, the right-hand operand, the >=

operator tests if the left-hand operand is greater than or equal to the right-hand operand, and the <= operator tests if the left-hand operand is less than or equal to the right-hand operand. These operators can be used in the if statement similarly to how == is used in the preceding example. The expressions using the operators return a value of type bool and so you can use them to assign values to Boolean variables:

```
int x = 10;
int y = 11;
bool b = (x > y);
if (b) std::cout << "numbers same";
else std::cout << "numbers not same";
```

The assignment operator (=) has a higher precedence than the greater than (>=) operator, but we have used the parentheses to make it explicit that the value is tested before being used to assign the variable. You can use the ! operator to negate a logical value. So, using the value of b obtained previously, you can write the following:

```
if (!b) std::cout << "numbers not same";
else std::cout << "numbers same";
```

You can combine two logical expressions using the && (AND) and || (OR) operators. An expression with the && operator is true only if both operands are true, whereas an expression with the || operator is true if either, or both, operands are true:

```
int x = 10, y = 10, z = 9;
if ((x == y) || (y < z))
std::cout << "one or both are true";
```

This code involves three tests; the first tests if the x and y variables have the same value, the second tests if the variable y is less than z, and then there is a test to see if either or both of the first two tests are true.

In a || expression such as this, where the first operand (x==y) is true, the total logical expression will be true regardless of the value of the right operand (here, y < z). So there is no point in testing the second expression. Correspondingly, in an && expression, if the first operand is false then the entire expression must be false, and so the right-hand part of the expression need not be tested. The compiler will provide code to perform this *shortcircuiting* for you:

```
if ((x != 0) && (0.5 > 1/x))
{
// reciprocal is less than 0.5
```

```
}
```

This code tests to see if the reciprocal of x is less than 0.5 (or, conversely, that x is greater than 2). If the x variable has value 0 then the test 1/x is an error but, in this case, the expression will never be executed because the left operand to && is false.

Bitwise shift operators

Bitwise shift operators shift the bits in the left-hand operand integer the specified number of bits given in the right-hand operand, in the specified direction. A shift by one bit left multiplies the number by two, a shift one bit to the right divides by 2. In the following a 2-byte integer is bit-shifted:

```
unsigned short s1 = 0x0010;
unsigned short s2 = s1 << 8;
std::cout << std::hex << std::showbase;
std::cout << s2 << std::endl;
// 0x1000
s2 = s2 << 3;
std::cout << s2 << std::endl;
// 0x8000
```

In this example, the s1 variable has the fifth bit set (0x0010 or 16). The s2 variable has this value, shifted left by 8 bits, so the single bit is shifted to the 13th bit, and the bottom 8 bits are all set to 0 (0x10000 or 4,096). This means that 0x0010 has been multiplied by 2^8, or 256,

to give 0x1000. Next, the value is shifted left by another 3 bits, and the result is 0x8000; the top bit is set.
The operator discards any bits that overflow, so if you have the top bit set and shift the integer one bit left, that top bit will be discarded:

```
s2 = s2 << 1;
std::cout << s2 << std::endl;
// 0
```

A final shift left by one bit results in a value 0.
It is important to remember that, when used with a stream, the operator << means *insert into the stream*, and when used with integers, it means *bitwise shift*.

Assignment operators

The assignment operator = assigns an lvalue (a variable) on the left with the result of the rvalue (a variable or expression) on the right:

```
int x = 10;
x = x + 10;
```

The first line declares an integer and initializes it to 10. The second line alters the variable by adding another 10 to it, so now the variable x has a value of 20. This is assignment. C++ allows you to change the value of a variable based on the variable's value using an abbreviated syntax. The previous lines can be written as follows:

```
int x = 10;
x += 10;
```

An increment operator such as this (and the decrement operator) can be applied to integers and floating-point types. If the operator is applied to a pointer, then the operand indicates how many whole item addresses the pointer is changed by. For example, if an int is 4 bytes and you add 10 to an int pointer, the actual pointer value is incremented by 40 (10 times 4 bytes).
In addition to the increment (+=) and decrement (-=) assignments, you can have assignments for multiply (*=), divide (/=), and remainder (%=). All of these except for the last one (%=) can be used for both floating-point types and integers. The remainder assignment can only be used on integers.
You can also perform bitwise assignment operations on integers: left shift (<<=), right shift (>>=), bitwise AND (&=), bitwise OR (|=), and bitwise exclusive OR (^=). It usually only makes sense to apply these to unsigned integers. So, multiplying by eight can be carried out by both of these two lines:

```
i *= 8;
i <<= 3;
```

Controlling execution flow

C++ provides many ways to test values and loop through code.

Using conditional statements

The most frequently used conditional statement is if. In its simplest form, the if statement takes a logical expression in a pair of parentheses and is immediately followed by the statement that is executed if the condition is true:

```
int i;
std::cin >> i;
if (i > 10) std::cout << "much too high!" << std::endl;
```

You can also use the else statement to catch occasions when the condition is false:

```
int i;
std::cin >> i;
if (i > 10) std::cout << "much too high!" << std::endl;
```

```
else std::cout << "within range" << std::endl;
```

If you want to execute several statements, you can use braces ({}) to define a code block. The condition is a logical expression and C++ will convert from numeric types to a bool, where 0 is false and anything not 0 is true. If you are not careful, this can be a source of an error that is not only difficult to notice, but also can have an unexpected side-effect. Consider the following code, which asks for input from the console and then tests to see if the user enters -1:

```
int i;
std::cin >> i;
if (i == -1) std::cout << "typed -1" << endl;
std::cout << "i = " << i << endl;
```

This is contrived, but you may be asking for values in a loop and then performing actions on those values, except when the user enters -1, at which point the loop finishes. If you mistype, you may end up with the following code:

```
int i;
std::cin >> i;
if (i = -1) std::cout << "typed -1" << endl;
std::cout << "i = " << i << endl;
```

In this case, the assignment operator (=) is used instead of the *equality* operator (==). There is just one character difference, but this code is still correct C++ and the compiler is happy to compile it.

The result is that, regardless of what you type at the console, the variable i is assigned to -1, and since -1 is not zero, the condition in the if statement is true, hence the true clause of the statement is executed. Since the variable has been assigned to -1, this may alter logic further on in your code. The way to avoid this bug is to take advantage of the requirement that in an assignment the left-hand side must be an lvalue. Perform your test as follows:

```
if (-1 == i) std::cout << "typed -1" << endl;
```

Here, the logical expression is (-1 == i), and since the == operator is commutative (the order of the operands does not matter; you get the same result), this is exactly the same as you intended in the preceding test. However, if you mistype the operator, you get the following:

```
if (-1 = i) std::cout << "typed -1" << endl;
```

In this case, the assignment has an rvalue on the left-hand side, and this will cause the compiler to issue an error (in Visual C++ this is C2106 '=' : left operand must be l-value).

You are allowed to declare a variable in an if statement, and the scope of the variable is in the statement blocks. For example, a function that returns an integer can be called as follows:

```
if (int i = getValue()) {
// i != 0 // can use i here
} else {
// i == 0 // can use i here
}
```

While this is perfectly legal C++, there are few reasons why you would want to do this.

In some cases, the conditional operator ?: can be used instead of an if statement. The operator executes the expression to the left of the ? operator and, if the conditional expression is true, it executes the expression to the right of the ?. If the conditional expression is false, it executes the expression to the right of the :. The expression that the operator executes provides the return value of the conditional operator.

For example, the following code determines the maximum of two variables, a and b:

```
int max;
if (a > b) max = a;
else max = b;
```

This can be expressed with the following single statement:

```
int max = (a > b) ? a : b;
```

The main choice is which ever is most readable in the code. Clearly, if the assignment expressions are large it may well be best to split them over lines in an if statement. However, it is useful to use the conditional statement in other statements. For example:

```
int number;
std::cin >> number;
std::cout << "there "
<< ((number == 1) ? "is " : "are ")
<< number << " item"
<< ((number == 1) ? "" : "s")
<< std::endl;
```

This code determines if the variable number is 1 and if so it prints on the console there is 1 item. This is because in both conditionals, if the value of the number variable is 1, the test is true and the first expression is used. Note that there is a pair of parentheses around the entire operator. The reason is that the stream << operator is overloaded, and you want the compiler to choose the version that takes a string, which is the type returned by the operator rather than bool, which is the type of the expression (number == 1).

If the value returned by the conditional operator is an lvalue then you can use it on the lefthand side of an assignment. This means that you can write the following, rather odd, code:

```
int i = 10, j = 0;
((i < j) ? i : j) = 7;
// i is 10, j is 7
i = 0, j = 10;
((i < j) ? i : j) = 7;
// i is 7, j is 10
```

The conditional operator checks to see if i is less than j and if so it assigns a value to i; otherwise, it assigns j with that value. This code is terse, but it lacks readability. It is far better in this case to use an if statement.

Selecting

If you want to test to see if a variable is one of several values, using multiple if statements becomes cumbersome. The C++ switch statement fulfills this purpose much better. The basic syntax is shown here:

```
int i;
std::cin >> i;
switch(i)
{
```

```
case 1:
std::cout << "one" << std::endl;
break;
case 2:
std::cout << "two" << std::endl;
break;
default:
std::cout << "other" << std::endl;
}
```

Each case is essentially a label as to the specific code to be run if the selected variable is the specified value. The default clause is for values where there exists no case. You do not have to have a default clause, which means that you are testing only for specified cases. The default clause could be for the most common case (in which case, the cases filter out the less likely values) or it could be for exceptional values (in which case, the cases handle the most likely values).

A switch statement can only test integer types (which includes enum), and you can only test for constants. The char type is an integer, and this means that you can use characters in the case items, but only individual characters; you cannot use strings:

```
char c;
std::cin >> c;
switch(c)
{
case 'a':
std::cout << "character a" << std::endl;
break;
case 'z':
std::cout << "character z" << std::endl;
break;
default:
std::cout << "other character" << std::endl;
}
```

The break statement indicates the end of the statements executed for a case. If you do not specify it, execution will *fall through* and the following case statements will be executed even though they have been specified for a different case:

```
switch(i)
{
case 1:
std::cout << "one" << std::endl;
// fall thru
case 2:
std::cout << "less than three" << std::endl;
```

```
break;
case 3:
std::cout << "three" << std::endl;
break;
case 4:
break;
default:
std::cout << "other" << std::endl;
}
```

This code shows the importance of the break statement. A value of 1 will print both one and less than three to the console, because execution *falls through* to the preceding case, even though that case is for another value.
It is usual to have different code for different cases, so you will most often finish a case with break. It is easy to miss out a break by mistake, and this will lead to unusual behavior. It is good practice to document your code when deliberately missing out the break statement so that you know that if a break is missing, it is likely to be a mistake.
You can provide zero or more statements for each case. If there is more than one statement, they are all executed for that specific case. If you provide no statements (as for case 4 in this example) then it means that no statements will be executed, not even those in the default clause.
The break statement means *break out of this code block*, and it behaves like this in the loop statements while and for as well. There are other ways that you can break out of a switch. A case could call return to finish the function where the switch is declared; it can call goto to jump to a label, or it can call throw to throw an exception that will be caught by an exception handler outside the switch, or even outside the function.
So far, the cases are in numeric order. This is not a requirement, but it does make the code more readable, and clearly, if you want to *fall through* the case statements (as in case 1 here), you should pay attention to the order the case items.
If you need to declare a temporary variable in a case handler then you must define a code block using braces, and this will make the scope of the variable localized to just that code

block. You can, of course, use any variable declared outside of the switch statement in any of the case handlers.

Since enumerated constants are integers, you can test an enum in a switch statement:

```
enum suits { clubs, diamonds, hearts, spades };
void print_name(suits card)
{
switch(card)
{
case suits::clubs:
std::cout << "card is a club";
break;
default:
std::cout << "card is not a club";
}
}
```

Although the enum here is not scoped (it is neither enum class nor enum struct), it is not required to specify the scope of the value in the case, but it makes the code more obvious what the constant refers to.

Looping

Most programs will need to loop through some code. C++ provides several ways to do this, either by iterating with an indexed value or testing a logical condition.

Looping with iteration

There are two versions of the for statement, iteration and range-based. The latter was introduced in C++11. The iteration version has the following format:

```
for (init_expression; condition; loop_expression)
loop_statement;
```

You can provide one or more loop statements, and for more than one statement, you should provide a code block using braces. The purpose of the loop may be served by the loop expression, in which case you may not want a loop statement to be executed; here, you use the null statement, ; which means *do nothing*.

Within the parentheses are three expressions separated by semicolons. The first expression allows you to declare and initialize a loop variable. This variable is scoped to the for statement, so you can only use it in the for expressions or in the loop statements that follow. If you want more than one loop variable, you can declare them in this expression using the comma operator.

The for statement will loop while the condition expression is true; so if you are using a loop variable, you can use this expression to check the value of the loop variable. The third expression is called at the end of the loop, after the loop statement has been called; following this, the condition expression is called to see if the loop should continue. This final expression is often used to update the value of the loop variable. For example:

```
for (int i = 0; i < 10; ++i)
{
std::cout << i;
}
```

In this code, the loop variable is i and it is initialized to zero. Next, the condition is checked, and since i will be less than 10, the statement will be executed (printing the value to the console). The next action is the loop expression; ++i, is called, which increments the loop variable, i, and then the condition is checked, and so on. Since the condition is i < 10, this means that this loop will run ten times with a value of i between 0 and 9 (so you will see **0123456789** on the console).

The loop expression can be any expression you like, but often it increments or decrements a value. You do not have to change the loop variable value by 1; for example, you can use i -= 5 as the loop expression to decrease the variable by 5 on each loop. The loop variable can be any type you like; it does not have to be integer, it does not even have to be numeric (for example, it could be a pointer, or an **iterator object** described in Chapter 8, *Using the Standard Library Containers*), and the condition and loop expression do not have to use the loop variable. In fact, you do not have to declare a loop variable at all!

If you do not provide a loop condition then the loop will be infinite, unless you provide a check in the loop:

```
for (int i = 0; ; ++i)
{
std::cout << i << std::endl;
if (i == 10) break;
}
```

This uses the break statement introduced earlier with the switch statement. It indicates that execution exits the for loop, and you can also use return, goto, or throw. You will rarely see a statement that finishes using goto; however, you may see the following:

```
for (;;)
{
// code
}
```

In this case, there is no loop variable, no loop expression, and no conditional. This is an everlasting loop, and the code within the loop determines when the loop finishes.

The third expression in the for statement, the loop expression, can be anything you like; the only property is that it is executed at the end of a loop. You may choose to change another variable in this expression, or you can even provide several expressions separated by the comma operator. For example, if you have two functions, one called poll_data that returns true if there is more data available and false when there is no more data, and a function called get_data that returns the next available data item, you could use for as follows (bear in mind; this is a contrived example, to make a point):

```
for (int i = -1; poll_data(); i = get_data())
{
if (i != -1) std::cout << i << std::endl;
}
```

When poll_data returns a false value, the loop will end. The if statement is needed because the first time the loop is called, get_data has not yet been called. A better version is as follows:

```
for (; poll_data() ;)
{
int i = get_data();
std::cout << i << std::endl;
}
```

Keep this example in mind for the following section.

There is one other keyword that you can use in a for loop. In many cases, your for loop will have many lines of code and at some point, you may decide that the current loop has completed and you want to start the next loop (or, more specifically, execute the loop expression and then test the condition). To do this, you can call continue:

```
for (float divisor = 0.f; divisor < 10.f; ++divisor)
{
std::cout << divisor;
if (divisor == 0)
{
std::cout << std::endl;
continue;
}
std::cout << " " << (1 / divisor) << std::endl;
}
```

In this code, we print the reciprocal of the numbers 0 to 9 (0.f is a 4-byte floating-point literal). The first line in the for loop prints the loop variable, and the next line checks to see if the variable is zero. If it is, it prints a new line and continues, that is, the last line in the for loop is not executed. The reason is that the last line prints the reciprocal and it would be an error to divide any number by zero.

C++11 introduces another way to use the for loop, which is intended to be used with containers. The C++ standard library contains **templates** for container classes. These classes contain collections of objects, and provide access to those items in a standard way. The standard way is to iterate through collections using an **iterator** object. More details about how to do this will be given in Chapter 8, *Using the Standard Library Containers*; the syntax requires an understanding of pointers and iterators, so we will not cover them here. The range-based for loop gives a simple mechanism to access items in a container without explicitly using iterators.

The syntax is simple:
```
for (for_declaration : expression) loop_statement;
```
The first thing to point out is that there are only two expressions and they are separated by a colon (:). The first expression is used to declare the loop variable, which is of the type of the items in the collection being iterated through. The second expression gives access to the collection.

In C++ terms, the collections that can be used are those that define a begin and end function that gives access to iterators, and also to stack-based arrays (that the compiler knows the size of).

The Standard Library defines a container object called a vector. The vector template is a class that contains items of the type specified in the angle brackets (<>); in the following code, the vector is initialized in a special way that is new to C++11, called **list initialization**. This syntax allows you to specify the initial values of the vector in a list between curly braces. The following code creates and initializes a vector, and then uses an iteration for loop to print out all the values:
```
using namespace std;
vector<string> beatles = { "John", "Paul", "George", "Ringo" };
for (int i = 0; i < beatles.size(); ++i)
{
cout << beatles.at(i) << endl;
}
```

Here a using statement is used so that the classes vector and string do not have to be used with fully qualified names.

The vector class has a member function called size (called through the . operator, which means "call this function on this object") that returns the number of items in the vector. Each item is accessed using the at function passing the item's index. The one big problem with this code is that it uses random access, that is, it accesses each item using its index. This is a property of vector, but other Standard Library container types do not have random access. The following uses the range-based for:
```
vector<string> beatles = { "John", "Paul", "George", "Ringo" };
for (string musician : beatles)
{
cout << musician << endl;
}
```
This syntax works with any of the standard container types and for arrays allocated on the stack:
```
int birth_years[] = { 1940, 1942, 1943, 1940 };
```

```
for (int birth_year : birth_years)
{
cout << birth_year << endl;
}
```

In this case, the compiler knows the size of the array (because the compiler has allocated the array) and so it can determine the range. The range-based for loop will iterate through all the items in the container, but as with the previous version you can leave the for loop using break, return, throw, or goto, and you can indicate that the next loop should be executed using the continue statement.

Conditional loops

In the previous section we gave a contrived example, where the condition in the for loop polled for data:

```
for (; poll_data() ;)
{
int i = get_data();
std::cout << i << std::endl;
}
```

In this example, there is no loop variable used in the condition. This is a candidate for the while conditional loop:

```
while (poll_data())
{
int i = get_data();
std::cout << i << std::endl;
}
```

The statement will continue to loop until the expression (poll_data in this case) has a value of false. As with for, you can exit the while loop with break, return, throw, or goto, and you can indicate that the next loop should be executed using the continue statement.

The first time the while statement is called, the condition is tested before the loop is executed; in some cases you may want the loop executed at least once, and then test the condition (most likely dependent upon the action in the loop) to see if the loop should be repeated. The way to do this is to use the do-while loop:

```
int i = 5;
do
{
std::cout << i-- << std::endl;
} while (i > 0);
```

Note the semicolon after the while clause. This is required.

This loop will print 5 to 1 in reverse order. The reason is that the loop starts with i initialized to 5. The statement in the loop decrements the variable through a postfix operator, which means the value before the decrement is passed to the stream. At the end of the loop, the while clause tests to see if the variable is greater than zero. If this test is true, the loop is repeated. When the loop is called with i assigned to 1, the value of 1 is printed to the console and the variable decremented to zero, and the while clause will test an expression that is false and the looping will finish.

The difference between the two types of loop is that the condition is tested before the loop is executed in the while loop, and so the loop may not be executed. In a do-while loop, the condition is called after the loop, which means that, with a do-while loop, the loop statements are always called at least once.

Jumping

C++ supports jumps, and in most cases, there are better ways to branch code; however, for completeness, we will cover the mechanism here. There are two parts to a jump: a labeled statement to jump to and the goto statement. A label has the same naming rules as a variable; it is declared suffixed with a colon, and it must be before a statement. The goto statement is called using the label's name:

```
int main()
{
for (int i = 0; i < 10; ++i)
{
std::cout << i << std::endl;
if (i == 5) goto end;
}
end:
std::cout << "end";
}
```

The label must be in the same function as the calling goto.

Jumps are rarely used, because they encourage you to write non-structured code. However, if you have a routine with highly nested loops or if statements, it may make more sense and be more readable to use a goto to jump to clean up code.

Using C++ language features

Let's now use the features you have learned in this chapter to write an application. This example is a simple command-line calculator; you type an expression such as *6 * 7*, and the application parses the input and performs the calculation.

Start Visual C++ and click the **File** menu, and then **New**, and finally, click on the **File...** option to get the **New File** dialog. In the left-hand pane, click on **Visual C++**, and in the middle pane, click on **C++ File (.cpp),** and then click on the **Open** button. Before you do anything else, save this file. Using a Visual C++ console (a command line, which has the Visual C++ environment), navigate to the Beginning_C++ folder you created in the previous chapter and create a new folder called Chapter_02. Now, in Visual C++, on the **File** menu, click **Save Source1.cpp As...** and in the **Save File As** dialog locate the Chapter_02 folder you just created. In the **File name** box, type **calc.cpp** and click on the **Save** button.

The application will use std::cout and std::string; so at the top of the file, add the headers that define these and, so that you do not have to use fully qualified names, add a using statement:

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

You will pass the expression via the command-line, so add a main function that takes command line parameters at the bottom of the file:

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

The application handles expressions in the form arg1 op arg2 where op is an operator and arg1 and arg2 are the arguments. This means that, when the application is called, it must have four parameters; the first is the command used to start the application and the last three are the expression. The first code in the main function should ensure that the right number of parameters is provided, so at the top of this function add a condition, as follows:

```
if (argc != 4)
{
usage();
return 1;
}
```

If the command is called with more or less than four parameters, a function usage is called, and then the main function returns, stopping the application.

Add the usage function before the main function, as follows:

```
void usage()
{
cout << endl;
cout << "calc arg1 op arg2" << endl;
cout << "arg1 and arg2 are the arguments" << endl;
cout << "op is an operator, one of + - / or *" << endl;
}
```

This simply explains how to use the command and explains the parameters. At this point, you can compile the application. Since you are using the C++ Standard Library, you will need to compile with support for C++ exceptions, so type the following at the commandline:

```
C:\Beginning_C++Chapter_02\cl /EHsc calc.cpp
```

If you typed in the code without any mistakes, the file should compile. If you get any errors from the compiler, check the source file to see if the code is exactly as given in the preceding code. You may get the following error:

```
'cl' is not recognized as an internal or external command,
operable program or batch file.
```

This means that the console is not set up with the Visual C++ environment, so either close it down and start the console via the Windows Start menu, or run the **vcvarsall.bat** batch file. The steps to do both of these were given in the previous chapter.

Once the code has compiled you may run it. Start by running it with the correct number of parameters (for example, calc 6 * 7), and then try it with an incorrect number of parameters (for example, calc 6 * 7 / 3). Note that the space between the parameters is important:

```
C:\Beginning_C++Chapter_02>calc 6 * 7
C:\Beginning_C++Chapter_02>calc 6 * 7 / 3
calc arg1 op arg2
arg1 and arg2 are the arguments
op is an operator, one of + - / or *
```

In the first case, the application does nothing, so all you see is a blank line. In the second example, the code has determined that there are not enough parameters, and so it prints the usage information to the console.

Next, you need to do some simple parsing of the parameters to check that the user has passed valid values. At the bottom of the main function, add the following:

```
string opArg = argv[2];
if (opArg.length() > 1)
{
cout << endl << "operator should be a single character" << endl;
usage();
return 1;
}
```

The first line initializes a C++ std::string object with the third command-line parameter, which should be the operator in the expression. This simple example only allows a single character for the operator, so the subsequent lines check to make sure that the operator is a single character. The C++ std::string class has a member function called length that returns the number of characters in the string.

The argv[2] parameter will have a length of at least one character (a parameter with no length will not be treated as a command-line parameter!), so we have to check if the user typed an operator longer than one character.

Next you need to test to ensure that the parameter is one of the restricted set allowed and, if the user types another operator, print an error and stop the processing. At the bottom of the main function, add the following:

```
char op = opArg.at(0);
if (op == 44 || op == 46 || op < 42 || op > 47)
{
cout << endl << "operator not recognized" << endl;
usage();
return 1;
}
```

The tests are going to be made on a character, so you need to extract this character from the string object. This code uses the at function, which is passed the index of the character you need. (Chapter 8, *Using the Standard Library Containers*, will give more details about the members of the std::string class.) The next line checks to see if the character is not supported. The code relies on the following values for the characters that we support:

Character	**Value**
+	42
*	43
-	45
/	47

As you can see, if the character is less than 42 or greater than 47 it will be incorrect, but between 42 and 47 there are two characters that we also want to reject: , (44) and . (46). This is why we have the preceding conditional: "if the character is less than 42 or greater than 47, or it is 44 or 46, then reject it."

The char data type is an integer, which is why the test uses integer literals. You could have used character literals, so the following change is just as valid:

```
if (op == ',' || op == '.' || op < '+' || op > '/')
{
cout << endl << "operator not recognized" << endl;
usage();
return 1;
}
```

You should use whichever you find the most readable. Since it makes less sense to check whether one character is *greater than* another, this book will use the former.

At this point, you can compile the code and test it. First try with an operator that is more than one character (for example, **) and confirm that you get the message that the operator should be a single character. Secondly, test with a character that is not a recognized operator; try any character other than +, *, -, or /, but it is also worth trying . and ,.

Bear in mind that the command prompt has special actions for some symbols, such as "&" and "|", and the command prompt may give you an error from it by parsing the commandline before even calling your code.

The next thing to do is to convert the arguments into a form that the code can use. The command-line parameters are passed to the program in an array of strings; however, we are interpreting some of those parameters as floating-point numbers (in fact, double-precision floating-point numbers). The C runtime provides a function called atof, which is available through the C++ Standard Library (in this case, <iostream> includes files that include <cmath>, where atof is declared).

It is a bit counter-intuitive to get access to a math function such as atof through including a file associated with stream input and output. If this makes you uneasy, you can add a line after the include lines to include the <cmath> file. As mentioned in the previous chapter, the C++ Standard Library headers have been written to ensure that a header file is only included once, so including <cmath> twice has no ill effect. This was not done in the preceding code, because it was argued that atof is a string function and the code includes the <string> header and, indeed, <cmath> is included via the files the <string> header includes.

Add the following lines to the bottom of the main function. The first two lines convert the second and fourth parameters (remember, C++ arrays are zero-based indexed) to double values. The final line declares a variable to hold the result:

```
double arg1 = atof(argv[1]);
double arg2 = atof(argv[3]);
double result = 0;
```

Now we need to determine which operator was passed and perform the requested action. We will do this with a switch statement. We know that the op variable will be valid, and so we do not have to provide a default clause to catch the values we have not tested for. Add a switch statement to the bottom of the function:

```
double arg1 = atof(argv[1]);
double arg2 = atof(argv[3]);
double result = 0;
switch(op)
{
}
```

The first three cases, +, -, and *, are straightforward:

```
switch (op)
{
case '+':
result = arg1 + arg2;
break;
case '-':
result = arg1 - arg2;
break;
case '*':
result = arg1 * arg2;
break;
}
```

Again, since char is an integer, you can use it in a switch statement, but C++ allows you to check for the character values. In this case, using characters rather than numbers makes the code much more readable.

After the switch, add the final code to print out the result:

```
cout << endl;
cout << arg1 << " " << op << " " << arg2;
cout << " = " << result << endl;
```

You can now compile the code and test it with calculations that involve +, -, and *.

Division is a problem, because it is invalid to divide by zero. To test this out, add the following lines to the bottom of the switch:

```
case '/':
result = arg1 / arg2;
break;
```

Compile and run the code, passing zero as the final parameter:

```
C:\Beginning_C++Chapter_02>calc 1 / 0
1 / 0 = inf
```

The code ran successfully, and printed out the expression, but it says that the result is an odd value of inf. What is happening here?

The division by zero assigned result to a value of NAN, which is a constant defined in <math.h> (included via <cmath>), and means "not a number." The double overload of the insertion operator for the cout object tests to see if the number has a valid value, and if the number has a value of NAN, it prints the string **inf**. In our application, we can test for a zero divisor, and we treat the user action of passing a zero as being an error. Thus, change the code so that it reads as follows:

```
case '/':
if (arg2 == 0) {
cout << endl << "divide by zero!" << endl;
return 1;
} else {
result = arg1 / arg2;
}
break;
```

Now when the user passes zero as a divisor, you will get a divide by zero! message.
You can now compile the full example and test it out. The application supports floatingpoint arithmetic using the +, -, *, and / operators, and will handle the case of dividing by zero.

Exploring built-in types

C++ provides integer, floating point, and Boolean types. The char type is an integer but it can be used to hold individual characters and so its data can be viewed as a number or as a character. The C++ Standard Library provides the string class to allow you to use and manipulate strings of characters. Strings will be covered in depth in Chapter 9, *Using Strings*.

As the name suggests, integer types contain integral values where there are no fractional parts. If you perform calculations with integers you should expect that any fractional parts will be discarded unless you take steps to retain them (for example, through the remainder operator %). Floating point types hold numbers that may have a fractional part; because floating point types can hold numbers in a mantissa exponent format, they can hold exceptionally large or exceptionally small numbers.
A variable is an instance of a type; it is the memory allocated to hold the data that the type can hold. Integer and floating point variable declarations can be modified to tell the compiler how much memory to allocate, and thus the limits of the data that the variable can hold and the precision of the calculations performed on the variable. In addition, you can also indicate if the variable will hold a number where the sign is important. If the number is being used to hold bitmaps (where the bits do not make up a number, but have a separate meaning of their own) then it usually makes no sense to use a signed type.
In some cases, you will be using C++ to unpack data from a file or a network stream so that you can manipulate it. In this case, you will need to know whether the data is a floating point or integral, signed or unsigned, how many bytes are used and what order those bytes will be in. The order of the bytes (whether the first byte in a multi-byte number is the low or high part of the number) is determined by the processor which you are compiling for, and in most cases, you will not need to worry about it.
Similarly, sometimes you may need to know about the size of a variable and how it is aligned in memory; in particular, when you are using records of data, known in C++ as structs. C++ provides the sizeof operator to give the number of bytes used to hold a variable and the alignof operator to determine the alignment of the type in memory. For basic types, the sizeof and alignof operators return the same value; it is only necessary to call the alignof operator on custom types where it will return the alignment of the largest data member in the type.

Integers

As the name suggests, an integer holds integral data, numbers that have no fractional part. For this reason, it makes little sense to do any arithmetic with an integer where the fractional part is important; in this case, you should use floating point numbers. An example of this was shown in the last chapter:

```
int height = 480;
int width = 640;
int aspect_ratio = width / height;
```

This gives an aspect ratio of 1, which is clearly untrue and serves no purpose. Even if you assign the result to a floating-point number, you will get the same result:

```
float aspect_ratio = width / height;
```

The reason is that the arithmetic is performed in the expression `width / height`, which will use the division operator for integers that will throw away any fractional part of the result. To use the floating-point division operator you will have to cast one or other of the operands to a floating-point number so the floating-point operator is used:

```
float aspect_ratio = width / (float)height;
```

This will assign a value of 1.3333 (or 4 : 3) to the `aspect_ratio` variable. The cast operator used here is the C cast operator, which forces data of one type to be used as data of another type. (It is used because we have not yet introduced the C++ cast operators, and the syntax of C cast operators is clear.) There is no type safety in this cast. C++ provides cast operators, which are discussed in the following text, some of which will cast in a type-safe way, which becomes important when you use pointers to objects of custom types.

C++ provides integer types of various sizes, as summarized in the following table. These are the five standard integer types. The standard says that an `int` is the natural size of the processor and will have a value between (and including) `INT_MIN` and `INT_MAX` (defined in the `<climits>` header file). The size of the integer type has *at least as much storage as those preceding it in the list*, so an `int` is, at least, as big as a `short int` and a `long long int` types, which is at least as big as a `long int` type. The phrase *at least as big as* is not much use if the types are all the same size, so the `<climits>` header file defines ranges for the other fundamental integer types too. It is implementation-specific how many bytes are needed to store these integer ranges. This table gives the ranges of the fundamental types and the sizes on x86, 32-bit processors:

Type	**Range**	**Size in bytes**
`signed char`	-128 to 127	1
`short int`	-32768 to 32767	2
`int`	-2147483648 to 2147483647	4
`long int`	-2147483648 to 2147483647	4
`long long int`	-9223372036854775808 to 9223372036854775807	8

In practice, rather than the `short int` type, you will use `short`; for `long int`, you will use `long`; and for `long long int`, you will typically use `long long`. As you can see from this table, the `int` and `long int` types are the same size, but they are still two different types.

Other than the `char` type, by default integer types are signed, that is, they can hold negative as well as positive numbers (for example, a variable of type `short` can have a value between -32,768 and 32,767). You can use the `signed` keyword to explicitly indicate that the type is signed. You can also have unsigned equivalents by using the `unsigned` keyword, which will give you an extra bit, but will also mean that bitwise operators and shift operators will work as you expect. You may find `unsigned` used without a type, in which case it refers to `unsigned int`. Similarly, `signed` used without a type refers to `signed int`.

The `char` type is a separate type to both `unsigned char` and `signed char`. The standard says that every bit in a char is used to hold character information, and so it is implementation-dependent as to whether a char can be treated as being able to hold negative numbers. If you want a char to hold a signed number, you should specifically use signed char.

The standard is imprecise about the size of the standard integer types and this may be an issue if you are writing code (for example, accessing data in a file, or a network stream) that contains a stream of bytes. The <cstdlib> header file defines named types that will hold specific ranges of data. These types have names which have the number of bits used in the range (although the actual type may require more bits). So, there are types with names such as int16_t and uint16_t, where the first type is a signed integer that will hold a range of 16-bit values and the second type is an unsigned integer. There are also types declared for 8-, 32-, and 64-bit values.

The following shows the actual sizes of these types determined by the sizeof operator on an x86 machine:

```
// #include <cstdint>
using namespace std; // Values for x86
cout << sizeof(int8_t) << endl; // 1
cout << sizeof(int16_t) << endl; // 2
cout << sizeof(int32_t) << endl; // 4
cout << sizeof(int64_t) << endl; // 8
```

In addition, the <cstdlib> header file defines types with names such as int_least16_t and uint_least16_t using the same naming scheme as before, and with versions for 8-, 16-,32-, and 64-bits. The least part of the name means that the type will hold values with at least the specified number of bits, but there could be more. There are also types with names such as int_fast16_t and uint_fast16_t with versions for 8-, 16-, 32-, and 64-bits which are regarded as the fastest types that can hold that number of bits.

Specifying integer literals

To assign a value to an integer variable you provide a number that has no fractional part. The compiler will identify the type with the nearest precision that the number represents and attempt to assign the integer, performing a conversion if necessary.

To explicitly specify that a literal is a long value, you use the l or L suffix. Similarly, for an unsigned long, you use the suffix ul or UL. For long long values, you use ll or LL suffix, and use ull or ULL for unsigned long long. The u (or U) suffix is for unsigned (that is, unsigned int) and you do not need a suffix for int. The following illustrates this, using uppercase suffixes:

```
int i = 3;
signed s = 3;
unsigned int ui = 3U;
long l = 3L;
unsigned long ul = 3UL;
long long ll = 3LL;
unsigned long long ull = 3ULL;
```

Using a 10-based number system to specify a number that is a bitmap is confusing and cumbersome. The bits in a bitmap are the powers of 2, so it makes more sense to use a number system that is a power of 2. C++ allows you to provide numbers in octal (base 8) or hexadecimal (base 16). To provide a literal in an octal you prefix the number with a zero character (0). To provide a literal in a hexadecimal you prefix the number with the 0x character sequence. Octal numbers use the digits 0 through 7, but hexadecimal numbers need 16 digits, which means 0 through 9 and a through f (or A through F), where A is 10 in base 10 and F is 15 in base 10:

```
unsigned long long every_other = 0xAAAAAAAAAAAAAAAA;
unsigned long long each_other = 0x5555555555555555;
cout << hex << showbase << uppercase;
cout << every_other << endl;
cout << each_other << endl;
```

In this code, two 64-bit (in Visual C++) integers are assigned bitmap values, with every other bit set to 1. The first variable starts with the bottom bit set, the second variable starts with the bottom bit unset, and the second lowest bit set. Before inserting the numbers, the stream is modified with three manipulators. The first hex indicates that integers should be printed on the console as hexadecimals, and showbase means that the leading 0x will be printed. By default, the alphabetic digits (A to F) will be given in lowercase and, to specify that uppercase must be used, you use uppercase. Once the stream has been modified, the setting remains until it is changed. To change the stream subsequently to use lowercase for alphabetic hexadecimal digits you insert nouppercase into the stream and to print the number without the base, insert the noshowbase manipulator. To use octal digits, you insert the oct manipulator and to use decimals, insert the dec manipulator.

When you specify large numbers like this, it becomes difficult to see if you have specified the right number of digits. You can group together digits using a single quote ('):

```
unsigned long long every_other = 0xAAAA'AAAA'AAAA'AAAA;
int billion = 1'000'000'000;
```

The compiler ignores the quote; it is just used as a visual aid. In the first example, the quote groups the digits into two byte groups; in the second case the quote groups a decimal number in thousands and millions.

Using bitset to show bit patterns

There is no manipulator to tell the cout object to print an integer as a bitmap, but you can simulate the behavior using a bitset object:

```
// #include <bitset>
unsigned long long every_other = 0xAAAAAAAAAAAAAAAA;
unsigned long long each_other = 0x5555555555555555;
bitset<64> bs_every(every_other);
bitset<64> bs_each(each_other);
cout << bs_every << endl;
cout << bs_each << endl;
```

The result is:

```
1010101010101010101010101010101010101010101010101010101010101010
0101010101010101010101010101010101010101010101010101010101010101
```

Here the bitset class is **parameterized,** which means that you provide a parameter through the angle brackets (<>) and in this case, 64 is used, indicating that the bitset object will accommodate 64-bits. In both cases, the initialization of the bitset object is carried out using a syntax that looks like a function call (in fact, it does call a function called a **constructor**) and this is the preferred way to initialize an object. Inserting the bitset object into the stream, prints out each bit starting with the highest bit. (The reason for this is that there is an operator << function defined, which takes a bitset object, as is the case for most of the standard library classes).

The bitset class is useful for accessing and setting individual bits as an alternative to using bitwise operators:

```
bs_every.set(0);
every_other = bs_every.to_ullong();
cout << bs_every << endl;
cout << every_other << endl;
```

The set function will set the bit in the specified position to a value of 1. The to_ullong function will return a long long number that the bitset represents.

The call to the set function and the assignment has the same result as the following:

```
every_other |= 0x0000000000000001;
```

Determining integer byte order

The order of bytes in an integer is implementation-dependent; it depends on how the processor handles integers. In most cases, you do not need to know. However, if you are reading bytes from a file in binary mode, or bytes from a network stream, and you need to interpret two or more bytes as parts of an integer, you will need to know what order they are in, and if necessary convert them to the order recognized by the processor.

The C network library (on Windows, it is called the **Winsock** library) contains a collection of functions that convert unsigned short and unsigned long types from the network order to the host order (that is, the order used by the processor on the current machine) and vice versa. The network order is big endian. **Big endian** means that the first byte will be the highest byte in the integer, whereas **little endian** means that the first byte is the smallest byte. When you transmit an integer to another machine you first convert from the order used by the processor of the source machine (the host order) to the network order and the receiving machine converts the integer from network order to the host order of the receiving machine before using the data.

The functions to change the byte order are ntohs and ntohl; for converting unsigned short and unsigned long from the network order to the host order and htons and htonl for the conversion from the host order to the network order. Knowing the byte order will be important when you view memory when you are debugging code (for example, as in Chapter 10, *Diagnostics and Debugging*).

It is easy to write code to reverse the byte order:

```
unsigned short reverse(unsigned short us)
{
return ((us & 0xff) << 8) | ((us & 0xff00) >> 8);
}
```

This uses bitwise operators to separate the two bytes that are assumed to make up the unsigned short into the lower byte, which is shifted eight bits left, and the upper byte that is shifted eight bits right and these two numbers are recombined as an unsigned short using the bitwise OR operator, |. It is simple to write versions of this function for 4-byte and 8-byte integers.

Floating point types

There are three basic floating point types:

float (single precision)
double (double precision)
long double (extended precision)

All of these are signed. The actual format of the number in memory, and the number of bytes used, is specific to the C++ implementation, but the <cfloat> header file gives the ranges. The following table gives the positive ranges and number of bytes used on x86, 32-bit processors:

Type	**Range**	**Size in bytes**
float	1.175494351e-38 to 3.402823466e+38	4
double	2.2250738585072014e-308 to 1.7976931348623158e+308	8
long double	2.2250738585072014e-308 to 1.7976931348623158e+308	8

As you can see, in Visual C++ double and long double have the same ranges, but they are still two distinct types.

Specifying floating point literals

A literal used to initialize a double is specified as a floating point by using either the scientific format, or simply by providing a decimal point:

```
double one = 1.0;
double two = 2.;
double one_million = 1e6;
```

The first example indicates that the variable one is assigned to a floating-point value of 1.0. The trailing zero is not important, as shown in the second variable, two; however, the trailing zero does make the code more readable since the period is easy to overlook. The third example uses scientific notation. The first part is the mantissa and can be signed and the part after the e is the exponent. The exponent is the power-of-10 magnitude of the number (which can be negative). The variable is assigned to a value of the mantissa multiplied by 10 and raised to the exponent. Although it is not recommended, you can write the following:

```
double one = 0.0001e4;
double one_billion = 1000e6;
```

The compiler will interpret the numbers appropriately. The first example is perverse, but the second makes some sense; it shows in your code that a billion is a thousand million. These examples assign double precision floating point values to double variables. To specify a value for single precision variables so that you can assign a float variable, use the f (or F) suffix. Similarly, for a long double literal use the l (or L) suffix:

```
float one = 1.f;
float two = 2f; // error
long double one_million = 1e6L;
```

If you use these suffixes, you still have to provide the number in the right format. A literal of 2f is incorrect; you have to provide a decimal point, 2.f. When you specify floating point numbers with a large number of digits you can use the single quote (') to group digits. As stated before, this is just a visual aid to the programmer:

```
double one_billion = 1'000'000'000.;
```

Characters and strings

The string class and C string functions will be covered in Chapter 9, *Using String*; this section covers the basic use of character variables in your code.

Character types

The char type is an integer, so signed char and unsigned char exist too. These are three distinct types; the signed char and unsigned char types should be treated as numeric types. The char type is used to hold a single character in the implementation's character set. In Visual C++, this is an eight-bit integer that can hold characters from the ISO-8859 or UTF-8 character set. These character sets are able to represent the characters used in English and most European languages. Characters from other languages take up more than one byte and C++ provides the char16_t type to hold 16-bit characters and char32_t to hold 32-bit characters.

There is also a type called wchar_t (wide character) that will be able to hold characters from the largest extended character set. In general, when you see a C Runtime Library or C++ Standard Library function with a w prefix, it will use wide character strings rather than char strings. So, the cout object will allow you to insert char strings and the wcout object will allow you to insert wide character strings.

The C++ standard says that every bit in a char is used to hold character information, and so it is implementation-dependent as to whether a char can be treated as being able to hold negative numbers. The following illustrates this:

```
char c = '~';
cout << c << " " << (signed short)c << endl;
c += 2;
cout << c << " " << (signed short)c << endl;
```

The range of a signed char is -128 to 127, but this code uses the separate type char and attempts to use it in the same way. The variable c is first assigned to the ASCII character ~

(126). When you insert a character into an output stream it will attempt to print a character rather than a number, so the next line prints this character to the console, to get the numeric value the code converts the variable to a signed short integer. (Again, a C cast is used for clarity.) Next, the variable is incremented by two, that is, the character is two characters further in the character set, which means the first character in the extended ASCII character set; the result is this:

```
~ 126
C -128
```

The first character in the extended character set is C-cedilla.
It is rather counter-intuitive that a value of 126 incremented by two results is a value of -128, and this comes about from overflow calculations with signed types. Even if this is intentional, it is best to avoid doing this.

In Visual C++, the C-cedilla character is treated as -128 so you can write the following to have the same effect:

```
char c = -128;
```

This is implementation-specific, so for portable code you should not rely upon it.

Using character macros

The <cctype> header contains various macros that you can use to examine the type of character a char holds. These are the C runtime macros declared in <ctype.h>. Some of the more useful macros to test character values are explained in the following table. Bear in mind that, since these are C routines, they will not return bool values; instead they return an int with a value of non-zero for true and zero for false.

Macro	**Tests if the character is:**
isalnum	An alphanumeric character, A to Z, a to z, 0 to 9
isalpha	An alphabetic character, A to Z, a to z
isascii	An ASCII character, 0x00 to 0x7f
isblank	A space or horizontal tab
iscntrl	A control character, 0x00 to 0x1f or 0x7f
isdigit	A decimal digit 0 to 9
isgraph	A printable character other than space, 0x21 to 0x7e
islower	A lowercase character, a to z
isprint	A printable character, 0x20 to 0x7e
ispunct	A punctuation character, ! " # $ % & ' ( ) * + , - . / : ; < = > ? @ [ ] ^ _ ` { | } ~ \
isspace	A space
isupper	An uppercase character, A to Z
isxdigit	A hexadecimal digit, 0 to 9, a to f, A to F

For example, the following code loops while reading in a single character from the input stream (after each character, you need to press the *Enter* key). The loop finishes when a nonnumeric
value is provided:

```
char c;
do
{
cin >> c
} while(isdigit(c));
```

There are also macros to change characters. Again, these will return an int value, which you should convert to a char.

Macro	**Returns**
toupper	The uppercase version of the character
tolower	The lowercase version of the character

In the following code, the character typed at the console is echoed back until the user types q or Q. If the character typed is a lowercase character, the echoed character is converted to uppercase:

```
char c;
do
{
cin >> c;
if (islower(c)) c = toupper(c);
cout << c << endl;
} while (c != 'Q');
```

Specifying character literals

You can initialize a char variable with literal characters. This will be a character from the supported character set. The ASCII character set includes some unprintable characters, and so that you can use these, C++ provides two character sequences using the backslash character (\).

Name	ASCII name	C++ sequence
Newline	LF	\n
Horizontal tab	HT	\t
Vertical tab	VT	\v
Backspace	BS	\b
Carriage return	CR	\r
Form feed	FF	\f
Alert	BEL	\a
Backslash	\	\\
Question mark	?	\?
Single quote	'	\'
Double quote	"	\"

In addition, you can give the numeric value of that character as an octal or hexadecimal number. To provide an octal number you give the number as three characters (prefixed with one or two 0 characters if necessary) prefixed with a backslash. For a hexadecimal number, you prefix it with \x. The character M, is the character number 77 in decimal, 115 in octal, and 4d in hexadecimal, so you can initialize a character variable with an M character in three ways:

```
char m1 = 'M';
char m2 = '\115';
char m3 = '\x4d';
```

For completeness, it is worth pointing out that you can initialize a char as an integer, so the following will also initialize each variable to an M character:

```
char m4 = 0115; // octal
char m5 = 0x4d; // hexadecimal
```

All of these methods are valid.

Specifying string literals

Strings are made up of one or more characters, and you can use the escaped characters in string literals too:

```
cout << "This is \x43\x2b\05\3n";
```

This rather unreadable string will be printed on the console as This is C++ followed by a newline. The capital C is 43 in hexadecimal and the + symbol is 2b in hexadecimal and 53 in octal. The \n character is a newline. Escaped characters are useful for printing characters that are not in the character set your C++ compiler uses and for some unprintable characters (for example, \t to insert a horizontal tab). The cout object buffers the characters before writing them to the output stream. If you use \n as a newline it is treated like any other

character in the buffer. The endl manipulator will insert \n into the buffer and then flush it so the characters are immediately written to the console.
The *empty*, or NULL character, is a \0. This is an important character because it is unprintable, and it has no use other than to mark the end of a sequence of characters in a string. The empty string is "", but since strings are delimited by the NULL character the memory taken up by a string variable initialized with the empty string will have one character, that is \0.
The newline character allows you to put a newline within a string. This is useful if the only formatting you'll do is with paragraphs and you are printing short paragraphs:

```
cout << "Mary had a little lamb,n its fleece was white as snow."
<< endl;
```

This prints two lines on the console:

```
Mary had a little lamb,
its fleece was white as snow.
```

However, you may want to initialize a string with a long sequence of characters and the limitations of the editor you are using may mean you want to split the string over several lines. You do this by putting each fragment of the string within double quotes:

```
cout << "And everywhere that Mary went, "
"the lamb was sure to go."
<< endl;
```

You will see the following on the console:

```
And everywhere that Mary went, the lamb was sure to go.
```

There is no newline printed other than the one explicitly requested at the end with endl.
This syntax allows you to make long strings more readable in your code; you can, of course, use the newline characters \n, in such strings.

Unicode literals

A wchar_t variable can also be initialized with a character and the compiler will promote the character to a wide character essentially by using the character's byte and assigning the remaining (higher) bytes to zero. However, it makes more sense to assign such a variable with a wide character, and you do this using the L prefix.

```
wchar_t dollar = L'$';
wchar_t euro = L'\u20a0';
wcout << dollar;
```

Notice that, rather than using the cout object, this code uses wcout, the wide character version. The syntax of using the \u prefix within the quotes indicates that the following character is a Unicode character.
Bear in mind that, to show the Unicode character, you need to use a console that will show Unicode characters and, by default, the Windows console is set to **Code Page 850** which will not show Unicode characters. You can change the mode of the output console by calling _setmode (defined in <io.h>) on the standard output stream, stdout, specifying the UTF-16 file mode (using _O_U16TEXT defined in <fcntl.h>):

```
_setmode(_fileno(stdout), _O_U16TEXT);
```

You can find a list of all of the characters supported by Unicode at h t t p ://u n i c o d e . o r g /c h a r t s /.
UTF-16 characters can also be assigned to char16_t variables, and UTF-32 characters can be assigned to char32_t variables.

Raw strings

When you use a raw string literal you essentially switch off the meaning of escape characters. Whatever you type into a raw string becomes its content, even if you use whitespace including newlines. The raw string is delimited with R"(and)". That is, the string is between the inner parentheses.

```
cout << R"(newline is \n in C++ and "quoted text" use quotes)";
```

Note that, the () is part of the syntax and is not part of the string. The preceding code prints the following to the console:

```
newline is \n in C++ and "quoted text" use quotes
```

Normally in a string `\n` is an escaped character and will be translated as a newline, but in a raw string it is not translated and is printed as two characters.

In a normal C++ string, you will have to escape some of the characters; for example, the double quote will have to be escaped to `\"` and the backslash escaped to `\\`. The following will give the same result without using a raw string:

```
cout << "newline is \\n in C++ and \"quoted text\" use quotes";
```

You can also have newlines in raw strings:

```
cout << R"(Mary had a little lamb,
its fleece was white as snow)"
cout << endl;
```

In this code, the newline after the comma will be printed to the console. Unfortunately, all whitespace will be printed on the console, so assuming that in the preceding code the indentation is three spaces and the `cout` is indented once, you will see the following on the console:

```
Mary had a little lamb,
its fleece was white as snow
```

There are 14 spaces in front of `its` because there were 14 spaces in front of `its` in the source code. For this reason, you should be wary about using raw strings.

Perhaps, the best use for raw strings is to initialize variables with file paths on Windows. The folder separation character in Windows is a backslash, which means that for a literal string that represents a file path you will have to escape each of these separators; thus, the string will have a lot of double backslashes, with the possibility of missing one. With raw strings this escaping is not necessary. The two string variables in the following represent the same string:

```
string path1 = "C:\\Beginning_C++\\Chapter_03\\readme.txt";
string path2 = R"(C:\Beginning_C++\Chapter_03\readme.txt)";
```

These two strings have the same contents, but the second one is more readable since the C++ literal string does not have escaped backslashes.

The requirement for escaping backslashes is only needed for literal strings declared in your code; it is an indication to the compiler of how to interpret the character. If you obtain a file path from a function (or via `argv[0]`), the separator will be backslashes.

String byte order

Extended character sets use more than one byte per character. If such characters are stored in a file, the order of the bytes becomes important. In this situation, the writer of the character must use the same order that will be used by potential readers.

One way to do this is to use a **Byte Order Mark** (**BOM**). This is a known number of bytes with a known pattern, and is typically placed as the first item in a stream so that the reader of the stream can use it to determine the byte order of the remaining characters in the stream. Unicode defines the 16-bit character, `\uFEFF`, and the non-character, `\uFFFE`, as byte order marks. In the case of `\uFEFF`, all bits are set except for bit 8 (if the lowest bit is labeled as bit 0). This BOM can be prefixed to data that is passed between machines. The destination machine can read the BOM into a 16-bit variable and test the bits. If bit 8 is zero it means the two machines have the same byte order and so the characters can be read as two byte values in the order in the stream. If bit 0 is zero then it means that the destination machine reads 16-bit variables in the opposite order to the source, and so action must be taken to ensure that, with 16-bit characters, the bytes are read in the right order.

The Unicode byte order mark (BOM) is serialized as follows (in hexadecimal):

Character set Byte order mark

UTF-8	EF BB BF
UTF-16 big endian	FE FF
UTF-16 little endian	FF FE
UTF-32 big endian	00 00 FE FF
UTF-32 little endian	FF FE 00 00

Bear this in mind that, when you read data from a file. The character sequence, FE FF, will be very rare in a non-Unicode file, and so if you read these as the first two bytes in a file it means that the file is Unicode. Since \uFEFF and \uFFFE are not printable Unicode characters, it means that a file that starts with either of these has a byte order mark, and you can then use the BOM to determine how to interpret the remaining bytes in the file.

Boolean

The bool type holds a Boolean value, that is, just one of two values: true or false. C++ allows you to treat 0 (zero) as being false and anything non-zero as being true but this can lead to mistakes, so it is better to get in to the habit of explicitly checking values:

```
int use_pointer(int *p)
{
if (p) { /* not a null pointer */ }
if (p != nullptr) { /* not a null pointer */ }
return 0;
}
```

The second of these two is preferable, since it is clearer what you are comparing.
Note that, even if a pointer is not a nullptr, it still may not be a valid pointer, but it is common practice to assign a pointer to nullptr to convey some other meaning, perhaps to say that the pointer operation is not appropriate.
You can insert Boolean values into an output stream. However, the default behavior is to treat a Boolean as an integer. If you want cout to output bool values with string names, then insert the manipulator boolalpha in to the stream; this will make the stream print true or false to the console. The default behavior can be achieved by using the noboolalpha manipulator.

void

In some cases, you need to indicate that a function does not have parameters or will not return a value; in both cases, you can use the keyword void:

```
void print_message(void)
{
cout << "no inputs, no return value" << endl;
}
```

The use of void in the parameter list is optional; an empty pair of parentheses is acceptable and is preferable. This is the only way to indicate that a function returns no value other than returning void.
Note that void is not really a type because you cannot create a void variable; it is the absence of a type. As you'll find out in the next chapter, you can create pointers of the type void, but you will not be able to use the memory that such pointers point to without casting to a typed pointer: to use the memory you have to decide the type of the data that the memory holds.

Initializers

Initializers were touched upon in the last chapter, but we will go into more depth here. For built-in types, you must initialize a variable before you use it. For custom types, it is possible for the type to define a default value, but there are some issues in doing this, which will be covered in Chapter 6, *Classes*.

In all versions of C++, there are three ways to initialize a built-in type: assignment, function syntax, or calling a constructor. In C++11 another way to initialize variables was introduced: construction through a list initializer. These four ways are shown here:

```
int i = 1;
int j = int(2);
int k(3);
int m{4};
```

The first of these three is the clearest; it shows, using an easy to understand syntax, that the variable is being initialized to a value. The second example initializes a variable by calling the type as if it is a function. The third example calls the constructor of the int type. This is a typical way to initialize custom types, so it is best to reserve this syntax just for custom types.
The fourth syntax is new to C++11 and initializes the variable using an initialize list between curly braces ({}). Just to confuse things slightly, you can also initialize a built-in type using the same syntax as an assignment to a single item list:

```
int n = { 5 };
```

This is really confusing things, the type n is an integer, not an array. Recall that, in the last chapter, we created an array with the birth dates of The Beatles:

```
int birth_years[] = { 1940, 1942, 1943, 1940 };
```

This creates an array of four integers; the type of each item is int but the type of the array variable is int*. The variable points to memory that holds four integers. Similarly, you can also initialize a variable to be an array of one item:

```
int john[] = { 1940 };
```

This is exactly the same initialization code that C++11 allows to initialize a single integer. In addition, the same syntax is used to initialize instances of record types (structs), adding another level of potential confusion about what the syntax means.

It is best to avoid using the curly brace syntax for variable initialization and use it exclusively for initializing lists. However, there are advantages to this syntax for casting, as explained shortly.
The curly brace syntax can be used to provide the initial values for any of the collection classes in the C++ Standard Library, as well as for C++ arrays. Even when used to initialize a collection object, there is a potential for confusion. For example, consider the vector collection class. This can hold a collection of the type provided through a pair of angled brackets (<>). The capacity of an object of this class can grow as you add more items to the object, but you can optimize its use by specifying an initial capacity:

```
vector<int> a1 (42);
cout << " size " << a1.size() << endl;
for (int i : a1) cout << i << endl;
```

The first line of this code says: create a vector object that can hold integers and start by reserving space for 42 integers, each initialized to a value of zero. The second line prints the size of the vector to the console (42) and the third line prints all the items in the array to the console and it will print 42 zeros.
Now consider the following:

```
vector<int> a2 {42};
cout << " size " << a2.size() << endl;
for (int i : a2) cout << i << endl;
```

There is only one change here: the parentheses have been changed to braces but it means that the initialization has been changed entirely. The first line now means: create a vector that can hold integers and initialize it with the single integer, 42. The size of a2 is 1 and the last line will print just one value, 42.
The great power of C++ is that it should be easy to write correct code, and to persuade the compiler to help you to avoid mistakes. The use of braces for single item initialization increases the possibility of hard-to-find errors.

Default values

Variables of built-in types should be initialized before you first use them, but there are some situations when the compiler will provide a default value.

If you declare a variable at file scope, or globally in your project, and you do not give it an initial value, the compiler will give it a default value. For example:

```
int outside;
int main()
{
outside++;
cout << outside << endl;
}
```

This code will compile and run, printing a value of 1; the compiler has initialized outside to 0, which is then incremented to 1. The following code will not compile:

```
int main()
{
int inside;
inside++;
cout << inside << endl;
}
```

The compiler will complain that the increment operator is being used on an uninitialized variable.

In the last chapter, we saw another example of the compiler providing a default value: static.

```
int counter()
{
static int count;
return ++count;
}
```

This is a simple function that maintains a count. The variable count is marked with the static storage class modifier, meaning that the variable has the same lifetime as the application (allocated when the code starts and deallocated when the program ends); however, it has internal linkage, meaning the variable can only be used within the scope of where it is declared, the counter function. The compiler will initialize the count variable with a default value of zero, so that the first time the counter function is called it will return a value of 1.

The new initialize list syntax of C++11 provides a way for you to declare a variable and specify that you want it initialized by the compiler to the default value for that type:

```
int a {};
```

Of course, when reading this code, you have to know what the default value for an int is (it is zero). Again, it is much easier and more explicit to simply initialize the variable to a value:

```
int a = 0;
```

The rules for default values are simple: a value of zero. Integers and floating point numbers have a default value of 0, for a character the default value is \0, for a bool it is false, and for a pointer the default is the constant, nullptr.

Declarations without a type

C++11 introduces a mechanism for declaring that a variable's type should be determined from the data it is initialized with, that is, auto.

There is a minor confusion here because prior to C++11, the auto key was used to declare **automatic** variables, that is, variables that are automatically allocated on the stack in a function. Other than variables declared at file scope or as static, all the other variables in this book so far have been automatic variables and automatic variables are the most widely used **storage class** (explained shortly). Since it was optional and applicable to most variables, the auto keyword was rarely used in C++, so C++11 took advantage of this, removed the old meaning, and gave auto a new meaning.

If you are compiling old C++ code with a C++11 compiler and that old code uses auto, you will get errors because the new compiler will assume auto will be used with variables with no specified type. If this happens, simply search and delete each instance of auto; it was redundant in C++ prior to C++11, and there was little reason for a developer to use it.

The auto keyword means that the compiler should create a variable with the type of the data that is assigned to it. The variable can only have a single type, the type the compiler decides is the type it needs for the data assigned to it, and you cannot use the variable elsewhere to hold data of a different type. Because the compiler needs to determine the type from an initializer, it means that all auto variables must be initialized:

```
auto i = 42; // int
auto l = 42l; // long
auto ll = 42ll; // long long
auto f = 1.0f; // float
auto d = 1.0; // double
auto c = 'q'; // char
auto b = true; // bool
```

Note that there is no syntax to specify that an integer value is a single byte or two bytes, so you cannot create unsigned char variables or short variables this way.

This is a trivial use of the auto keyword and you should not use it this way. The power of auto is when you use containers that can result in some fairly complicated looking types:

```
// #include <string>
// #include <vector>
// #include <tuple>
vector<tuple<string, int> > beatles;
beatles.push_back(make_tuple("John", 1940));
beatles.push_back(make_tuple("Paul", 1942));
beatles.push_back(make_tuple("George", 1943));
beatles.push_back(make_tuple("Ringo", 1940));
for (tuple<string, int> musician : beatles)
{
cout << get<0>(musician) << " " << get<1>(musician) << endl;
}
```

This code uses the vector container we have used before, but it stores two value items using a tuple. The tuple class is simple; you declare a list of the types of items in the tuple object in the declaration between the angle brackets. So, the tuple<string, int> declaration says that the object will hold a string and an integer, in that order. The make_tuple function is provided by the C++ Standard Library and will create a tuple object containing the two values. The push_back function will put the item into the vector container. After the four calls to the push_back function, the beatles variable will contain four items and each one is a tuple with a name and birth year.

The range for loops through the container and on each loop assigns the musician variable with the next item in the container. The values in the tuple are printed to the console in the statements in the for loop. An item in the tuple is accessed using the get parameterized function (from <tuple>) where the parameter in the angle brackets indicates the index of the item (indexed from zero) to get from the tuple object passed as a parameter in the parentheses. In this example, the call to get<0> gets the name which is printed out, then a space, and then get<1> gets the year item in the tuple. The result of this code is:

```
John 1940
Paul 1942
```

```
George 1943
Ringo 1940
```

This text has poor formatting because it does not take into account the length of the names. This can be addressed using manipulators explained in Chapter 9, *Using String*.

Take another look at the for loop:

```
for (tuple<string, int> musician : beatles)
{
cout << get<0>(musician) << " " << get<1>(musician) << endl;
}
```

The type of musician is tuple<string, int>;, this is a fairly simple type, and as you use the standard template more you could end up with some complicated types (particularly when you use **iterators**). This is where auto becomes useful. The following code is the same, but easier to read:

```
for (auto musician : beatles)
{
cout << get<0>(musician) << " " << get<1>(musician) << endl;
}
```

The musician variable is still typed, it is a tuple<string, int>, but auto means you do not have to explicitly code this.

Storage classes

When declaring a variable, you can specify its storage class which indicates the lifetime, linkage (what other code can access it), and memory location of the variable.

You have already seen one storage class, static, which when applied to a variable in a function means that the variable can only be accessed within that function, but its lifetime is the same as the program. However, static can be used on variables declared at file scope, in which case it indicates that the variable can only be used in the current file, which is called **internal linkage**. If you omit the static keyword on a variable, defined at file scope, then it has an **external linkage,** which means the name of the variable is visible to code in other files. The static keyword can be used on a data member of a class, and on methods defined on a class, both of which have interesting effects that will be described in Chapter 6, *Classes*.

The static keyword says that the variable can only be used in the current file. The extern keyword indicates the opposite; the variable (or function) has external linkage and can be accessed in other files in the project. In most cases, you will define a variable in one source file, and then declare it as extern in a header file so that the same variable can be used in other source files.

The final storage class specifier is thread_local. This is new to C++11 and it only applies to multithreaded code. This book does not cover threading, so only a brief description will be given here.

A thread is a unit of execution and concurrency. You can have more than one thread running in a program, and it is possible to have two or more threads running the same code at the same time. This means that two different threads of execution could access and alter the same variable. Since concurrent access may have undesirable effects, multithreaded code often involves taking action to ensure that only one thread can access data at any time. If such code is not written carefully there is a danger of deadlocks, where the execution of threads is paused (in the worst cases, indefinitely) for exclusive access to the variable, negating the benefit of using threads.

The thread_local storage class indicates that each thread will have its own copy of a variable. So, if two threads access the same function and a variable in that function is marked as thread_local, it means that each thread only sees the changes it makes.

You will sometimes see the storage class register used in older C++ code. This is now deprecated. It was used as a hint to the compiler that the variable has important consequences on the performance of the program and suggests to the compiler that if possible it should use a CPU register to hold the variable. The compiler could ignore this suggestion. In fact, in C++11 the compiler literally does ignore the keyword; code with register variables will compile with no errors or warnings and the compiler will optimize the code however it feels is necessary.

Although it is not a storage class specifier, the volatile keyword has an effect on compiler code optimization. The volatile keyword indicates that a variable (perhaps through **Direct Memory Access** (**DMA**) to some hardware) can be altered by an external action, and so it is important that the compiler *does not* apply any optimizations.

There is one other storage class modifier called mutable. This can only be used on class members and so it will be covered in Chapter 6, *Classes*.

Using type aliases

Sometimes the names of types can become quite cumbersome. If you use nested namespaces, the name of a type includes all of the namespaces used. If you define parameterized types (examples used so far in this chapter are vector and tuple), the parameters increase the name of the type. For example, earlier we saw a container for the names and birth years of musicians:

```
// #include <string>
// #include <vector>
// #include <tuple>
vector<tuple<string, int> > beatles;
```

Here, the container is a vector and it holds items that are tuple items, each of which will hold a string and an integer. To make the type easier to use you could define a preprocessor symbol:

```
#define name_year tuple<string, int>
```

Now you can use name_year instead of the tuple in your code, and the preprocessor will replace the symbol with the type before the code is compiled:

```
vector<name_year> beatles;
```

However, because #define is a simple search and replace, there can be problems as explained earlier in this book. C++ provides the typedef statement to create an alias for a type:

```
typedef tuple<string, int> name_year_t;
vector<name_year_t> beatles;
```

Here, an alias called name_year_t is created for tuple<string, int>.

With a typedef, the alias usually comes at the end of the line preceded by the type it aliases. This is the opposite order to #define, where the symbol you are defining comes after #define, followed by its definition. Note also typedef is terminated with a semicolon. It becomes much more complicated with function pointers, as you'll see in Chapter 5, *Using Functions*.

Now, wherever you want use the tuple, you can use the alias:

```
for (name_year_t musician : beatles)
{
cout << get<0>(musician) << " " << get<1>(musician) << endl;
}
```

You can typedef aliases:

```
typedef tuple<string, int> name_year_t;
typedef vector<name_year_t> musician_collection_t;
musician_collection_t beatles2;
```

The type of the beatles2 variable is vector<tuple<string, int>>. It is important to

note that typedef creates an alias; it does not create a new type, so you can switch between the original type and its alias.
The typedef keyword is a well-established way to create aliases in C++.
C++11 introduces another way to create a type alias, the using statement:

```
using name_year = tuple<string, int>;
```

Again, this does not create a new type, it creates a new name for the same type, and semantically, this is the same as typedef. The using syntax can be more readable than using a typedef and it also allows you to use templates.
The using method of creating an alias is more readable than typedef because the use of the assignment follows the convention used for variables, that is, the new name on the left is used for the type on the right of the =.

Aggregating data in record types

Often you will have data that is related and must be used together: an aggregated type. Such a record type allows you to encapsulate data into a single variable. C++ inherits from C struct and union, as ways of providing records.

Structures

In most applications, you will want to associate several data items together. For example, you may want to define a time record that has an integer for each of the following: the hour, the minute, and the second of the specified time. You can declare them like this:

```
// start work
int start_sec = 0;
int start_min = 30;
int start_hour = 8;
// end work
int end_sec = 0
int end_min = 0;
int end_hour = 17;
```

This approach becomes quite cumbersome and error-prone. There is no encapsulation, that is, the _min variables can be used in isolation to the other variables. Does the *minutes past the hour* make sense when it is used without the hour that it refers to? You can define a structure that associates these items:

```
struct time_of_day
{
int sec;
int min;
int hour;
};
```

Now you have the three values as part of one record, which means that you can declare variables of this type; although you can access individual items it is clear that the data is associated with the other members:

```
time_of_day start_work;
start_work.sec = 0;
start_work.min = 30;
start_work.hour = 8;
time_of_day end_work;
end_work.sec = 0;
end_work.min = 0;
end_work.hour = 17;
print_time(start_work);
print_time(end_work);
```

Now we have two variables: one that represents the start time, and the other that represents the end time. The members of a struct are encapsulated within the struct, that is, you access the member through the instance of the struct. To do this, you use the dot operator. In this code, start_work.sec means that you are accessing the sec member of the instance of the time_of_day structure called start_work. The members of a structure are public by default, that is, code outside the struct has access to the members.

Classes and structures can indicate the level of member access, and Chapter 6, *Classes*, will show how to do this. For example, it is possible to mark some members of a struct as private, which means that only code that is a member of the type can access the member.

A helper function called print_time is called to print the data to the console:

```
void print_time(time_of_day time)
{
cout << setw(2) << setfill('0') << time.hour << ":";
cout << setw(2) << setfill('0') << time.min << ":";
cout << setw(2) << setfill('0') << time.sec << endl;
}
```

In this case, the setw and setfill manipulators are used to set the width of the next inserted item to two characters and to fill any unfilled places with zeros (more details will be given in Chapter 9, *Using String*; in effect, setw gives the size of the column occupied by the next inserted data, and setfill specifies the padding character used).

Chapter 5, *Using Functions*, will go into more detail about the mechanism of passing structures to functions and the most efficient way to do it, but for the purpose of this section we will use the simplest syntax here. The important point is that the caller has associated the three items of data together using a struct and all the items can be passed to a function as a unit.

Initializing

There are several ways to initialize an instance of a structure. The preceding code shows one method: accessing the member using the dot operator, and assigning it a value. You can also assign values to an instance of a struct through a specially provided function called a constructor. Since there are special rules about how to name a constructor and what you can do in them, this will be left until Chapter 6, *Classes*.

You can also initialize structures using the list initializer syntax using curly braces ({}). The items in the braces should match the members of the struct in the order of the members as declared. If you provide fewer values than there are members, the remaining members are initialized to zero. Indeed, if you provide no items between the curly braces then all members are set to zero. It is an error to provide more initializers than there are members. So, use the time_of_day record type defined previously:

```
time_of_day lunch {0, 0, 13};
time_of_day midnight {};
time_of_day midnight_30 {0, 30};
```

In the first example, the lunch variable is initialized to 1 PM. Notice that, because the hour member is declared as the third member in the type, it is initialized using the third item in the initialize list. In the second example, all members are set to zero, and of course, zero hours is midnight. The third example provides two values, so these are used to initialize sec and min.

You can have a member of a struct that is a struct itself, and this is initialized using nested braces:

```
struct working_hours
{
time_of_day start_work;
time_of_day end_work;
};
working_hours weekday{ {0, 30, 8}, {0, 0, 17} };
cout << "weekday:" << endl;
```

```
print_time(weekday.start_work);
print_time(weekday.end_work);
```

Structure fields

A structure can have members that are as small as a single bit, called a **bit-field**. In this case, you declare an integer member with the number of bits that the member will take up. You are able to declare unnamed members. For example, you may have a structure that holds information about the length of an item, and whether the item has been changed (is dirty). The item this refers to has a maximum size of 1,023, so you need an integer with at least 10 bits of width to hold this. You could use an unsigned short to hold both the length and the dirty information:

```
void print_item_data(unsigned short item)
{
unsigned short size = (item & 0x3ff);
char *dirty = (item > 0x7fff) ? "yes" : "no";
```

```
cout << "length " << size << ", ";
cout << "is dirty: " << dirty << endl;
}
```

This code separates the two pieces of information, and then prints them out. A bitmap like this is quite unfriendly to code. You can use a struct to hold this information using an unsigned short to hold the 10 bits of length information and a bool to hold the dirty information. Using bit fields you can define the structure like this:

```
struct item_length
{
unsigned short len : 10;
unsigned short : 5;
bool dirty : 1;
};
```

The len member is marked as unsigned short but only 10 bits are needed, so this is mentioned using the colon syntax. Similarly, a Boolean yes/no value can be held in just one bit. The structure indicates that there are five bits between the two values that are not used, and so have no name.

Fields are simply a convenience. Although it looks like the item_length structure should only take up 16 bits (unsigned short), there is no guarantee that the compiler will do this. If you receive an unsigned short from a file or network stream you will have to extract the bits yourself:

```
unsigned short us = get_length();
item_length slen;
slen.len = us & 0x3ff;
slen.dirty = us > 0x7fff;
```

Using structure names

In some cases, you may need to use a type before you have actually defined it. As long as you do not use the members, you can declare a type before defining it:

```
struct time_of_day;
void print_day(time_of_day time);
```

This could be declared in a header, where it says that there is a function defined somewhere else that takes a time_of_day record and prints it out. To be able to declare the print_day function, you have to have declared the time_of_day name. The time_of_day struct must be defined somewhere else in your code before the function is defined, otherwise you will get an *undefined type* error.

There is, however, an exception: a type can hold pointers to instances of the same type before the type is fully declared. This is because the compiler knows the size of a pointer, so it can allocate sufficient memory for the member. It is not until the entire type has been defined before you can create an instance of the type. The classic example of this is a linked list, but since this requires using pointers and dynamic allocation, that will be left to the next chapter.

Determining alignment

One of the uses of structs is that if you know how data is held in memory you can deal with a struct as a block of memory. This is useful if you have a hardware device that is mapped into memory, where different memory locations refer to values controlling or returning values from the device. One way to access the device would be to define a struct that matches the memory layout of the device's direct memory access to C++ types. Further, structs are also useful for files, or for packets of data that need to be transmitted over the network: you manipulate the struct and then copy the memory occupied by the struct to the file or to the network stream.

The members of the struct are arranged in memory in the order that they are declared in the type. The items will take up *at least* as much memory as each type requires. A member may take more memory than the type requires, and the reason for this is a mechanism called **alignment**.

The compiler will place variables in memory in the way that is the most efficient, in terms of memory usage, or speed of access. The various types will be aligned to alignment boundaries. For example, a 32-bit integer will be aligned to a four-byte boundary, and if the next available memory location is not on this boundary the compiler will skip a few bytes and put the integer at the next alignment boundary. You can test the alignment of a specific type using the alignof operator passing the type name:

```
cout << "alignment boundary for int is " 0
<< alignof(int) << endl; // 4
cout << "alignment boundary for double is "
<< alignof(double) << endl; // 8
```

The alignment of an int is 4 and this means that an int variable will be placed at the next four-byte boundary in memory. The alignment of a double is 8 and this makes sense because in Visual C++ a double occupies eight bytes. So far, it looks like the result of alignof is the same as sizeof; however, this is not so.

```
cout << "alignment boundary for time_of_day is "
<< alignof(time_of_day) << endl; // 4
```

This example prints the alignment of the time_of_day struct, which we previously defined to be three integers. The alignment of this struct is 4, that is, the alignment of the largest item in the struct. This means that an instance of time_of_day will be placed on a 4-byte boundary; it does not say how the items within the time_of_day variable will be aligned.

As an example, consider the following struct, which has four members that occupy respectively one, two, four, and eight bytes:

```
struct test
{
uint8_t uc;
uint16_t us;
uint32_t ui;
uint64_t ull;
}
```

The compiler will tell you that the alignment is 8 (the alignment of the largest item, ull), but that the size is 16, which may appear a little odd. If every item were aligned on 8-byte boundaries, then the size would have to be 32 (four times eight). If the items were stored in memory and packed as efficiently possible, then the size would be 15. Instead, what is happening is that the second item is aligned on a two-byte boundary, which means that there is one byte of unused space between uc and us.

If you want to align the internal items onto, say, the same boundaries as used by a uint32_t variable, you can mark an item with alignas and give the alignment that you need. Note that, because 8 is bigger than 4, any item aligned on an 8-byte boundary will also be aligned on a 4-byte boundary:

```
struct test
{
uint8_t uc;
alignas(uint32_t) uint16_t us;
uint32_t ui;
uint64_t ull;
}
```

The uc item will be aligned on a 4-byte boundary already (alignof(test) will be 8), and it will occupy one byte. The us member is a uint16_t but it is marked with alignas(uint32_t) to say that it should be aligned in the same way as a uint32_t, that is, on a 4-byte boundary. This means that both uc and us will be on 4-byte boundaries with padding provided. Of course, the ui member will also be aligned on a 4-byte boundary because it is a uint32_t.

If the struct has just these first three members, then the size would be 12. However, the struct has another member, the 8-byte ull member. This must be aligned on an 8-byte boundary, which means 16 bytes from the beginning of the struct, and to do this there needs to be 4 bytes of padding between ui and ull. As a consequence, the size of test is now reported as 24: 4 bytes for uc and for us (because the following item ui has to be aligned on the next four-byte boundary), 8 bytes for ull (because it is an 8-byte integer), and 8 bytes for ui because the following item (ull) has to be on the next 8-byte boundary.

The following diagram shows the location in memory of the various members of the test type:

You cannot use alignas to relax alignment requirements, so you cannot mark a uint64_t variable to be aligned on a two-byte boundary that is not also an eight-byte boundary.

In most cases, you will not need to worry about alignment; however, if you are accessing memory-mapped devices, or binary data from files, it is convenient if you can directly map this data to a struct and in this case, you will find that you will have to pay close attention to alignment. This is known as **plain old data** and you will often see structs referred to as **POD types**.

POD is an informal description, and sometimes it is used to describe types that have a simple construction and do not have virtual members (see Chapter 6, *Classes* and Chapter 7, *Introduction to Object-Oriented Programming*). The standard library provides a function in <type_traits> called is_pod that tests a type for these members.

Storing data in the same memory with unions

A union is a struct where all the members occupy the same memory. The size of such a type is the size of the largest member. Since a union can only hold one item of data, it is a mechanism to interpret the data in more than one way.

An example of a union is in the VARIANT type that is used to pass data between **Object Linking and Embedding** (**OLE**) objects in Microsoft's **Component Object Model** (**COM**). The VARIANT type can hold data of any of the data types that COM is able to transmit between OLE objects. Sometimes OLE objects will be in the same process, but it is possible for them to be in different processes on the same machine or on different machines. COM guarantees that it can transmit VARIANT without the developer providing any additional networking code. The structure is complicated, but an edited version is shown here:

```
// edited version
struct VARIANT
{
```

```
unsigned short vt;
union
{
unsigned char bVal;
short iVal;
long lVal;
long long llVal;
float fltVal;
double dblVal;
};
};
```

Notice that you can use a union without a name: this is an anonymous union and from a member access point of view you access a member of the union as if it is a member of the VARIANT that contains it. The union contains a member for every type that can be transmitted between OLE objects, and the vt member indicates which one is used. When you create a VARIANT instance, you have to set vt to the appropriate value and then initialize the related member:

```
enum VARENUM
{
VT_EMPTY = 0,
VT_NULL = 1,
VT_UI1 = 17,
VT_I2 = 2,
VT_I4 = 3,
VT_I8 = 20,
VT_R4 = 4,
VT_R8 = 5
};
```

This record ensures that only the memory that is needed is used, and the code that transmits the data from one process to another will be able to read the vt member to determine how the data needs to be processed so that it can be transmitted:

```
// pseudo code, real VARIANT should not be handled like this
VARIANT var {}; // clear all items
var.vt = VT_I4; // specify the type
var.lVal = 42; // set the appropriate member
pass_to_object(var);
```

Note that you must be disciplined and only initialize the appropriate member. When your code receives a VARIANT, you must read vt to see which member you should use to access the data.

In general, when using unions you should access only the item that you initialize:

```
union d_or_i {double d; long long i};
d_or_i test;
test.i = 42;
cout << test.i << endl; // correct use
cout << test.d << endl; // nonsense printed
```

Accessing runtime type information

C++ provides an operator called typeid that will return type information about a variable (or a type) at runtime. **Runtime Type Information** (**RTTI**) is significant when you use custom types that can be used in a **polymorphic** way; details will be left until later chapters. RTTI allows you to check at runtime the type of a variable and process the variable accordingly. RTTI is returned through a type_info object (in the <typeinfo> header file):

```
cout << "int type name: " << typeid(int).name() << endl;
int i = 42;
```

```
cout << "i type name: " << typeid(i).name() << endl;
```

In both cases, you'll see **int** printed as the type. The type_info class defines comparison operators (== and !=) so you can compare types:

```
auto a = i;
if (typeid(a) == typeid(int))
{
cout << "we can treat a as an int" << endl;
}
```

Determining type limits

The <limits> header contains a template class called numeric_limits and this is used through the specializations provided for each of the built-in types. The way to use these classes is to provide the type you want information for in the angle brackets and then call the static members on the class using the scope resolution operator (::). (Full details of static functions on classes will be given in Chapter 6, *Classes*). The following prints the limits of the int type to the console:

```
cout << "The int type can have values between ";
cout << numeric_limits<int>::min() << " and ";
cout << numeric_limits<int>::max() << endl;
```

Converting between types

Even if you try exceptionally hard to use the correct types in your code, at some point you will find that you will have to convert between types. For example, you may be using library functions that return a value of a particular type, or you may be reading in data from an external source that is a different type to your routine.

With built-in types, there are standard rules about conversion between different types, some of which will be automatic. For example, if you have an expression like a + b, and a and b are different types, then, if it is possible, the compiler will automatically convert one variable's value to the type of the other and the + operator for that type will be called.
In other cases, you may need to force one type to another type so that the right operator is called and this will require a cast of some kind. C++ allows you to use C-like casts, but these do not have runtime tests, so it is far better to use C++ casts, which have various levels of runtime checks and type safety.

Type conversions

Built-in conversions can have one of two outcomes: promotion or narrowing. A promotion is when a smaller type is promoted to a larger type and you will not lose data. A narrowing conversion happens when a value from a larger type is converted to a smaller type with potential loss of data.

Promoting conversions

In a mixed type expression, the compiler will attempt to promote smaller types to the larger type. So, a char or a short can be used in an expression where an int is needed because it can be promoted to the larger type with no loss of data.
Consider a function declared as taking a parameter that is an int:

```
void f(int i);
```

We can write:

```
short s = 42;
f(s); // s is promoted to int
```

Here the variable s is silently converted to an int. There are some cases that may appear odd:

```
bool b = true;
f(b); // b is promoted to int
```

Again, the conversion is silent. The compiler assumes you know what you are doing and that your intention is that you want false treated as 0 and true treated as 1.

Narrowing conversions

In some cases, *narrowing* occurs. Be very careful of this because it loses data. In the following, an attempt is made to convert a double into an int.

```
int i = 0.0;
```

This is allowed, but the compiler will issue a warning:

```
C4244: 'initializing': conversion from 'double' to 'int', possible loss of
data
```

This code is clearly wrong, but the mistake is not an error because it may be intentional. For example, in the following code we have a function that has a parameter that is a floating point, and within the routine the parameter is used to initialize an int:

```
void calculation(double d)
{
// code
int i = d;
// use i
// other code
}
```

This may be intentional, but because there will be a loss of precision you should document why you are doing this. At the very least, use a cast operator so that it is obvious that you understand the consequence of the action.

Narrowing to bool

As mentioned previously, pointers, integers, and floating point values can be implicitly converted to bool where a nonzero value converts to true and a zero value converts to false. This can result in a nasty bug that is difficult to notice:

```
int x = 0;
if (x = 1) cout << "not zero" << endl;
else cout << "is zero" << endl;
```

Here, the compiler sees the assignment expression x = 1, which is a bug; it should be the comparison x == 1. However, this is valid C++ because the value of the expression is 1 and the compiler, helpfully, converts this to a bool value of true. This code will compile without even a warning, and not only will it produce a result that is the opposite of what you expect (you'll see not zero printed on the console), but also the assignment will have changed the value of the variable propagating the error throughout the program.

It is easy to avoid this bug by getting into the habit of always constructing a comparison so that the rvalue of a potential assignment is on the left. In a comparison, there will be no concept of rvalue or lvalue, so this uses the compiler to catch an assignment when it is not intended:

```
if (1 = x) // error
cout << "not zero" << endl;
```

Converting signed types

Signed to unsigned conversions can happen and can cause unexpected results. For example:

```
int s = -3;
unsigned int u = s;
```

The unsigned short variable will be assigned with a value of 0xfffffffd, that is, the two's compliment of 3. This may be the result you want, but it is an odd way of getting it. Interestingly, if you try and compare these two variables, the compiler will issue a warning:

```
if (u < s) // C4018
cout << "u is smaller than s" << endl;
```

The Visual C++ warning C4018 given here is '<': signed/unsigned mismatch, which says that you cannot compare a signed and unsigned type, and to do so would need a cast.

Casting

In some cases, you will have to convert between types. For example, this may be because the data is provided in a different type to the routines that you use to process it. You may have a library that processes floating point numbers as float, but your data is inputted as double. You are aware that the conversion will lose precision but know that this will have little effect on the final result so you do not want the compiler warning you. What you want to do is tell the compiler that the coercion of one type to another is acceptable.
The following table summarizes the various cast operations you can use in C++11:

Name	**Syntax**
Construction	{}
Remove const requirement	const_cast
Cast with no runtime checks	static_cast
Bitwise casting of types	reinterpret_cast
Cast between class pointers, with runtime checks	dynamic_cast
C style	()
Function style	()

Casting away const-ness

As mentioned in the last chapter, the const specifier is used to indicate to the compiler that an item will not change, and that any attempt by your code to change the item is an error. There is another way to use this specifier, which will be explored in the next chapter. When const is applied to a pointer, it indicates that the memory that the pointer points to cannot be changed:

```
char *ptr = "0123456";
// possibly lots of code
ptr[3] = '\0'; // RUNTIME ERROR!
```

This badly written code tells the compiler to create a string constant with the value 0123456 and then put the address of this memory into the string pointer ptr. The final line attempts to write to the string. This will compile but it will cause an access violation at runtime. Applying const to the pointer declaration will ensure that the compiler will check for such situations:

```
const char *ptr = "0123456";
```

More typical cases are applying const to pointers that are function parameters and the intention is the same: it indicates to the compiler that the data the pointer points to should be read-only. However, there may be situations when you want to remove the const property of such a pointer, and this is carried out using the const_cast operator:

```
char * pWriteable = const_cast<char *>(ptr);
pWriteable[3] = '\0';
```

The syntax is simple. The type that you want to convert to is given in the angle brackets (<>) and the variable (which is a const pointer) is provided in the parentheses.
You can also cast a pointer *to* a const pointer. This means that you can have one pointer that you can use to access the memory so that you can write to it, and then after you have made changes, you create a const pointer to the memory, in effect making the memory read-only through the pointer.

Clearly, once you have cast away the const-ness of a pointer you take responsibility for the damage that you do by writing to the memory, so the const_cast operator in your code is a good marker for you to examine code during a code review.

Casting without runtime checks

Most casts are performed using the static_cast operator, and this can be used to convert pointers to related pointer types as well as converting between numeric types. There are no runtime checks performed so you should be certain that the conversion is acceptable:

```
double pi = 3.1415;
int pi_whole = static_cast<int>(pi);
```

Here a double is converted to an int, which means that the fractional part is discarded. Normally the compiler would issue a warning that data is lost, but the static_cast operator shows that this is your intention and hence the warning is not given.

The operator is often used to convert void* pointers to a typed pointer. In the following code the unsafe_d function assumes that the parameter is a pointer to a double value in memory, and so it can convert the void* pointer to a double* pointer. The * operator used with the pd pointer *dereferences* the pointer to give the data that it points to. Thus, the *pd expression will return a double.

```
void unsafe_d(void* pData)
{
double* pd = static_cast<double*>(pData);
cout << *pd << endl;
}
```

This is unsafe because you rely on the caller to ensure that the pointer actually points to a double. It could be called like this:

```
void main()
{
double pi = 3.1415;
unsafe_d(&pi); // works as expected
int pi_whole = static_cast<int>(pi);
unsafe_d(&pi_whole); // oops!
}
```

The & operator returns the memory address of the operand as a typed pointer. In the first case, a double* pointer is obtained and passed to the unsafe_d function. The compiler will automatically convert this pointer to the void* parameter. The compiler does this automatically with no checks that the pointer will be used correctly in the function. This is illustrated by the second call to unsafe_d where the int* pointer is converted to the void* parameter and then in the unsafe_d function it is cast by static_cast to a double* even though the pointer points to an int. Consequently, the dereference will return unpredictable data and cout will print nonsense.

Casting pointers without runtime checks

The reinterpret_cast operator allows pointers to one type to be converted to pointers of another type, and it can convert from a pointer to an integer and an integer to a pointer:

```
double pi = 3.1415;
int i = reinterpret_cast<int>(&pi);
cout << hex << i << endl;
```

Unlike static_cast, this operator always involves a pointer: converting between pointers, converting from a pointer to an integral type, or converting from an integral type to a pointer. In this example, a pointer to a double variable is converted to an int and the value printed to the console. In effect, this prints out the memory address of the variable.

Casting with runtime checks

The dynamic_cast operator is used to convert pointers between related classes, and for this reason it will be explained in Chapter 6, *Classes*. This operator involves runtime checks so that the conversion is only performed if the operand can be converted to the specified type. If the conversion is not possible then the operator returns nullptr, giving you an opportunity to only use converted pointers that point to an actual object of that type.

Casting with list initializer

The C++ compiler will allow some implicit conversions; in some cases, they may be intentional and in some cases, they may not be. For example, the following code is similar to code shown before: a variable is initialized to a double value and then later in the code it is used to initialize an int. The compiler will perform the conversion, and will issue a warning:

```
double pi = 3.1415;
// possibly loss of code
int i = pi;
```

If you ignore warnings then you may not notice this loss of precision, which may cause a problem. One way to get around this issue is to use initialization using curly braces:

```
int i = {pi};
```

In this case, if pi can be converted to an int without loss (for example, if pi is a short) then the code will compile without even a warning. However, if pi is an incompatible type (in this case, a double) the compiler will issue an error:

```
C2397: conversion from 'double' to 'int' requires a narrowing conversion
```

Here's an interesting example. The char type is an integer, but the << operator for char from the osteam class interprets a char variable as being a character, not a number, as per the following:

```
char c = 35;
cout << c << endl;
```

This will print # on the console, not 35**,** because 35 is the ASCII code for "#". To get the variable to be treated as a number you can use one of the following:

```
cout << static_cast<short>(c) << endl;
cout << short{ c } << endl;
```

As you can see, the second version (construction) is just as readable, but is shorter than the first.

Using C casts

Finally, you can use C style casts, but these are provided only so that you can compile legacy code. You should use one of the C++ casts instead. For completeness, the C style casts are shown here:

```
double pi = 3.1415;
float f1 = (float)pi;
float f2 = float(pi);
```

There are two versions: the first cast operator has the parentheses around the type to cast to, and in the second one the cast looks like a function call. In both cases, it would be better to use static_cast so that there is compile-time checking.

Using C++ types

In this final part of the chapter, we will develop a command-line application that allows you to print out the contents of a file in a mixed alphanumeric and hex format.

The application must be run with the name of a file, but optionally you can specify how many lines to print. The application will print on the console the contents of the file, 16 bytes per line. On the left, it gives the hex representation and on the right, it gives the printable representation (or a dot if the character is not in the printable non-extended ASCII range).

Create a new folder under C:\Beginning_C++ called Chapter_03**.** Start Visual C++ and create a C++ source file, and save it to the folder you just created as hexdump.cpp. Add a simple main function that allows the application to accept parameters, and provides support for input and output using C++ streams:

```
#include <iostream>
using namespace std;
int main(int argc, char* argv[])
{
}
```

The application will have up to two parameters: the first is the name of the file and the second is the number of 16-byte blocks to print on the command-line. This means that you'll need to check that the parameters are valid. Start by adding a `usage` function to give the application parameters and, if called with a non-null parameter, print out an error message:

```
void usage(const char* msg)
{
cout << "filedump filename blocks" << endl;
cout << "filename (mandatory) is the name of the file to dump"
<< endl;
cout << "blocks (option) is the number of 16 byte blocks "
<< endl;
if (nullptr == msg) return;
cout << endl << "Error! ";
cout << msg << endl;
}
```

Add this function before the `main` function so that you can call it from there. The function can be called with a pointer to a C string or with `nullptr`. The parameter is `const`, indicating to the compiler that the string will not be changed in the function, so if there is any attempt to change the string, the compiler will generate an error.

Add the following lines to the `main` function:

```
int main(int argc, char* argv[])
{
if (argc < 2) {
usage("not enough parameters");
return 1;
}
if (argc > 3) {
usage("too many parameters");
return 1;
```

```
} // the second parameter is file name
string filename = argv[1];
}
```

Compile the file and confirm that there are no typos. Since this application uses the C++ Standard Library, you have to provide support for C++ exceptions with the `/EHsc` switch:

```
cl /EHsc hexdump.cpp
```

You can test the application invoking it from the command-line with zero, one, two, and then three parameters. Confirm that the application will only allow it to be called with one or two parameters on the command-line (which actually means two or three parameters since `argc` and `argv` include the application name).

The next task is to determine if the user provided a number to indicate how many 16-byte blocks to dump to the console and, if so, convert the string provided by the command line into an integer. This code will perform the conversion from a string to a number using the `istringstream` class, so you need to include the header file where this class is defined.

Add the following to the top of the file:

```
#include <iostream>
#include <sstream>
```

After the declaration of the `filename` variable add the following highlighted code:

```
string filename = argv[1];
int blocks = 1; // default value
if (3 == argc) {
// we have been passed the number of blocks
istringstream ss(argv[2]);
ss >> blocks;
if (ss.fail() || 0 >= blocks) {
// cannot convert to a number
usage("second parameter: must be a number," "and greater than
zero");
return 1;
}
}
```

By default, the application will dump one line of data (up to 16 bytes) from the file. If the user provided a different number of lines, the string-formatted number is converted to an integer by using an istringstream object. This is initialized with the argument and then the number is extracted from the stream object. If the user typed a value of zero, or if they typed a value that cannot be interpreted as a string, the code prints an error message. The error string is split over two lines, but it is still one string.

Notice that, the if statement uses short-circuiting; that is, if the first part of the expression (ss.fail(), meaning the conversion failed), is true, then the second expression (0 >= blocks, that is blocks must be greater than zero) will not be evaluated.

Compile this code and try it several times. For example:

```
hexdump readme.txt
hexdump readme.txt 10
hexdump readme.txt 0
hexdump readme.txt -1
```

The first two should run without an error; the second two should generate an error. Don't worry that readme.txt does not exist, as it is only here as a test parameter.

Next, you'll add the code to open a file and process it. Since you'll use the ifstream class to input data from a file, add the following header file to the top of the file:

```
#include <iostream>
#include <sstream>
#include <fstream>
```

Then at the bottom of the main function add the code to open the file:

```
ifstream file(filename, ios::binary);
if (!file.good())
{
usage("first parameter: file does not exist");
return;
}
while (blocks-- && read16(file) != -1);
file.close();
```

The first line creates a stream object called file and this is attached to the file specified through the path given in filename. If the file cannot be found, the good function will return false. This code negates the value using the ! operator so that if the file *does not exist* the statements in the braces following the if are executed. If the file exists and the ifstream object can open it, the data is read 16 bytes at a time in a while loop. Notice that at the end of this code the close function is called on the file object. It is good practice to explicitly close resources when you have finished with them.

The file will be accessed by the read16 function on a byte-by-byte basis, including nonprintable bytes, so that control characters like \r or \n have no special meaning and are still read in. However, the stream class treats the \r character in a special way: this is treated as the end of a line and normally the stream will silently consume this character. To prevent this, we open the file in binary mode using ios::binary.

Review the while statement again:

```
while (blocks-- && read16(file) != -1);
```

There are two expressions here. The first expression decrements the blocks variable, which holds the number of 16-byte blocks that will be printed. The postfix decrement means that the value of the expression is the value of the variable *before* the decrement, so if the expression is called when blocks is zero the whole expression is short-circuited and the while looping ends. If the first expression is non-zero, then the read16 function is called and if this returns a value of -1 (the end of the file is reached), the looping finishes. The actual work of the loop occurs within the read16 function, so the while loop statement is the empty statement.

Now you must implement the read16 function just above the main function. This function will use a constant that defines the length of each block, so add the following declaration near the top of the file:

```
using namespace std;
const int block_length = 16;
```

Just before the main function, add this code:

```
int read16(ifstream& stm)
{
if (stm.eof()) return -1;
int flags = cout.flags();
cout << hex;
string line;
// print bytes
cout.setf(flags);
return line.length();
}
```

This is just skeleton code for the function. You'll add more code in a moment.

This function will read up to 16 bytes at a time and print the contents of those bytes to the console. The return value is the number of bytes that were read or -1 if the end of the file was reached. Notice the syntax used to pass the stream object to the function. This is a **reference**, a type of pointer to the actual object. The reason why a reference is used is because if we do not do this the function will get a *copy* of the stream. References will be covered in the next chapter and using object references as function parameters will be covered in Chapter 5, *Using Functions*.

The first line of this function test is to verify whether the end of the file has been reached, and if so, no more processing can be done and the value of -1 is returned. The code will manipulate the cout object (for example, inserting the hex manipulator); and so that you always know the state of this object outside of the function, the function ensures that when it returns the cout object is in the same state as when the function was called. The initial formatting state of the cout object is obtained by calling the flags function, and this is used to reset the cout object just before the function returns by calling the setf function.

This function does nothing, so it is safe to compile the file and confirm that you have no typos.

The read16 function does three things:

1. It reads in, byte-by-byte, up to 16 bytes.
2. It prints out the hex value of each byte.
3. It prints out the printable value of the byte.

This means that each line has two parts: the hex part on the left, and the printable part on the right. Replace the comment in the function with the highlighted code:

```
string line;
for (int i = 0; i < block_length; ++i) {
// read a single character from the stream
unsigned char c = stm.get();
if (stm.eof())
break;
// need to make sure that all hex are printed
// two character padded with zeros
cout << setw(2) << setfill('0');
cout << static_cast<short>(c) << " ";
if (isprint(c) == 0) line += '.';
else line += c;
}
```

The for loop will loop for a maximum of block_length times. The first statement reads a single character from the stream. This byte is read in as raw data. If get finds that there are no more characters in the stream, it will set a flag in the stream object, and this is tested by calling the eof function. If the eof function returns true it means that the end of the file has been reached and so the for loop finishes, but the function does not return immediately. The reason is that *some* bytes may have been read so more processing must be carried out.

The remainder of the statements in the loop do two things:

There are statements to print the hex value of the character on the console

There's a statement to store the character in a printable form in the line variable

We have already set the cout object to output hex values, but if the byte is less that 0x10 the value will not be printed prefixed with a zero. To get this format, we insert the setw manipulator to say that the data inserted will take up two character positions and setfill to say that a 0 character is used to pad the string. These two manipulators are available in the <iomanip> header, so add them to the top of the file:

```
#include <fstream>
#include <iomanip>
```

Normally, when you insert a char into a stream the character value is shown, so the char variable is cast to a short so that the stream will print the hex numeric value. Finally, a single space is printed between each item.

The final lines in the for loop are shown here:

```
if (isprint(c) == 0) line += '.';
else line += c;
```

This code checks to see if the byte is a printable character (" " to "~") using the isprint macro and if the character is printable it is appended to the end of the line variable. If the byte is not printable, a dot is appended to the end of the line variable as a placeholder.

The code so far will print the hex representation of the bytes to the console one after the other and the only formatting is the space between bytes. If you want to test the code, you can compile this and run it on the source file:

```
hexdump hexdump.cpp 5
```

You will see something unintelligible, such as the following:

```
C:\Beginning_C++\Chapter_03>hexdump hexdump.cpp 5
23 69 6e 63 6c 75 64 65 20 3c 69 6f 73 74 72 65 61 6d 3e 0d 0a
23 69 6e 63 6c 75 64 65 20 3c 73 73 74 72 65 61 6d 3e 0d 0a 23
69 6e 63 6c 75 64 65 20 3c 66 73 74 72 65 61 6d 3e 0d 0a 23 69
6e 63 6c 75 64 65 20 3c 69 6f 6d 61 6e 69 70 3e 0d
```

The 23 values are #, 20 are spaces, and 0d and 0a are returns and newlines.

We now need to print the character representations in the line variable and carry out some formatting, and add line breaks. After the for loop, add the following:

```
string padding = " ";
if (line.length() < block_length)
{
padding += string(
3 * (block_length - line.length()), ' ');
}
cout << padding;
cout << line << endl;
```

There will be at least *two* spaces between the hex display and the character display. One space will come from the last character printed out in the `for` loop, and the second space is provided in the initialization of the `padding` variable.
The maximum number of bytes on each line should be 16 bytes (`block_length`) and thus 16 hex values printed on the console. If a fewer number of bytes are read then extra padding is required so that over successive lines the character representations align. The actual number of bytes read, will be the length of the `line` variable obtained by calling the `length` function, so the number of missing bytes is the expression `block_length - line.length()`. Since every hex representation takes up three characters (two for the digits and one space), the padding needed is three times the number of missing bytes. To create the appropriate number of spaces, the string constructor is called with two parameters: the number of copies and the character to copy.
Finally, this padding string is printed to the console followed by the character representation of the bytes.

At this point you should be able to compile the code with no errors or warnings. When you run the code on the source file, you should see something like this:

```
C:\Beginning_C++\Chapter_03>hexdump hexdump.cpp 5
23 69 6e 63 6c 75 64 65 20 3c 69 6f 73 74 72 65 #include <iostre
61 6d 3e 0d 0a 23 69 6e 63 6c 75 64 65 20 3c 73 am>..#include <s
73 74 72 65 61 6d 3e 0d 0a 23 69 6e 63 6c 75 64 stream>..#includ
65 20 3c 66 73 74 72 65 61 6d 3e 0d 0a 23 69 6e e <fstream>..#in
63 6c 75 64 65 20 3c 69 6f 6d 61 6e 69 70 3e 0d clude <iomanip>.
```

Now the bytes make more sense. Since the application does not change the files it dumps, it is safe to use this tool on binary files, including itself:

C++ allows you to have direct access to memory through pointers. This gives you a lot of flexibility, and potentially it allows you to improve the performance of your code by eliminating some unnecessary copying of data. However, it also provides an extra source of errors; some can be fatal for your application or worse (yes, worse than fatal!) because poor use of memory buffers can open security holes in your code that can allow malware to take over the machine. Clearly pointers are an important aspect of C++.
In this chapter, you'll see how to declare pointers and initialize them to memory locations, how to allocate memory on the stack and, C++ free store, and how to use C++ arrays.

Using memory in C++

C++ uses the same syntax as C to declare pointer variables and assign them to memory addresses, and it has C-like pointer arithmetic. Like C, C++ also allows you to allocate memory on the stack, so there is automatic memory cleanup when the stack frame is destroyed, and dynamic allocation (on the C++ free store) where the programmer has the responsibility to release memory. This section will cover these concepts.

Using C++ pointer syntax

The syntax to access memory in C++ is straightforward. The `&` operator returns the address of an object. That *object* can be a variable, a built-in type or the instance of a custom type, or even a function (function pointers will be covered in the next chapter). The address is assigned a typed pointer variable or a `void*` pointer. A `void*` pointer should be treated as merely storage for the memory address because you cannot access data and you cannot perform pointer arithmetic (that is, manipulate the pointer value using arithmetic operators) on a void* pointer. Pointer variables are usually declared using a type and the * symbol. For example:

```
int i = 42;
int *pi = &i;
```

In this code, the variable i is an integer, and the compiler and linker will determine where this variable will be allocated. Usually, a variable in a function will be on a stack frame, as described in a later section. At runtime, the stack will be created (essentially a chunk of memory will be allocated) and space will be reserved in the stack memory for the variable i. The program then puts a value (42) in that memory. Next, the address of the memory allocated for the variable i is placed in the variable pi. The memory usage of the previous code is illustrated in the following diagram:

The pointer holds a value of 0x007ef8c (notice that the lowest byte is stored in the lowest byte in memory; this is for an x86 machine). The memory location 0x007ef8c has a value of 0x0000002a, that is, a value of 42, the value of the variable i. Since pi is also a variable, it also occupies space in memory, and in this case the compiler has put the pointer *lower* in memory than the data it points to and, in this case, the two variables are not contiguous. With variables allocated on the stack like this, you should make no assumptions about where in memory the variables are allocated, nor their location in relation to other variables.

This code assumes a 32-bit operating system, and so the pointer pi occupies 32 bits and contains a 32-bit address. If the operating system is 64 bits then the pointer will be 64 bits wide (but the integer may still be 32 bits). In this book, we will use 32-bit pointers for the simple convenience that 32-bit addresses take less typing than 64-bit addresses.

The typed pointer is declared with a * symbol and we will refer to this as an int* pointer because the pointer points to memory that holds an int. When declaring a pointer, the convention is to put the * next to the variable name rather than next to the type. This syntax emphasizes that the *type pointed* to is an int. However, it is important to use this syntax if you declare more than one variable in a single statement:

```
int *pi, i;
```

It is clear that the first variable is an int* pointer and the second is an int. The following is not so clear:

```
int* pi, i;
```

You might interpret this to mean that the type of both variables is int*, *but this is not the case*, as this declares a pointer and an int. If you want to declare two pointers, then apply * to each variable:

```
int *p1, *p2;
```

It is probably better just to declare the two pointers on separate lines.

When you apply the sizeof operator to a pointer, you will get the size of the pointer, not what it points to. Thus, on an x86 machine, sizeof(int*) will return 4; and on an x64 machine, it will return 8. This is an important observation, especially when we discuss C++ built-in arrays in a later section.

To access the data pointed to by a pointer, you must **dereference** it using the * operator:

```
int i = 42;
int *pi = &i;
int j = *pi;
```

Used like this on the right-hand side of an assignment, the dereferenced pointer gives access to the value pointed to by the pointer, so j is initialized to 42. Compare this to the declaration of a pointer, where the * symbol is also used, but has a different meaning.

The dereference operator does more than give read access to the data at the memory location. As long as the pointer does not restrict it (using the const keyword; see later), you can dereference the pointer to write to a memory location too:

```
int i = 42;
cout << i << endl;
int *pi { &i };
*pi = 99;
cout << i << endl;
```

In this code, the pointer pi points to the location in memory of the variable i (in this case, using the brace syntax). Assigning the dereferenced pointer assigns the value to the location that the pointer points to. The result is that on the last line, the variable i will have a value of 99 and not 42.

Using null pointers

A pointer could point to anywhere in the memory installed in your computer, and assignment through a dereferenced pointer means that you could potentially write over sensitive memory used by your operating system, or (through direct memory access) write to memory used by hardware on your machine. However, operating systems will usually give an executable a specific memory range that it can access, and attempts to access memory out of this range will cause an operating system memory access violation.

For this reason, you should almost always obtain pointer values using the & operator or from a call to an operating system function. You should not give a pointer an absolute address. The only exception to this is the C++ constant for an invalid memory address, nullptr:

```
int *pi = nullptr;
// code
int i = 42;
pi = &i;
// code
if (nullptr != pi) cout << *pi << endl;
```

This code initializes the pointer pi to nullptr. Later in the code, the pointer is initialized to the address of an integer variable. Still later in the code, the pointer is used, but rather than calling it immediately, the pointer is first checked to ensure that it has been initialized to a non-null value. The compiler will check to see if you are about to use a variable that has not been initialized, but if you are writing library code, the compiler will not know whether the callers of your code will use pointers correctly.

The type of constant nullptr is not an integer, it is std::nullptr_t. All pointer types can be implicitly converted to this type, so nullptr can be used to initialize variables of all pointer types.

Types of memory

In general, you can regard memory as being one of four types:

Static or global
String pool
Automatic or stack
Free store

When you declare a variable at the global level, or if you have a variable declared in a function as static, then the compiler will ensure that the variable is allocated from memory that has the same lifetime as the application--the variable is created when the application starts and deleted when the application ends.

When you use a string literal, the data will also, effectively, be a global variable, but stored in a different part of the executable. For a Windows executable, string literals are stored in the .rdata PE/COFF section of the executable. The .rdata section of the file is for readonly initialized data, and hence you cannot change the data. Visual C++ allows you to go a step further and gives you an option of **string pooling**. Consider this:

```
char *p1 { "hello" };
char *p2 { "hello" };
```

```
cout << hex;
cout << reinterpret_cast<int>(p1) << endl;
cout << reinterpret_cast<int>(p2) << endl;
```

In this code, two pointers are initialized with the address of the string literal hello. In the following two lines, the address of each pointer is printed on the console. Since the << operator for char* treats the variable as a pointer to a string, it will print the string rather than the address of the pointer. To get around this, we call the reinterpret_cast operator to convert the pointer to an integer and print the value of the integer.

If you compile the code at the command line using the Visual C++ compiler, you will see two different addresses printed. These two addresses are in the .rdata section and are both read-only. If you compile this code with the /GF switch to enable string pooling (which is default for Visual C++ projects), the compiler will see that the two string literals are the same and will only store one copy in the .rdata section, so the result of this code will be that a single address will be printed on the console twice.

In this code, the two variables p1 and p2 are automatic variables, that is, they are created on the stack created for the current function. When a function is called, a chunk of memory is allocated for the function and this contains space for the parameters passed to the function and the return address of the code that called the function, as well as space for the automatic variables declared in the function. When the function finishes, the stack frame is destroyed.

The **calling convention** of the function determines whether the calling function or the called function has the responsibility to do this. In Visual C++, the default is the __cdecl calling convention, which means the calling function cleans up the stack. The __stdcall calling convention is used by Windows operating system functions and the stack clean up is carried out by the called function. More details will be given in the next chapter.

Automatic variables only last as long as the function and the address of such variables only make any sense within the function. Later in this chapter, you will see how to create arrays of data. Arrays allocated as automatic variables are allocated on the stack to a fixed size determined at compile time. It is possible with large arrays that you could exceed the size of the stack, particularly with functions that are called recursively. On Windows, the default stack size is 1 MB, and on x86 Linux, it is 2 MB. Visual C++ allows you to specify a bigger stack with the /F compiler switch (or the /STACK linker switch). The gcc compiler allows you to change the default stack size with the --stack switch.

The final type of memory is **dynamic memory** created on the **free store** or sometimes known as the **heap**. This is the most flexible way of using memory. As the name suggests, you allocate memory at runtime of a size determined at runtime. The implementation of the free store depends on the C++ implementation but you should regard the free store as having the same lifetime as your application, so memory allocated from the free store should last at least as long as your application.

However, there are potential dangers here, particularly for long-lived applications. All memory allocated from the free store should be returned back to the free store when you have finished with it so that the free store manager can reuse the memory. If you do not return memory appropriately, then potentially the free store manager could run out of memory, which will prompt it to ask the operating system for more memory, and consequently, the memory usage of your application will grow over time, causing performance issues due to memory paging.

Pointer arithmetic

A pointer points to memory, and the type of the pointer determines the type of the data that can be accessed through the pointer. So, an int* pointer will point to an integer in memory, and you dereference the pointer (*) to get the integer. If the pointer allows it (it is not marked as const), you can change its value through pointer arithmetic. For example, you can increment or decrement a pointer. What happens to the value of the memory address depends on the type of the pointer. Since a typed pointer points to a type, any pointer arithmetic will change the pointer in units of the *size* of that type.

If you increment an int* pointer, it will point to the *next* integer in memory and the change in the memory address depends on the size of the integer. This is equivalent to array indexing, where an expression such as v[1] means you should start at the memory location of the first item in v and then move one item further in memory and return the item there:

```
int v[] { 1, 2, 3, 4, 5 };
int *pv = v;
*pv = 11;
v[1] = 12;
pv[2] = 13;
*(pv + 3) = 14;
```

The first line allocates an array of five integers on the stack and initializes the values to the numbers 1 to 5. In this example, because an initialization list is used, the compiler will create space for the required number of items, hence the size of the array is not given. If you give the size of the array between the brackets, then the initialization list must not have more items than the array size. If the list has fewer items, then the rest of the items in the array are initialized to the default value (usually zero).

The next line in this code obtains a pointer to the first item in the array. This line is significant: an array name is treated as a pointer to the first item in the array. The following lines alter array items in various ways. The first of these (*pv) changes the first item in the array by dereferencing the pointer and assigning it a value. The second (v[1]) uses array indexing to assign a value to the second item in the array. The third (pv[2]) uses indexing, but this time with a pointer, and assigns a value to the third value in the array. And the final example (*(pv + 3)) uses pointer arithmetic to determine the address of the fourth item in the array (remember the first item has an index of 0) and then dereferences the pointer to assign the item a value. After these, the array contains the values { 11, 12, 13, 14, 5 } and the memory layout is illustrated here:

If you have a memory buffer containing values (in this example, allocated via an array) and you want to multiply each value by 3, you can do this using pointer arithmetic:

```
int v[] { 1, 2, 3, 4, 5 };
int *pv = v;
for (int i = 0; i < 5; ++i)
{
*pv++ *= 3;
}
```

The loop statement is complicated, and you will need to refer back to the operator precedence given in Chapter 2, *Understanding Language Features*. The postfix increment operator has the highest precedence, the next highest precedence is the dereference operator (*), and finally, the *= operator has the lowest of the three operators, so the operators are run in this order: ++, *, *=. The postfix operator returns the value *before* the increment, so although the pointer is incremented to the next item in memory, the expression uses the address before the increment. This address is then dereferenced which is assigned by the assignment operator that replaces the item with the value multiplied by 3. This illustrates an important difference between pointers and array names; you can increment a pointer but you cannot increment an array:

```
pv += 1; // can do this
v += 1; // error
```

You can, of course use indexing (with []) on both array names and pointers.

Using arrays

As the name suggests, a C++ built-in array is zero or more items of data of the same type. In C++, square brackets are used to declare arrays and to access array elements:

```
int squares[4];
for (int i = 0; i < 4; ++i)
{
squares[i] = i * i;
}
```

The squares variable is an array of integers. The first line allocates enough memory for *four* integers and then the for loop initializes the memory with the first four squares. The memory allocated by the compiler from the stack is contiguous and the items in the array are sequential, so the memory location of squares[3] is sizeof(int) following on from squares[2]. Since the array is created on the stack, the size of the array is an instruction to the compiler; this is not dynamic allocation, so the size has to be a constant.

There is a potential problem here: the size of the array is mentioned twice, once in the declaration and then again in the for loop. If you use two different values, then you may initialize too few items, or you could potentially access memory outside the array. The ranged for syntax allows you to get access to each item in the array; the compiler can determine the size of the array and will use this in the ranged for loop. In the following code, there is a deliberate mistake that shows an issue with array sizes:

```
int squares[5];
for (int i = 0; i < 4; ++i)
{
squares[i] = i * i;
}
for(int i : squares)
{
cout << i << endl;
}
```

The size of the array and the range of the first for loop do not agree and consequently the last item will not be initialized. The ranged for loop, however, will loop through all five items and so will print out some random value for the value of the last value. What if the same code is used but the squares array is declared to have three items? It depends on the compiler you are using and whether you are compiling a debug build, but clearly you will be writing to memory *outside* of that allocated to the array.

There are some ways to mitigate these issues. The first one has been mentioned in an earlier chapter: declare a constant for the size of the array and use that whenever your code needs to know the array size:

```
constexpr int sq_size = 4;
int squares[sq_size];
for (int i = 0; i < sq_size; ++i)
{
squares[i] = i * i;
}
```

The array declaration must have a constant for the size, and that is managed by using the sq_size constant variable.

You may also want to calculate the size of an already allocated array. The sizeof operator, when applied to an array, returns the size in bytes of the *entire* array, so you can determine the size of the array by dividing this value by the size of a single item:

```
int squares[4];
```

```
for (int i = 0; i < sizeof(squares)/sizeof(squares[0]); ++i)
{
squares[i] = i * i;
}
```
This is safer code, but clearly it is verbose. The C runtime library contains a macro called _countof that performs this calculation.

Function parameters

As illustrated, there is an automatic conversion of an array to the appropriate pointer type and this occurs if you pass an array to a function, or return it from a function. This decay to a dumb pointer means that other code can make no assumption about an array size. A pointer could point to memory allocated on the stack where the memory lifetime is determined by the function, or a global variable where the memory lifetime is that of the program, or it could be to memory that is dynamically allocated and the memory is determined by the programmer. There is nothing in a pointer declaration that indicates the type of memory or who is responsible for the deallocation of the memory. Nor is there any information in a dumb pointer of how much memory the pointer points to. When you write code using pointers, you have to be disciplined about how you use them.
A function can have an array parameter, but this means a lot less than it appear to indicate:
```
// there are four tires on each car
bool safe_car(double tire_pressures[4]);
```

This function will check that each member of the array has a value between the minimum and maximum values allowed. There are four tires in use at any one time on a car, so the function *should* be called with an array of four values. The problem is that although it appears that the compiler *should* check that the array passed to the function is the appropriate size, it doesn't. You can call this function like this:
```
double car[4] = get_car_tire_pressures();
if (!safe_car(car)) cout << "take off the road!" << endl;
double truck[8] = get_truck_tire_pressures();
if (!safe_car(truck)) cout << "take off the road!" << endl;
```
Of course, it should have been obvious to the developer that a truck is not a car, and so this developer should not have written this code, but the usual advantage of a compiled language is that the compiler will perform some *sanity checks* for you. In the case of array parameters, it won't.
The reason is that the array is passed as a pointer, so although the parameter appears to be a built-in array, you cannot use facilities you are used to using with arrays like ranged for. In fact, if the safe_car function calls sizeof(tire_pressures), it will get the size of a double pointer and not 16, the size in bytes of a four int array.
This *decay to a pointer* feature of array parameters means that functions will only ever know the size of an array parameter if you explicitly tell it the size. You can use an empty pair of square brackets to indicate that the item should be passed an array, but it really is just the same as a pointer:
```
bool safe_car(double tire_pressures[], int size);
```
Here the function has a parameter that indicates the size of the array. The preceding function is exactly the same as declaring the first parameter to be a pointer. The following is not an overload of the function; it is the *same* function:
```
bool safe_car(double *tire_pressures, int size);
```
The important point is that when you pass an array to a function, the *first dimension* of the array is treated as a pointer. So far arrays have been single dimensional, but they may have more than one dimension.

Multidimensional arrays

Arrays can be multidimensional and to add another dimension you add another set of square brackets:

```
int two[2];
int four_by_three[4][3];
```

The first example creates an array of two integers, the second creates a two-dimensional array with 12 integers arranged so that there are four rows of three columns. Of course, *row* and *column* are arbitrary and treat the two-dimensional array like a conventional spreadsheet table, but it helps to visualize how the data is arranged in memory.

Note that there are square brackets around every dimension. C++ is different to other languages in this respect, so a declaration of `int x[10,10]` will be reported as an error by the C++ compiler.

Initializing multidimensional arrays involves a pair of braces and the data in the order that it will be used to initialize the dimensions:

```
int four_by_three[4][3] { 11,12,13,21,22,23,31,32,33,41,42,43 };
```

In this example, the values having the highest digit reflect the left-most index and the lower digit reflect, the right-most index (in both cases, one more than the actual index). Clearly, you can split this over several lines and use whitespace to group values together to make this more readable. You can also use nested braces. For example:

```
int four_by_three[4][3] = { {11,12,13}, {21,22,23},
{31,32,33}, {41,42,43} };
```

If you read the dimensions going left to right, you can read the initialization going into deeper levels of nesting. There are four rows, so within the outer braces there are four sets of nested braces. There are three columns, and so within the nested braces there are three initialization values.

Nested braces are not just a convenience for formatting your C++ code, because if you provide an empty pair of braces the compiler will use the default value:

```
int four_by_three[4][3] = { {11,12,13}, {}, {31,32,33}, {41,42,43} };
```

Here, the second-row items are initialized to 0.

When you increase the dimensions, the principle applies: increase the nesting for the right most dimension:

```
int four_by_three_by_two[4][3][2]
= { { {111,112}, {121,122}, {131,132} },
{ {211,212}, {221,222}, {231,232} },
{ {311,312}, {321,322}, {331,332} },
{ {411,412}, {421,422}, {431,432} }
};
```

This is four rows of three columns of pairs (as you can see, when the dimensions increase it becomes apparent that the terms **rows** and **columns** are largely arbitrary).

You access items using the same syntax:

```
cout << four_by_three_by_two[3][2][0] << endl; // prints 431
```

In terms of the memory layout, the compiler interprets the syntax in the following way. The first index determines the offset from the beginning of the array in chunks of six integers (3 * 2), the second index indicates the offset within one of these six integer *chunks* itself in chunks of two integers, and the third index is the offset in terms of individual integers. Thus `[3][2][0]` is *(3 * 6) + (2 * 2) + 0 = 22* integers from the beginning, treating the first integer as index zero.

A multidimensional array is treated as arrays of arrays, so the type of each "row" is `int[3][2]` and we know from the declaration that there are four of them.

Passing multidimensional arrays to functions

You can pass a multidimensional array to a function:

```
// pass the torque of the wheel nuts of all wheels
```

```
bool safe_torques(double nut_torques[4][5]);
```

This compiles and you can access the parameter as a 4x5 array, assuming that this vehicle has four wheels with five nuts on each one.

As stated earlier, when you pass an array, the first dimension will be treated as a pointer, so while you can pass a 4x5 array to this function, you can also pass a 2x5 array and the compiler will not complain. However, if you pass a 4x3 array (that is, the second dimension is not the same as declared in the function), the compiler will issue an error that the array is incompatible. The parameter may be more accurately described as being double row[][5]. Since the size of the first dimension is not available, the function should be declared with the size of that dimension:

```
bool safe_torques(double nut_torques[][5], int num_wheels);
```

This says that nut_torques is one or more "rows", each of which has five items. Since the array does not provide information about the number of rows it has, you should provide it. Another way to declare this is:

```
bool safe_torques(double (*nut_torques)[5], int num_wheels);
```

The brackets are important here, if you omit them and use double *nut_torques[5], then it means the * will refer to the type in the array, that is, the compiler will treat nut_torques as a five-element array of double* pointers. We have seen an example of such an array before:

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

The argv parameter is an array of char* pointers. You can also declare the argv parameter as char** which has the same meaning.

In general, if you intend to pass arrays to a function it is best to use custom types, or use the C++ array types.

Using ranged for with multidimensional arrays is a bit more complicated than appears on first sight, and requires the use of a reference as explained in the section later in this chapter.

Using arrays of characters

Strings will be covered in more detail in Chapter 9, *Using Strings*, but it is worth pointing out here that C strings are arrays of characters and are accessed through pointer variables. This means that if you want to manipulate strings, you must manipulate the memory that the pointer points to, and not manipulate the pointer itself.

Comparing strings

The following allocates two string buffers and it calls the strcpy_s function to initialize each with the same string:

```
char p1[6];
strcpy_s(p1, 6, "hello");
char p2[6];
strcpy_s(p2, 6, p1);
bool b = (p1 == p2);
```

The strcpy_c function will copy characters from the pointer given in the last parameter (until the terminating NUL), into the buffer given in the first parameter, whose maximum size is given in the second parameter. These two pointers are compared in the final line, and this will return a value of false. The problem is that the compare function is comparing the values of the pointers, not what the pointers point to. The two buffers have the same string, but the pointers are different, so b will be false.

The correct way to compare strings is to compare the data character by character to see if they are equal. The C runtime provides strcmp that compares two string buffers character by character, and the std::string class defines a function called compare that will also perform such a comparison; however, be wary of the value returned from these functions:

```
string s1("string");
string s2("string");
int result = s1.compare(s2);
```

The return value is not a bool type indicating if the two strings are the same; it is an int. These compare functions carry out a lexicographical compare and return a negative value if the parameter (s2 in this code) is greater than the operand (s1) lexicographically, and a positive number if the operand is greater than the parameter. If the two strings are the same, the function returns 0. Remember that a bool is false for a value of 0 and true for non-zero values. The standard library provides an overload for the == operator for std::string, so it is safe to write code like this:

```
if (s1 == s2)
{
cout << "strings are the same" << endl;
}
```

The operator will compare the strings contained in the two variables.

Preventing buffer overruns

The C runtime library for manipulating strings is notorious for allowing buffer overruns. For example, the strcpy function copies one string to another, and you get access to this through the <cstring> header, which is included by the <iostream> header. You may be tempted to write something like this:

```
char pHello[5]; // enough space for 5 characters
strcpy(pHello, "hello");
```

The problem is that strcpy will copy all the character up to, and including the terminating NULL character and so you will be copying six characters into an array with space for only *five*. You could be taking a string from the user input (say, from a text box on a web page) and think that the array you have allocated is big enough, but a malicious user could provide an excessively long string deliberately bigger than the buffer so that it overwrites other parts of your program. Such *buffer overruns* have caused a lot of programs to be subjected to hackers taking control of servers, so much so that the C string functions have all been replaced by safer versions. Indeed, if you are tempted to type the preceding code, you'll find that strcpy is available, but the Visual C++ compiler will issue an error:

```
error C4996: 'strcpy': This function or variable may be unsafe.
Consider using strcpy_s instead. To disable deprecation, use
_CRT_SECURE_NO_WARNINGS. See online help for details.
```

If you have existing code that uses strcpy, and you need to make that code compile, you can define the symbol before <cstring>:

```
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
```

An initial attempt to prevent this issue is to call strncpy, which will copy a specific number of characters:

```
char pHello[5]; // enough space for 5 characters
strncpy(pHello, "hello", 5);
```

The function will copy up to five characters and then stop. The problem is that the string to copy has five characters and so the result will be no NULL termination. The safer version of this function has a parameter that you can use to say how big the destination buffer is:

```
size_t size = sizeof(pHello)/sizeof(pHello[0]);
strncpy_s(pHello, size, "hello", 5);
```

At runtime this will still cause a problem. You have told the function that the buffer is five characters in size and it will determine that this is not big enough to hold the six characters that you have asked it to copy. Rather than allowing the program to silently continue and

the buffer overrun to cause problems, the safer string functions will call a function called the **constraint handler** and the default version will shut down the program on the rationale that a buffer overrun means that the program is compromised.
The C runtime library strings functions were originally written to return the result of the function, the safer versions now return an error value. The strncpy_s function can also be told to truncate the copy rather than call the constraint handler:

```
strncpy_s(pHello, size, "hello", _TRUNCATE);
```

The C++ string class protects you from such issues.

Using pointers in C++

Pointers are clearly very important in C++, but as with any powerful feature, there are issues and dangers, so it is worth pointing out some of the major issues. A pointer points to a single location in memory, and the type of the pointer indicates how the memory location should be interpreted. The very most you can assume is the number of bytes at that position in memory is the size of the type of the pointer. That's it. This means that pointers are inherently unsafe. However, in C++ they are the quickest way to enable code within your process to access large amounts of data.

Accessing out of bounds

When you allocate a buffer, whether on the stack or on the free store, and you get a pointer, there is little to stop you from accessing memory you have not allocated--either before or after the position of the buffer. This means that when you use pointer arithmetic, or indexed access on arrays, that you check carefully that you are not going to access data out of bounds. Sometimes the error may not be immediately obvious:

```
int arr[] { 1, 2, 3, 4 };
for (int i = 0; i < 4; ++i)
{
arr[i] += arr[i + 1]; // oops, what happens when i == 3?
}
```

When you use indexing, you have to keep reminding yourself that arrays are indexed from zero so the highest index is the size of the array minus 1.

Pointers to deallocated memory

This applies to memory allocated on the stack and to memory dynamically allocated. The following is a poorly written function that returns a string allocated on the stack in a function:

```
char *get()
{
char c[] { "hello" };
return c;
}
```

The preceding code allocates a buffer of six characters and then initializes it with the five characters of the string literal hello, and the NULL termination character. The problem is that once the function finishes the stack frame is torn down so that the memory can be reused, and the pointer will point to memory that could be used by something else. This error is caused by poor programming, but it may not be as obvious as in this example. If the function uses several pointers and performs a pointer assignment, you may not immediately notice that you have returned a pointer to a stack-allocated object. The best course of action is simply not to return raw pointers from functions, but if you do want to use this style of programming, make sure that the memory buffer is passed in through a parameter (so the function does not own the buffer) or is dynamically allocated and you are passing ownership to the caller.
This leads on to another issue. If you call delete on a pointer and then later in your code,

try to access the pointer, you will be accessing memory that is potentially being used by other variables. To alleviate this problem, you can get into the habit of assigning a pointer to null_ptr when you delete it and check for null_ptr before using a pointer. Alternatively, you can use a smart pointer object which will do this for you. Smart pointers will be covered in Chapter 6, *Classes*.

Converting pointers

You can either have typed pointers, or the void* pointer. Typed pointers will access the memory as if it is the specified type (this has interesting consequences when you have inheritance with classes, but that will be left for Chapter 6, *Classes* and Chapter 7, *Introduction to Object-Oriented Programming*). Thus, if you cast a pointer to a different type and dereference it, the memory will be treated as containing the cast type. It rarely makes sense to do this. The void* pointer cannot be dereferenced, so you can never access data through a void* pointer, to access the data you have to cast the pointer.

The whole reason for the void* pointer type is that it can point to anything. In general, void* pointers should only be used when the type does not matter to that function. For example, the C malloc function returns a void* pointer because the function merely allocates memory; it does not care what that memory will be used for.

Constant pointers

Pointers can be declared as const which, depending on where you apply it, means that the memory the pointer points to is read-only through the pointer, or the value of the pointer is read-only:

```
char c[] { "hello" }; // c can be used as a pointer
*c = 'H'; // OK, can write thru the pointer
const char *ptc {c}; // pointer to constant
cout << ptc << endl; // OK, can read the memory pointed to
*ptc = 'Y'; // cannot write to the memory
char *const cp {c}; // constant pointer
*cp = 'y'; // can write thru the pointer
cp++; // cannot point to anything else
```

Here, ptc is a pointer to constant char, that is, although you can change what ptc points to, and you can read what it points to, you cannot use it to change the memory. On the other hand, cp is a constant pointer, which means you can both read and write the memory which the pointer points to, but you cannot change where it points to. It is typical to pass the const char* pointers to functions because the functions do not know where the string has been allocated or the size of the buffer (the caller may pass a literal which cannot be changed). Note that there is no const* operator so char const* is treated as const char*, a pointer to a constant buffer.

You can make a pointer constant, change it, or remove it using casts. The following does some fairly pointless changing around of the const keyword to prove the point:

```
char c[] { "hello" };
char *const cp1 { c }; // cannot point to any other memory
*cp1 = 'H'; // can change the memory
const char *ptc = const_cast<const char*>(cp1);
ptc++; // change where the pointer points to
char *const cp2 = const_cast<char *const>(ptc);
*cp2 = 'a'; // now points to Hallo
```

The pointers cp1 and cp2 can be used to change the memory they point to, but once assigned neither can point to other memory. The first const_cast casts away the constness to a pointer that can be changed to point to other memory, but cannot be used to alter that memory, ptc. The second const_cast casts away the const-ness of ptc so that the memory can be changed through the pointer, cp2.

Changing the type pointed to

The static_cast operator is used to convert with a compile time check, but not a runtime check, so this means that the pointers must be related. The void* pointer can be converted to any pointer, so the following compiles and makes sense:

```
int *pi = static_cast<int*>(malloc(sizeof(int)));
*pi = 42;
cout << *pi << endl;
free(pi);
```

The C malloc function returns a void* pointer so you have to convert it to be able to use the memory. (Of course, the C++ new operator removes the need for such casting.) The builtin types are not "related" enough for static_cast to convert between pointer types, so you cannot use static_cast to convert an int* pointer to a char* pointer, even though int and char are both integer types. For custom types that are related through inheritance, you can cast pointers using static_cast, but there is no runtime check that the cast is correct. To cast with runtime checks you should use dynamic_cast, and more details will be given in Chapters 6, *Classes* and Chapter 7, *Introduction to Object-Oriented Programming*.

The reinterpret_cast operator is the most flexible, and dangerous, of the cast operators because it will convert between any pointer types without any type checks. It is inherently unsafe. For example, the following code initializes a wide character array with a literal. The array wc will have six characters, hello followed by NULL. The wcout object interprets a wchar_t* pointer as a pointer to the first character in a wchar_t string, so inserting wc will print the string (every character until the NUL). To get the actual memory location, you have to convert the pointer to an integer:

```
wchar_t wc[] { L"hello" };
wcout << wc << " is stored in memory at ";
wcout << hex;
wcout << reinterpret_cast<int>(wc) << endl;
```

Similarly, if you insert a wchar_t into the wcout object, it will print the character, not the numeric value. So, to print out the codes for the individual characters, we need to cast the pointer to a suitable integer pointer. This code assumes that a short is the same size as a wchar_t:

```
wcout << "The characters are:" << endl;
short* ps = reinterpret_cast<short*>(wc);
do
{
wcout << *ps << endl;
} while (*ps++);
```

Allocating memory in code

C++ defines two operators, new and delete, that allocate memory from the free store and release memory back into the free store.

Allocating individual objects

The new operator is used with the type to allocate memory, and it will return a typed pointer to that memory:

```
int *p = new int; // allocate memory for one int
```

The new operator will call the *default constructor* for custom types for every object it creates (as explained in Chapter 6, *Classes*). Built-in types do not have constructors, so instead a type initialization will occur and this will usually initialize the object to zero (in this example, a zero integer).

In general, you should not use memory allocated for built-in types without explicitly initializing it. In fact, in Visual C++ the debug version of the new operator will initialize

memory to a value of 0xcd for every byte, as a visual reminder in the debugger that you have not initialized the memory. For custom types, it is left to the author of the type to initialize allocated memory.
It is important that when you have finished with memory that you return it back to the free store so that the allocator can reuse it. You do this by calling the delete operator:

```
delete p;
```

When you delete a pointer, the **destructor** for the object is called. For built-in types, this does nothing. It is good practice to initialize a pointer to nullptr, after you have deleted it, and if you use the convention of checking the value of a pointer before using it, this will protect you from using a deleted pointer. The C++ standard says that the delete operator will have no effect if you delete a pointer that has a value of nullptr.
C++ allows you to initialize a value at the time you call the new operator, in two ways:

```
int *p1 = new int (42);
int *p2 = new int {42};
```

For a custom type, the new operator will call a constructor on the type; for a built in type, the end result is the same, and is carried out by initializing the item to the value provided. You can also use initialized list syntax, as shown in the second line in the preceding code. It is important to note that the initialization is the memory pointed to, not the pointer variable.

Allocating arrays of objects

You can also create arrays of objects in dynamic memory using the new operator. You do this by providing the number of items you want created in a pair of square brackets. The following code allocates memory for two integers:

```
int *p = new int[2];
p[0] = 1;
*(p + 1) = 2;
for (int i = 0; i < 2; ++i) cout << p[i] << endl;
delete [] p;
```

The operator returns a pointer to the type allocated, and you can use pointer arithmetic or array indexing to access the memory. You cannot initialize the memory in the new statement; you have to do that after creating the buffer. When you use new to create a buffer for more than one object, you must use the appropriate version of the delete operator: the [] is used to indicate that more than one item is deleted and the destructor for each object will be called. It is important that you always use the right version of delete appropriate to the version of new used to create the pointer.
Custom types can define their own operator new and operator delete for individual objects, as well as operator new[] and operator delete[] for arrays of objects. The custom type author can use these to use custom memory allocation schemes for their objects.

Handling failed allocations

If the new operator cannot allocate the memory for an object, it will throw the std::bad_alloc exception and the pointer returned will be nullptr. Exceptions are covered in Chapter 10, *Diagnostics and Debugging*, so only a brief outline of the syntax will be given here. It is important that you check for failure to allocate memory in production code. The following code shows how to guard the allocation so that you can catch the std::bad_alloc exception and handle it:

```
// VERY_BIG_NUMER is a constant defined elsewhere
int *pi;
try
{
pi = new int[VERY_BIG_NUMBER];
// other code
}
catch(const std::bad_alloc& e)
{
cout << "cannot allocate" << endl;
return;
}
// use pointer
delete [] pi;
```

If any code in the try block throws an exception control it is passed to the catch clause, ignoring any other code that has not been executed yet. The catch clause checks the type of the exception object and if it is the correct type (in this case an allocation fault), it creates a reference to that object and passes control to the catch block, and the scope of the exception reference is this block. In this example, the code merely prints an error, but you would use it to take action to ensure that the memory allocation failure does not affect subsequent code.

Using other versions of the new operator

Further, a custom type can define a placement operator new, which allows you to provide one or more parameters to the custom new function. The syntax of the placement new is to provide the placement fields through parentheses.

The C++ Standard Library version of the new operator provides a version that can take the constant std::nothrow as a placement field. This version will not throw an exception if the allocation fails, instead, the failure can only be assessed from the value of the returned pointer:

```
int *pi = new (std::nothrow) int [VERY_BIG_NUMBER];
if (nullptr == pi)
{
cout << "cannot allocate" << endl;
}
else
{
// use pointer
delete [] pi;
}
```

The parentheses before the type are used to pass placement fields. If you use parentheses after the type, these will give a value to initialize the object if the allocation is successful.

Memory lifetime

The memory allocated by new will remain valid until you call delete. This means that you may have memory with long lifetimes, and the code may be passed around various functions in your code. Consider this code:

```
int *p1 = new int(42);
int *p2 = do_something(p1);
delete p1;
p1 = nullptr;
// what about p2?
```

This code creates a pointer and initializes the memory it points to and then passes the pointer to a function, which itself returns a pointer. Since the p1 pointer is no longer needed, it is deleted and assigned to nullptr so that it cannot be used again. This code looks fine, but the problem is what do you do with the pointer returned by the function? Imagine that the function simply manipulates the data pointed to by the pointer:

```
int *do_something(int *p)
{
*p *= 10;
return p;
}
```

In effect, calling do_something creates a copy of a pointer, but not a copy of what it points to. This means that when the p1 pointer is deleted, the memory it points to is no longer available, and so the pointer p2 points to the invalid memory.
This problem can be addressed using a mechanism called **Resource Acquisition Is Initialization** (**RAII**), which means using the features of C++ objects to manage resources. RAII in C++ needs classes and in particular, copy constructors and destructors. A smart pointer class can be used to manage a pointer so that when it is copied, the memory it points to is also copied. A destructor is a function that is called automatically when the object goes out of scope and so a smart pointer can use this to free memory. Smart pointers and destructors will be covered in Chapter 6, *Classes*.

The Windows SDK and pointers

Returning a pointer from a function has its inherent dangers: the responsibility for the memory is passed to the caller, and the caller must ensure that the memory is appropriately de-allocated, otherwise this could cause a memory leak with a corresponding loss of performance. In this section, we will look at some ways that the Window's **Software Development Kit** (**SDK**) provides access to memory buffers and learn some techniques used in C++.
First, it is worth pointing out that any function in the Windows SDK that returns a string, or has a string parameter, will come in two versions. The version suffixed with A indicates that the function uses ANSI strings, and the W version will use wide character strings. For the purpose of this discussion, it is easier to use the ANSI functions.
The GetCommandLineA function has the following prototype (taking into account the Windows SDK typedef):

```
char * __stdcall GetCommandLine();
```

All Windows functions are defined as using the __stdcall calling convention. Usually, you will see the typedef of WINAPI used for the __stdcall calling convention.
The function can be called like this:

```
//#include <windows.h>
cout << GetCommandLineA() << endl;
```

Notice that we are making no effort to do anything about freeing the returned buffer. The reason is that the pointer points to memory that lives the lifetime of your process, so you *should not* release it. Indeed, if you were to release it, how would you do it? You cannot guarantee that the function was written with the same compiler, or the same libraries that you are using, so you cannot use the C++ delete operator or the C free function.
When a function returns a buffer, it is important to consult the documentation to see who allocated the buffer, and who should release it.
Another example is GetEnvironmentStringsA:

```
char * __stdcall GetEnvironmentStrings();
```

This also returns a pointer to a buffer, but this time the documentation is clear that after using the buffer you should release it. The SDK provides a function to do this called FreeEnvironmentStrings. The buffer contains one string for each environment variable in the form name=value and each string is terminated by a NUL character. The last string in the buffer is simply a NUL character, that is, there are two NUL characters at the end of the buffer. These functions can be used like this:

```
char *pBuf = GetEnvironmentStringsA();
if (nullptr != pBuf)
{
char *pVar = pBuf;
while (*pVar)
{
```

```
cout << pVar << endl;
pVar += strlen(pVar) + 1;
}
FreeEnvironmentStringsA(pBuf);
}
```

The strlen function is part of the C runtime library and it returns the length of a string. You do not need to know how the GetEnvironmentStrings function allocates the buffer because the FreeEnvironmentStrings will call the correct deallocation code.

There are cases when the developer has the responsibility of allocating a buffer. The Windows SDK provides a function called GetEnvironmentVariable to return the value of a named environment variable. When you call this function, you do not know if the environment variable is set, or if it is set, or how big its value is, so this means that you will most likely have to allocate some memory. The prototype of the function is:

```
unsigned long __stdcall GetEnvironmentVariableA(const char *lpName,
char *lpBuffer, unsigned long nSize);
```

There are two parameters that are pointers to C strings. There is a problem here, a char* pointer could be passing *in* a string to the function, or it could be used to pass in a buffer for a string to be returned *out*. How do you know what a char* pointer is intended to be used for?

You are given a clue with the full parameter declaration. The lpName pointer is marked const so the function will not alter the string it points to; this means that it is an *in* parameter. This parameter is used to pass in the name of the environment variable you want to obtain. The other parameter is simply a char* pointer, so it could be used to pass a string *in* to the function or *out*, or indeed, both *in* and *out*. The only way to know how to use this parameter is to read the documentation. In this case, it is an *out* parameter; the function will return the value of the environment variable in lpBuffer if the variable exists, or if the variable does not exist, the function will leave the buffer untouched and return the value 0. It is your responsibility to allocate this buffer in whatever way you see fit, and you pass the size of this buffer in the last parameter, nSize.

The function's return value has two purposes. It is used to indicate that an error has occurred (just one value, 0, which means you have to call the GetLastError function to get the error), and it is also used to give you information about the buffer, lpBuffer. If the function succeeds, then the return value is the number of characters copied into the buffer excluding the NULL terminating character. However, if the function determines that the buffer is too small (it knows the size of the buffer from the nSize parameter) to hold the environment variable value, no copy will happen, and the function will return the required size of the buffer, which is the number of characters in the environment variable including the NULL terminator.

A common way to call this function is to call it twice, first with a zero-sized buffer and then use the return value to allocate a buffer before calling it again:

```
unsigned long size = GetEnvironmentVariableA("PATH", nullptr, 0);
if (0 == size)
{
cout << "variable does not exist " << endl;
}
else
{
char *val = new char[size];
if (GetEnvironmentVariableA("PATH", val, size) != 0)
{
cout << "PATH = ";
cout << val << endl;
}
delete [] val;
}
```

In general, as with all libraries, you have to read the documentation to determine how the parameters are used. The Windows documentation will tell you if a pointer parameter is in, out, or in/out. It will also tell you who owns the memory and whether you have the responsibility for allocating and/or freeing the memory.

Whenever you see a pointer parameter for a function, take special care to check the documentation as to what the pointer is used for and how the memory is managed.

Memory and the C++ Standard Library

The C++ Standard Library provides various classes to allow you to manipulate collections of objects. These classes, called the **Standard Template Library** (**STL**), provide a standard way to insert items into collection objects and ways to access the items and iterate through entire collections (called iterators). The STL defines collection classes that are implemented as queues, stacks, or as vectors with random access. These classes will be covered in depth Chapter 8, *Using the Standard Library Containers*, so in this section we will limit the discussion to just two classes that behave like C++ built in arrays.

Standard Library arrays

The C+ Standard Library provides two containers that give random access via an indexer to the data. These two containers also allow you to access the underlying memory and since they guarantee to store the items sequentially and contiguous in memory, they can be used when you are required to provide a pointer to a buffer. These two types are both templates, which means that you can use them to hold built-in and custom types. These two collection classes are array and vector.

Using the stack-based array class

The array class is defined in the <array> header file. The class allows you to create fixed sized arrays on the stack and, as with built-in arrays, they cannot shrink or expand at runtime. Since they are allocated on the stack, they do not require a call to a memory allocator at runtime, but clearly, they should be smaller than the stack frame size. This means that an array is a good choice for small arrays of items. The size of an array must be known at compile time and it is passed as a template parameter:

```
array<int, 4> arr { 1, 2, 3, 4 };
```

In this code, the first template parameter in the angle brackets (<>) is the type of each item in the array, and the second parameter is the number of items. This code initializes the array with an initialize list, but note that you still have to provide the size of the array in the template. This object will work like a built-in array (or indeed, any of the Standard Library containers) with ranged for:

```
for (int i : arr) cout << i << endl;
```

The reason is that array implements the begin and end functions that are required for this syntax. You can also use indexing to access items:

```
for (int i = 0; i < arr.size(); ++i) cout << arr[i] << endl;
```

The size function will return the size of the array and the square bracket indexer gives random access to members of the array. You can access memory outside of the bounds of the array, so for the previously defined array that has four members, you can access arr[10]. This may cause unexpected behavior at runtime, or even some kind of memory fault. To guard against this, the class provides a function, at, which will perform a range check and if the index is out of range the class will throw the C++ exception out_of_range.

The main advantage of using an array object is that you get compile time checks to see if you are inadvertently passing the object to a function as a dumb pointer. Consider this function:

```
void use_ten_ints(int*);
```

At runtime, the function does not know the size of the buffer passed to it, and in this case the documentation says that you must pass a buffer with 10 int type variables, but, as we have seen, C++ allows a built-in array to be used as a pointer:

```
int arr1[] { 1, 2, 3, 4 };
use_ten_ints(arr1); // oops will read past the end of the buffer
```

There is no compiler check, nor any runtime check to catch this error. The array class will not allow such an error to happen because there is no automatic conversion into a dumb pointer:

```
array<int, 4> arr2 { 1, 2, 3, 4 };
use_ten_ints(arr2); // will not compile
```

If you really insist in obtaining a dumb pointer, you can do this and be guaranteed to have access to the data as a contiguous block of memory where the items are stored sequentially:

```
use_ten_ints(&arr2[0]); // compiles, but on your head be it
use_ten_ints(arr2.data()); // ditto
```

The class is not just a wrapper around a built-in array, it also provides some additional functionality. For example:

```
array<int, 4> arr3;
arr3.fill(42); // put 42 in each item
arr2.swap(arr3); // swap items in arr2 with items in arr3
```

Using the dynamically allocated vector class

The Standard Library also provides the vector class in the <vector> header. Again, this class is a template, so you can use it with built-in and custom types. However, unlike array, the memory is dynamically allocated, which means that a vector can be expanded or shrunk at runtime. The items are stored contiguously so you can access the underlying buffer by calling the data function or accessing the address of the first item (to support resizing the collection, the buffer may change, so such pointers should only be used temporarily). And, of course, as with array, there is no automatic conversion to a dumb pointer. The vector class provides indexed random access with square bracket syntax and a range check with the at function. The class also implements the methods to allow the container to be used with Standard Library functions and with ranged for.

The vector class has more flexibility than the array class because you can insert items, and move items around, but this does come with some overhead. Because instances of the class allocate memory dynamically at runtime there is a cost of using an allocator and there is some extra overhead in initialization and destruction (when the vector object goes out of scope). Objects of the vector class also take more memory than the data it holds. For this reason, it is not suitable for small numbers of items (when array is a better choice).

References

A reference is an alias to an object. That is, it is another name for the object, and so access to the object is the same through a reference as it is through the object's variable name. A reference is declared using a & symbol on the reference name and it is initialized and accessed in exactly the same way as a variable:

```
int i = 42;
```

```
int *pi = &i; // pointer to an integer
int& ri1 = i; // reference to a variable
i = 99; // change the integer thru the variable
*pi = 101; // change the integer thru the pointer
ri1 = -1; // change the integer thru the reference
int& ri2 {i}; // another reference to the variable
int j = 1000;
```

```
pi = &j; // point to another integer
```
In this code, a variable is declared and initialized, then a pointer is initialized to point to this data, and a reference is initialized as an alias for the variable. Reference ri1 is initialized with an assignment operator, whereas reference ri2 is initialized using initializer list syntax.

The pointer and reference have two different meanings. The reference is not initialized to the value of the variable, the variable's data; it is an alias for the variable name.

Wherever the variable is used, the reference can be used; whatever you do to the reference is actually the same as performing the same operation on the variable. A pointer points to data, so you can change the data by dereferencing the pointer, but equally so, you can make the pointer point to any data and change that data by dereferencing the pointer (this is illustrated in the last two lines of the preceding code). You can have several aliases for a variable, and each must be initialized to the variable at the declaration. Once declared, you cannot make a reference refer to a different object.

The following code will not compile:
```
int& r1; // error, must refer to a variable
int& r2 = nullptr; // error, must refer to a variable
```
Since a reference is an alias for another variable, it cannot exist without being initialized to a variable. Likewise, you cannot initialize it to anything other than a variable name, so there is no concept of a *null reference*.

Once initialized, a reference is only ever an alias to the one variable. Indeed, when you use a reference as an operand to any operator, the operation is performed on the variable:
```
int x = 1, y = 2;
int& rx = x; // declaration, means rx is an alias for x
rx = y; // assignment, changes value of x to the value of y
```

In this code, rx is an alias to the variable x, so the assignment in the last line simply assigns x with the value of y: the assignment is performed on the aliased variable. Further, if you take the address of a reference, you are returned the address of the variable it references. While you can have a reference to an array, you cannot have an array of references.

Constant references

The reference used so far allows you to change the variable it is an alias for, therefore it has lvalue semantics. There are also const lvalue references, that is, a reference to an object that you can read, but not write to.

As with const pointers, you declare a const reference using the const keyword on a lvalue reference. This essentially makes the reference read-only: you can access the variable's data to read it, but not to change it.
```
int i = 42;
const int& ri = i;
ri = 99; // error!
```

Returning references

Sometimes an object will be passed to a function and the semantics of the function is that the object should be returned. An example of this is the << operator used with the stream objects. Calls to this operator are *chained*:
```
cout << "The value is " << 42;
```
This is actually a series of calls to functions called operator<<, one that takes a const char* pointer, and another that takes an int parameter. These functions also have an ostream parameter for the stream object that will be used. However, if this is simply an ostream parameter then it would mean that a copy of the parameter would be made, and the insertion would be performed on the copy. Stream objects often use buffering, so changes to a copy of a stream object may not have the desired effect. Further, to enable the

chaining of the insertion operators, the insertion functions will return the stream object passed as a parameter. The intention is to pass the same stream object through multiple function calls. If such a function returned an object then it would be a copy and not only would this means that a series of insertions would involve lots of copies being made, these copies would also be temporary and so any changes to the stream (for example, manipulators such as std::hex) would not persist. To address these issues, references are used. A typical prototype of such a function is:

```
ostream& operator<<(ostream& _Ostr, int _val);
```

Clearly you have to be careful about returning a reference since you have to ensure that the object lifetime lasts as long as the reference. This operator<< function will return the reference passed in the first parameter, but in the following code a reference is returned to an automatic variable:

```
string& hello()
{
string str ("hello");
return str; // don't do this!
} // str no longer exists at this point
```

In the preceding code, the string object only lives as long as the function, so the reference returned by this function will refer to an object that does not exist. Of course, you can return a reference to a static variable declared in a function.
Returning a reference from a function is a common idiom, but whenever you consider doing this make sure that the lifetime of the aliased variable is not the scope of the function.

Temporaries and references

The lvalue references must refer to a variable, but C++ has some odd rules when it comes to const references declared on the stack. If the reference is a const, the compiler will extend the lifetime of a temporary for the lifetime of the reference. For example, if you use the initialization list syntax, the compiler will create a temporary:

```
const int& cri { 42 };
```

In this code, the compiler will create a temporary int and initialize it to a value and then alias it to the cri reference (it is important that this reference is const). The temporary is available through the reference while it is in scope. This may look a little odd, but consider using a const reference in this function:

```
void use_string(const string& csr);
```

You can call this function with a string variable, a variable that will explicitly convert to a string or with a string literal:

```
string str { "hello" };
use_string(str); // a std::string object
const char *cstr = "hello";
use_string(cstr); // a C string can be converted to a std::string
use_string("hello"); // a literal can be converted to a std::string
```

In most cases, you'll not want to have a const reference to a built-in type, but with custom types where there will be an overhead in making copies there is an advantage and, as you can see here, the compiler will fall back to creating a temporary if required.

The rvalue references

C++11 defines a new type of reference, rvalue references. Prior to C++11, there was no way that code (like an assignment operator) could tell if the rvalue passed to it was a temporary object or not. If such a function is passed a reference to an object, then the function has to be careful not to change the reference because this would affect the object it refers to. If the reference is to a temporary object, then the function can do what it likes to the temporary object because the object will not live after the function completes. C++11 allows you to

write code specifically for temporary objects, so in the case of the assignment, the operator for temporary objects can just *move* the data from the temporary into the object being assigned. In contrast, if the reference is not to a temporary object then the data will have to be *copied*. If the data is large, then this prevents a potentially expensive allocation and copy. This enables so-called *move semantics*.
Consider this rather contrived code:

```
string global{ "global" };
string& get_global()
{
return global;
}
string& get_static()
{
static string str { "static" };
return str;
}
string get_temp()
{
return "temp";
}
```

The three functions return a string object. In the first two cases, the string has the lifetime of the program and so a reference can be returned. In the last function, the function returns a string literal, so a temporary string object is constructed. All three can be used to provide a string value. For example:

```
cout << get_global() << endl;
cout << get_static() << endl;
cout << get_temp() << endl;
```

All three can provide a string that can be used to assign a string object. The important point is that the first two functions return along a lived object, but the third function returns a temporary object, but these objects can be used the same.
If these functions returned access to a large object, you would not want to pass the object to another function, so instead, in most cases, you'll want to pass the objects returned by these functions as references. For example:

```
void use_string(string& rs);
```

The reference parameter prevents another copy of the string. However, this is just half of the story. The use_string function could manipulate the string. For example, the following function creates a new string from the parameter, but replaces the letters a, b, and o with an underscore (indicating the gaps in words without those letters, replicating what life would be like without donations of the blood types A, B, and O). A simple implementation would look like this:

```
void use_string(string& rs)
{
string s { rs };
for (size_t i = 0; i < s.length(); ++i)
{
if ('a' == s[i] || 'b' == s[i] || 'o' == s[i])
s[i] = '_';
}
cout << s << endl;
}
```

The string object has an index operator ([]), so you can treat it like an array of characters, both reading the values of characters and assigning values to character positions. The size of the string is obtained through the length function, which returns an unsigned int (typedef to size_t). Since the parameter is a reference, it means that any change to the string will be reflected in the string passed to the function. The intention of this code is to leave other variables intact, so it first makes a copy of the parameter. Then on the copy, the code iterates through all of the characters changing the a, b, and o characters to an underscore before printing out the result.

This code clearly has a copy overhead--creating the string, s, from the reference, rs; but this is necessary if we want to pass strings like those from get_global or get_static to this function because otherwise the changes would be made to the actual global and static variables.

However, the temporary string returned from get_temp is another situation. This temporary object only exists until the end of the statement that calls get_temp. Thus, it is possible to make changes to the variable knowing that it will affect nothing else. This means that you can use move semantics:

```
void use_string(string&& s)
{
for (size_t i = 0; i < s.length(); ++i)
{
if ('a' == s[i] || 'b' == s[i] || 'o' == s[i]) s[i] = '_';
}
cout << s << endl;
}
```

There are just two changes here. The first is that the parameter is identified as an rvalue reference using the && suffix to the type. The other change is that the changes are made on the object that the reference refers to because we know that it is a temporary and the changes will be discarded, so it will affect no other variables. Note that there are now *two* functions, overloads with the same name: one with an lvalue reference, and one with an rvalue reference. When you call this function, the compiler will call the right one according to the parameter passed to it:

```
use_string(get_global()); // string& version
use_string(get_static()); // string& version
use_string(get_temp()); // string&& version
use_string("C string"); // string&& version
string str{"C++ string"};
use_string(str); // string& version
```

Recall that get_global and get_static return references to objects that will live the lifetime of the program, and for this reason the compiler chooses the use_string version that takes an lvalue reference. The changes are made on a temporary variable within the function, and this has a copy overhead. The get_temp returns a temporary object and so the compiler calls the overload of use_string that takes an rvalue reference. This function alters the object that the reference refers to, but this does not matter because the object will not last beyond the semicolon at the end of the line. The same can be said for calling use_string with a C-like string literal: the compiler will create a temporary string object and call the overload that has an rvalue reference parameter. In the final example in this code, a C++ string object is created on the stack and passed to use_string.

The compiler sees that this object is an lvalue and potentially can be altered, so it calls the overload that takes an lvalue reference that is implemented in a way that only alters a temporary local variable in the function.

This example shows that the C++ compiler will detect when a parameter is a temporary object and will call the overload with an rvalue reference. Typically, this facility is used when writing *copy constructors* (special functions used to create a new custom type from an existing instance) and assignment operators so that these functions can implement the lvalue reference overload to copy the data from the parameter, and the rvalue reference overload to move the data from the temporary to the new object. Other uses are for writing custom types that are *move only*, where they use resources that cannot be copied, for example file handles.

Ranged for and references

As an example of what you can do with references, it is worth looking at the ranged for facility in C++11. The following code is quite straightforward; the array squares is initialized with the squares of 0 to 4:

```
constexpr int size = 4;
int squares[size];
for (int i = 0; i < size; ++i)
{
squares[i] = i * i;
}
```

The compiler knows the size of the array so you can use ranged for to print out the values in the array. In the following, on each iteration, the local variable j is a copy of the item in the array. As a copy, it means that you can read the value, but any changes made to the variable will not be reflected to the array. So, the following code works as expected; it prints out the contents of the array:

```
for (int j : squares)
{
cout << J << endl;
}
```

If you want to change the values in the array, then you have to have access to the actual values, and not a copy. The way to do this in a ranged for is to use a reference as the loop variable:

```
for (int& k : squares)
{
k *= 2;
}
```

Now, on every iteration, the k variable is an alias to an actual member in the array, so whatever you do to the k variable is actually performed on the array member. In this example, every member of the squares array is multiplied by 2. You cannot use int* for the type of k because the compiler sees that the type of the items in the array is int and will use this as the loop variable in the ranged for. Since a reference is an alias for a variable, the compiler will allow a reference as the loop variable, and moreover, since the reference is an alias, you can use it to change the actual array member.

Ranged for becomes interesting for multidimensional arrays. For example, in the following, a two-dimensional array is declared and an attempt is made to use nested loops using auto variables:

```
int arr[2][3] { { 2, 3, 4 }, { 5, 6, 7} };
for (auto row : arr)
{
for (auto col : row) // will not compile
{
cout << col << " " << endl;
}
}
```

Since a two-dimensional array is an array of arrays (each row is a one-dimensional array), the intention is to obtain each row in the outer loop and then in the inner loop access each item in the row. There are several issues with this approach, but the immediate issue is that this code will not compile.

The compiler will complain about the inner loop, saying that it cannot find a begin or end function for the type int*. The reason is that ranged for uses iterator objects and for arrays it uses the C++ Standard Library functions, begin and end, to create these objects. The compiler will see from the arr array in the outer ranged for that each item is an int[3] array, and so in the outer for loop the loop variable will be a *copy* of each element, in this case an int[3] array. You cannot copy arrays like this, so the compiler will provide a pointer to the first element, an int*, and this is used in the inner for loop.

The compiler will attempt to obtain iterators for int*, but this is not possible because an int* contains no information about how many items it points to. There is a version of begin and end defined for int[3] (and all sizes of arrays) but not for int*.
A simple change makes this code compile. Simply turn the row variable into a reference:
```
for (auto& row : arr)
{
for (auto col : row)
{
cout << col << " " << endl;
}
}
```
The reference parameter indicates that an alias is used for the int[3] array and, of course, an alias is the same as the element. Using auto hides the ugliness of what is actually going on. The inner loop variable is, of course, an int since this is the type of the item in the array. The outer loop variable is in fact int (&)[3]. That is, it is a reference to an int[3] (the parentheses used to indicate that it references an int[3] and is not an array of int&).

Using pointers in practice

A common requirement is to have a collection that can be an arbitrary size and can grow and shrink at runtime. The C++ Standard Library provides various classes to allow you to do this, as will be described in Chapter 8, *Using the Standard Library Containers*. The following example illustrates some of the principles of how these standard collections are implemented. In general, you should use the C++ Standard Library classes rather than implementing your own. Further, the Standard Library classes *encapsulate* code together in a class and since we have not covered classes yet, the following code will use functions that potentially can be called incorrectly. So, you should regard this example as just that, example code. A linked list is a common data structure. These are typically used for queues where the order of items is important. For example, a first-in-first-out queue where tasks are performed in the order that they are inserted in the queue. In this example, each task is represented as a structure that contains the task description and a pointer to the next task to be performed.

If the pointer to the next task is nullptr then this means the current task is the last task in the list:
```
struct task
{
task* pNext;
string description;
};
```
Recall from the last chapter that you access members of a structure using the dot operator through an instance:
```
task item;
item.descrription = "do something";
```
In this case, the compiler will create a string object initialized with the string literal do something and assign it to the description member of the instance called item. You can also create a task on the free store using the new operator:
```
task* pTask = new task;
// use the object
delete pTask;
```
In this case, the members of the object have to be accessed through a pointer, and C++

provides the -> operator to give you this access:

```
task* pTask = new task;
pTask->descrription = "do something";
// use the object
delete pTask;
```

Here the description member is assigned to the string. Note that since task is a structure there are no access restrictions, something that is important with classes and described in Chapter 6, *Classes*.

Creating the project

Create a new folder under C:\Beginning_C++ called Chapter_04**.** Start Visual C++ and create a C++ source file and save it to the folder you just created, as tasks.cpp. Add a simple main function without parameters, and provide support for input and output using C++ streams:

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
}
```

Above the main function, add a definition for the structure that represents a task in the list:

```
using namespace std;
struct task {
task* pNext;
string description;
};
```

This has two members. The guts of the object is the description item. In our example, executing a task will involve printing the description item to the console. In an actual project, you'll most likely have many data items associated with the task, and you may even have member functions to execute the task, but we have not yet covered member functions; that's a topic for Chapter 6, *Classes*.

The plumbing of the linked list is the other member, pNext. Note that the task structure has not been completely defined at the point that the pNext member is declared. This is not a problem because pNext is a *pointer*. You cannot have a data member of an undefined, or a partially defined type, because the compiler will not know how much memory to allocate for it. You can have a pointer member to a partially defined type because a pointer member is the same size irrespective of what it points to.

If we know the first link in a list, then we can access the whole list and, in our example, this will be a global variable. When constructing the list, the construction functions need to know the end of the list so that they can attach a new link to the list. Again, for convenience, we will make this a global variable. Add the following pointers after the definition of the task structure:

```
task* pHead = nullptr;
task* pCurrent = nullptr;
int main()
{
}
```

As it stands, the code does nothing, but it is a good opportunity to compile the file to test that there are no typos:

```
cl /EHsc tasks.cpp
```

Adding a task object to the list

The next thing to do to provide the code is to add a new task to the task list. This needs to

create a new task object and initialize it appropriately and then add it to the list by altering the last link in the list to point to the new link.
Above the main function, add the following function:

```
void queue_task(const string& name)
{
...
}
```

The parameter is a const reference because we will not change the parameter and we do not want the overhead of a copy being made. The first thing this function must do is create a new link, so add the following lines:

```
void queue_task(const string& name)
{
task* pTask = new task;
pTask->description = name;
pTask->pNext = nullptr;
}
```

The first line creates a new link on the free store, and the following lines initialize it. This is not necessarily the best way of initializing such an object, and a better mechanism, a constructor, will be covered in Chapter 6, *Classes*. Notice that the pNext item is initialized to nullptr; this indicates that the link will be at the end of the list.
The final part of this function adds the link to the list, that is, it makes the link the last in the list. However, if the list is empty, it means that this link is also the *first* link in the list. The code must perform both actions. Add the following code to the end of the function:

```
if (nullptr == pHead)
{
pHead = pTask;
pCurrent = pTask;
}
else
{
pCurrent->pNext = pTask;
pCurrent = pTask;
}
```

The first line checks to see if the list is empty. If pHead is nullptr, it means that there are no other links and so the current link is the first link, and so both pHead and pCurrent are initialized to the new link pointer. If there are existing links in the list, the link has to be added to the last link, so in the else clause the first line makes the last link point to the new link and the second line initializes pCurrent with the new link pointer, making the new link the last link for any new insertions to the list.
The items are added to the list by calling this function in the main function. In this example, we will queue the tasks to wallpaper a room. This involves removing the old wallpaper, filling any holes in the wall, sizing the wall (painting it with diluted paste to make the wall sticky), and then hanging the pasted wallpaper to the wall. You have to do these tasks in this order, you cannot change the order, so these tasks are ideal for a linked list. In the main function add the following lines:

```
queue_task("remove old wallpaper");
queue_task("fill holes");
queue_task("size walls");
queue_task("hang new wallpaper");
```

After the last line, the list has been created. The pHead variable points to the first item in the list and you can access any other item in the list simply by following the pNext member from one link to the next.

You can compile the code, but there is no output. Worse, as the code stands, there is a memory leak. The program has no code to delete the memory occupied by the task objects created on the free store by the new operator.

Deleting the task list

Iterating through the list is simple, you follow the pNext pointer from one link to the next. Before doing this, let's first fix the memory leak introduced in the last section. Above the main function, add the following function:

```
bool remove_head()
{
if (nullptr == pHead) return false;
task* pTask = pHead;
pHead = pHead->pNext;
delete pTask;
return (pHead != nullptr);
}
```

This function will remove the link at the beginning of the list and make sure that the pHead pointer points to the next link, which will become the new beginning of the list. The function returns a bool value indicating if there are any more links in the list. If this function returns false then it means the entire list has been deleted.
The first line checks to see if this function has been called with an empty list. Once we are reassured that the list has at least one link, we create a temporary copy of this pointer. The reason is that the intention is to delete the first item and make pHead point to the next item, and to do that we have to do those steps in reverse: make pHead point to the next item and then delete the item that pHead previously pointed to.
To delete the entire list, you need to iterate through the links, and this can be carried out using a while loop. Below the remove_head function, add the following:

```
void destroy_list()
{
while (remove_head());
}
```

To delete the entire list, and address the memory leak, add the following line to the bottom of the main function

```
destroy_list();
}
```

You can now compile the code, and run it. However, you'll see no output because all the code does is create a list and then delete it.

Iterating the task list

The next step is to iterate the list from the first link following each pNext pointer until we get to the end of the list. For each link accessed, the task should be executed. Start by writing a function that performs the execution by printing out the description of the task and then returning a pointer to the next task. Just above the main function, add the following code:

```
task *execute_task(const task* pTask)
{
if (nullptr == pTask) return nullptr;
cout << "executing " << pTask->description << endl;
return pTask->pNext;
}
```

The parameter here is marked as const because we will not change the task object pointed to by the pointer. This indicates to the compiler that if the code does try to change the object there is an issue. The first line checks to make sure that the function is not called with a null pointer. If it was then the following line would dereference an invalid pointer and cause a memory access fault. The last line returns the pointer to the next link (which could be nullptr for the last link in the list), so that the function can be called in a loop. After this function, add the following to iterate the entire list:

```
void execute_all()
{
task* pTask = pHead;
while (pTask != nullptr)
{
pTask = execute_task(pTask);
}
}
```

This code starts at the beginning, pHead, and calls execute_task on each link in the list until the function returns a nullptr. Add a call to this function towards the end of the main function:

```
execute_all();
destroy_list();
}
```

You can now compile and run the code. The result will be:

```
executing remove old wallpaper
executing fill holes
executing size walls
executing hang new wallpaper
```

Inserting items

One of the advantages of linked lists is that you can insert items into the list by only allocating one new item and changing the appropriate pointers to point to it, and make it point to the next item in the list. Contrast this to allocating an array of task objects; if you want to insert a new item somewhere in the middle, you would have to allocate a new array big enough for the old items and the new one and then copy the old items to the new array, copying in the new item in the right position.

The problem with the wallpaper task list is that the room has some painted wood and, as any decorator knows, it is best to paint the woodwork before hanging the wallpaper, and usually before sizing the walls. We need to insert a new task between filling any holes and sizing the walls. Further, before you do any decorating, you should cover any furniture in the room before doing anything else, so you need to add a new task to the beginning.

The first step is to find the position where we want to put our new task to paint the woodwork. We will look for the task that we want to be before the task we are inserting. Before main add the following:

```
task *find_task(const string& name)
{
task* pTask = pHead;
while (nullptr != pTask)
{
if (name == pTask->description) return pTask;
pTask = pTask->pNext;
}
return nullptr;
}
```

This code searches the entire list for a link with the description that matches the parameter. This is carried out through a loop which uses the string comparison operator, and if the required link is found, a pointer to that link is returned. If the comparison fails, the loop initializes the loop variable to the address of the next link and if this address is nullptr it means that the required task is not in the list.

After the list is created in the main function, add the following code to search for the fill holes task:

```
queue_task("hang new wallpaper");
```
// oops, forgot to paint
woodworktask* pTask = find_task("fill holes");
if (nullptr != pTask) {
// insert new item after pTask
}
```
execute_all();
```
If the find_task function returns a valid pointer, then we can add an item at this point.
The function to do this will allow you to add a new item after any item in the list that you pass to it and, if you pass nullptr, it will add the new item to the beginning. It's called insert_after, but clearly, if you pass nullptr it also means *insert before the beginning*.
Add the following just above the main function:
```
void insert_after(task* pTask, const string& name)
{
task* pNewTask = new task;
pNewTask->description = name;
if (nullptr != pTask)
{
pNewTask->pNext = pTask->pNext;
pTask->pNext = pNewTask;
}
}
```
The second parameter is a const reference because we will not change the string, but the first parameter is not a const pointer because we will be changing the object that it points to. This function creates a new task object and initializes the description member to the new task name. It then checks to see if the task pointer passed to the function is null. If it is not, then the new item can be inserted *after* the specified link in the list. To do this, the new link pNext member is initialized to be the next item in the list, and the pNext member of the previous link is initialized to the address of the new link.

What about inserting an item at the beginning, when the function is passed nullptr as the item to insert after? Add the following else clause.
```
void insert_after(task* pTask, const string& name)
{
task* pNewTask = new task;
pNewTask->description = name;
if (nullptr != pTask)
{
pNewTask->pNext = pTask->pNext;
pTask->pNext = pNewTask;
}
else {
pNewTask->pNext = pHead;
pHead = pNewTask;
}
}
```
Here, we make the pNext member of the new item to point to the old beginning of the list and then change pHead to point to the new item.
Now, in the main function, you can add a call to insert a new task to paint the woodwork, and since we also forgot to indicate that it is best to decorate a room after covering all furniture with dustsheets, add a task to do that first in the list:
```
task* pTask = find_task("fill holes");
if (nullptr != pTask)
{
insert_after(pTask, "paint woodwork");
}
```

```
insert_after(nullptr, "cover furniture");
```
You can now compile the code. When you run the code, you should see the tasks performed in the required order:
```
executing cover furniture
executing remove old wallpaper
executing fill holes
executing paint woodwork
executing size walls
executing hang new wallpaper
```

Defining C++ functions

At the most basic level, a function has parameters, has code to manipulate the parameters, and returns a value. C++ gives you several ways to determine these three aspects. In the following section, we will cover those parts of a C++ function from the left to the right of the declaration. Functions can also be **templated**, but this will be left to a later section.

Declaring and defining functions

A function must be defined exactly once, but through overloading, you can have many functions with the same name that differ by their parameters. Code that uses a function has to have access to the name of the function, and so it needs to have access to either the function definition (for example, the function is defined earlier in the source file) or the declaration of the function (also called the function prototype). The compiler uses the prototype to type-check that the *calling code* is calling the function, using the right types.
Typically, libraries are implemented as separate compiled library files and prototypes of the library functions are provided in header files so that many source files can use the functions by including the headers. However, if you know the function name, parameters, and return type, you can type the prototype yourself in your file.

Whichever you do, you are simply providing the information for the compiler to type-check the expression that calls function. It is up to the linker to locate the function in the library and either copy the code into the executable or set up the infrastructure to use the function from a shared library. Including the header file for a library does not mean that you will be able to use the functions from that library because in standard C++, the header file does not have information about the library that contains a function.
Visual C++ provides a pragma called comment, which can be used with the lib option as a message to the linker to link with a specific library. So #pragma comment(lib, "mylib") in a header file will tell the linker to link with mylib.lib. In general, it is better to use project management tools, such as **nmake** or **MSBuild**, to ensure that the right libraries are linked in the project.
Most of the C Runtime Library is implemented this way: the function is compiled in a static library or a dynamic link library, and the function prototypes are provided in a header file. You provide the library in the linker command line, and typically you will include the header file for the library so that the function prototypes are available to the compiler. As long as the linker knows about the library, you can type the prototype in your code (and describe it as *external linkage* so the compiler knows the function is defined elsewhere). This can save you from including some large files into your source files, files that will mostly have prototypes of functions that you will not use.
However, much of the C++ Standard Library is implemented in header files, which means that these files can be quite large. You can save compile time by including these header files in a precompiled header, as explained in Chapter 1, *Starting with C++*.
So far in this book, we have used one source file so all the functions are defined in the same

file as where they are used, and we have defined the function before calling it, that is, the function is defined *above* the code that calls it. You do not have to define the function before it is used as long as the function prototype is defined before the function is called:

```
int mult(int, int);
int main()
{
cout << mult(6, 7) << endl;
return 0;
}
int mult(int lhs, int rhs)
{
return lhs * rhs;
}
```

The mult function is defined after the main function, but this code will compile because the prototype is given before the main function. This is called a **forward declaration**. The prototype does not have to have the parameter names. This is because the compiler only needs to know the types of the parameters, not their names. However, since parameter names should be self-documenting, it is usually a good idea to give the parameter names so that you can see the purpose of the function.

Specifying linkage

In the previous example, the function is defined in the same source file, so there is *internal linkage*. If the function is defined in another file, the prototype will have *external linkage* and so the prototype will have to be defined like this:

```
extern int mult(int, int); // defined in another file
```

The extern keyword is one of many specifiers that you can add to a function declaration, and in the previous chapters we have seen others. For example, the static specifier can be used on a prototype to indicate that the function has internal linkage and the name can only be used in the current source file. In the preceding example, it is appropriate to mark the function as static in the prototype.

```
static int mult(int, int); // defined in this file
```

You can also declare a function as extern "C", which affects how the name of the function is stored in the object file. This is important for libraries, and will be covered shortly.

Inlining

If a function calculates a value that can be calculated at compile time, you can mark it on the left of the declaration with constexpr to indicate that the compiler can optimize the code by computing the value at compile time. If the function value can be calculated at compile time, it means that the parameters in the function call must be known at compile time and so they must be literals. The function must also be a single line. If these restrictions are not met, then the compiler is free to ignore the specifier.

Related is the inline specifier. This can be placed on the left of a function declaration as a suggestion to the compiler that, when other code calls the function, rather than the compiler inserting a jump to the function in memory (and the creation of a stack frame), the compiler should put a copy of the actual code in the calling function. Again, the compiler is free to ignore this specifier.

Determining the return type

Functions may be written to run a routine and not return a value. If this is the case, you must specify that the function returns void. In most cases, a function will return a value, if only to indicate that the function has completed correctly. There is no requirement that the calling function obtains the return value or does anything with it. The calling function can

simply ignore the return value.
There are two ways to specify the return type. The first way is to give the type before the function name. This is the method used in most of the examples so far. The second way is called the **trailing return type** and requires that you place auto as the return type before the function name and use the -> syntax to give the actual return type after the parameter list:

```
inline auto mult(int lhs, int rhs) -> int
{
return lhs * rhs;
}
```

This function is so simple that it is a good candidate to be inlined. The return type on the left is given as auto, meaning that the actual return type is specified after the parameter list. The -> int means that the return type is int. This syntax has the same effect as using int on the left. This syntax is useful when a function is templated and the return type may not be noticeable.
In this trivial example, you can omit the return type entirely and just use auto on the left of the function name. This syntax means that the compiler will deduce the return type from the actual value returned. Clearly the compiler will only know what the return type is from the function body, so you cannot provide a prototype for such functions.
Finally, if a function does not return at all (for example, if it goes into a never-ending loop to poll some value) you can mark it with the C++11 attribute [[noreturn]]. The compiler can use this attribute to write more efficient code because it knows that it does not need to provide code to return a value.

Naming the function

In general, function names have the same rules for variables: they must begin with a letter or an underscore and cannot contain spaces or other punctuation characters. Following the general principle of self-documenting code, you should name the function according to what it does. There is one exception and these are the special functions used to provide overloads for operators (which are mostly punctuation symbols). These functions have a name in the form of operatorx, where x is the operator that you will use in your code. A later section will explain how to implement operators with global functions.
Operators are one example of overloading. You can overload any function, that is, use the same name but provide implementations with different parameter types or different numbers of parameters.

Function parameters

Functions may have no parameters, in which case the function is defined with a pair of empty parentheses. A function definition must give the type and name of the parameters between the parentheses. In many cases, functions will have a fixed number of parameters, but you can write functions with a variable number of parameters. You can also define functions with default values for some of the parameters, in effect, providing a function that overloads itself on the number of parameters passed to the function. Variable argument lists and default arguments will be covered later.

Specifying exceptions

Functions can also be marked to indicate whether they will throw an exception. More details about exceptions will be given in Chapter 10, *Diagnostics and Debugging*, but there are two syntaxes you need to be aware of.
Earlier versions of C++ allowed you to use the throw specifier on a function in three ways: firstly, you can provide a comma separated list of the types of the exceptions that may be thrown by code in the function; secondly, you can provide an ellipsis (...) which means that the function may throw any exception; and thirdly, you can provide an empty pair of

parentheses, which means the function will not throw exceptions. The syntax looks like this:

```
int calculate(int param) throw(overflow_error)
{
// do something which potentially may overflow
}
```

The throw specifier has been deprecated in C++11 largely because the ability to indicate the type of exception was not useful. However, the version of throw that indicates that no exception will be thrown was found to be useful because it enables a compiler to optimize code by providing no code infrastructure to handle exceptions. C++11 retains this behavior with the noexcept specifier:

```
// C++11 style:
int increment(int param) noexcept
{
// check the parameter and handle overflow appropriately
}
```

Function body

After the return type, function name, and parameters have been determined, you then need to define the body of the function. The code for a function must appear between a pair of braces ({}). If the function returns a value, then the function must have at least one line (the last line in the function) with the return statement. This must return the appropriate type or a type that can be implicitly converted to the return type of the function. As mentioned before, if the function is declared as returning auto, then the compiler will deduce the return type. In this case, all the return statements *must* return the same type.

Using function parameters

When a function is called, the compiler checks all the overloads of the function to find one that matches the parameters in the calling code. If there is no exact match then standard and user-defined type conversions are performed, so the values provided by the calling code may be a different type from the parameters.

By default, parameters are passed by value and a copy is made, which means that the parameters are treated as local variables in the function. The writer of the function can decide to pass a parameter by reference, either through a pointer or a C++ reference. **Passby-reference** means that the variable in the calling code can be altered by the function, but this can be controlled by making the parameters const, in which case the reason for passby-reference is to prevent a (potentially costly) copy being made. Built-in arrays are always passed as a pointer to the first item to the array. The compiler will create temporaries when needed. For example, when a parameter is a const reference and the calling code passes a literal, a temporary object is created, and is only available to code in the function:

```
void f(const float&);
```

```
f(1.0); // OK, temporary float created
double d = 2.0;
f(d); // OK, temporary float created
```

Passing Initializer lists

You can pass an initializer list as a parameter if that list can be converted to the type of the parameter. For example:

```
struct point { int x; int y; };
void set_point(point pt);
int main()
{
point p;
p.x = 1; p.y = 1;
set_point(p);
set_point({ 1, 1 });
return 0;
}
```

This code defines a structure that has two members. In the main function, a new instance of point is created on the stack and it is initialized by accessing the members directly. The instance is then passed to a function that has a point parameter. Since the parameter of set_point is pass-by-value, the compiler creates a copy of the structure on the stack of the function. The second call of set_point does the same: the compiler will create a temporary point object on the stack of the function and initialize it with the values in the initializer list.

Using default parameters

There are situations when you have one or more parameters that have values that are so frequently used that you want them to be treated as a default value for the parameter, while still having the option of allowing the caller to provide a different value if necessary. To do this, you provide the default value in the parameter list of the definition:

```
void log_message(const string& msg, bool clear_screen = false)
{
if (clear_screen) clear_the_screen();
cout << msg << endl;
}
```

In most cases, this function is expected to be used to print a single message, but occasionally the user may want to have the screen cleared first (say, for the first message, or after a predetermined count of lines). To accommodate this use of the function, the clear_screen parameter is given a default value of false, but the caller still has the option of passing a value:

```
log_message("first message", true);
log_message("second message");
bool user_decision = ask_user();
log_message("third message", user_decision);
```

Note that the default values occur in the function definition, not in a function prototype, so if the log_message function is declared in a header file the prototype should be:

```
extern void log_message(const string& msg, bool clear_screen);
```

The parameters that can have default values are the right-most parameters.
You can treat each parameter with a default value as representing a separate overload of the function, so conceptually the log_message function should be treated as two functions:

```
extern void log_message(const string& msg, bool clear_screen);
extern void log_message(const string& msg); // conceptually
```

If you define a log_message function that has just a const string& parameter, then the compiler will not know whether to call that function or the version where clear_screen is given a default value of false.

Variable number of parameters

A function with default parameter values can be regarded as having a variable number of user-provided parameters, where you know at compile time the maximum number of parameters and their values if the caller chooses not to provide values. C++ also allows you to write functions where there is less certainty about the number of parameters, and the values passed to the function.
There are three ways to have a variable number of parameters: initializer lists, C-style variable argument lists, and variadic templated functions. The latter of these three will be addressed later in the chapter once templated functions have been covered.

Initializer lists

So far in this book, initializer lists have been treated as a kind of C++11 construct, a bit like built-in arrays. In fact, when you use the initializer list syntax using braces, the compiler actually creates an instance of the templated initialize_list class. If an initializer list is used to initialize another type (for example, to initialize a vector), the compiler creates an initialize_list object with the values given between the braces, and the container object is initialized using the initialize_list iterators. This ability to create an initialize_list object from a braced initializer list can be used by to give a function a variable number of parameters, albeit all of the parameters must be of the same type:

```
#include <initializer_list>
int sum(initializer_list<int> values)
{
int sum = 0;
for (int i : values) sum += i;
return sum;
}
int main()
{
cout << sum({}) << endl; // 0
cout << sum({-6, -5, -4, -3, -2, -1}) << endl; // -21
cout << sum({10, 20, 30}) << endl; // 60
return 0;
}
```

The sum function has a single parameter of initializer_list<int>, which can only be initialized with a list of integers. The initializer_list class has very few functions because it only exists to give access to the values in the braced list. Significantly, it implements a size function that returns the number of items in the list, and begin and end functions that return a pointer to the first item in the list, and to the position after the last item. These two functions are needed to give iterator access to the list, and it enables you to use the object with the ranged-for syntax.

This is typical in the C++ Standard Library. If a container holds data in a contiguous block of memory, then pointer arithmetic can use the pointer to the first item and a pointer immediately after the last item to determine how many items are in the container. Incrementing the first pointer gives sequential access to every item, and pointer arithmetic allows random access. All containers implement a begin and end function to give access to the container *iterators*.

In this example, the main function calls this function three times, each time with a braced initializer list, and the function will return a sum of the items in the list.

Clearly this technique means that each item in the *variable* parameter list has to be the same type (or a type that can be converted to the specified type). You would have the same result if the parameter had been a vector; the difference is that an initializer_list parameter requires less initialization.

Argument lists

C++ inherits from C the idea of argument lists. To do this, you use the ellipses syntax (...) as the last parameter to indicate that the caller can provide zero or more parameters. The compiler will check how the function is called and will allocate space on the stack for these extra parameters. To access the extra parameters, your code must include the <cstdarg> header file, which has macros that you can use to extract the extra parameters off the stack. This is inherently type-unsafe because the compiler cannot check that the parameters that the function will get off the stack at runtime will be the same type as the parameters put on the stack by the calling code. For example, the following is an implementation of a function that will sum integers:

```
int sum(int first, ...)
{
int sum = 0;
va_list args;
va_start(args, first);
int i = first;
while (i != -1)
{
sum += i;
i = va_arg(args, int);
}
va_end(args);
return sum;
}
```

The definition of the function must have at least one parameter so that the macros work; in this case the parameter is called first. It is important that your code leaves the stack in a consistent state and this is carried out using a variable of the va_list type. This variable is initialized at the beginning of the function by calling the va_start macro and the stack is restored to its previous state at the end of the function by calling the va_end macro.

The code in this function simply iterates through the argument list, and maintains a sum, and the loop finishes when the parameter has a value of -1. There are no macros to give information about how many parameters there are on the stack, nor are there any macros to give an indication of the type of the parameter on the stack. Your code has to assume the type of the variable and provide the desired type in the va_arg macro. In this example, va_arg is called, assuming that every parameter on the stack is an int.

Once all parameters have been read off the stack, the code calls va_end before returning the sum. The function can be called like this:

```
cout << sum(-1) << endl; // 0
cout << sum(-6, -5, -4, -3, -2, -1) << endl; // -20 !!!
cout << sum(10, 20, 30, -1) << endl; // 60
```

Since -1 is used to indicate the end of the list, it means that to sum a zero number of parameters, you have to pass at least one parameter, that is -1. In addition, the second line shows that you have a problem if you are passing a list of negative numbers (in this case -1 cannot be a parameter). This problem could be addressed in this implementation by choosing another *marker value*.

Another implementation could eschew the use of a marker for the end of the list, and instead use the first, required, argument to give the count of the parameters that follow:

```
int sum(int count, ...)
{
int sum = 0;
va_list args;
va_start(args, count);
while(count--)
{
int i = va_arg(args, int);
sum += i;
}
va_end(args);
return sum;
}
```

This time, the first value is the *number of arguments* that follow, and so the routine will extract this exact number of integers off the stack and sum them. The code is called like this:

```
cout << sum(0) << endl; // 0
cout << sum(6, -6, -5, -4, -3, -2, -1) << endl; // -21
cout << sum(3, 10, 20, 30) << endl; // 60
```

There is no convention for how to handle the issue of determining how many parameters have been passed.

The routine assumes that every item on the stack is an int, but there is no information about this in the prototype of the function, so the compiler cannot do type checking on the parameters actually used to call the function. If the caller provides a parameter of a different type, the wrong number of bytes could be read off the stack, making the results of all the other calls to va_arg invalid. Consider this:

```
cout << sum(3, 10., 20, 30) << endl;
```

It is easy to press both, the comma and period keys, at the same time, and this has happened after typing the 10 parameter. The period means that the 10 is a double, and so the compiler puts a double value on the stack. When the function reads values off the stack with the va_arg macro, it will read the 8-byte double as two 4-byte int values and for code produced by Visual C++ this results in a total sum of 1076101140. This illustrates the type unsafe aspect of argument lists: you cannot get the compiler to do type checks of the parameters passed to the function.
If your function has different types passed to it then you have to implement some mechanism to determine what those parameters are. A good example of argument lists is the C printf function:

```
int printf(const char *format, ...);
```

The required parameter of this function is a format string, and importantly this has an ordered list of variable parameters and their types. The format string provides the information that is not available via the <cstdarg> macros: the number of variable parameters and the type of each one. The implementation of the printf function will iterate through the format string and when it comes across a format specifier for a parameter (a character sequence starting with %) it will read the expected type off the stack with va_arg. It should be clear that C-style argument lists are not as flexible as they appear on first sight; moreover, they can be quite dangerous.

Function features

Functions are modularized pieces of code defined as part of your application, or in a library. If a function is written by another vendor it is important that your code calls the function in the way intended by the vendor. This means understanding the calling convention used and how it affects the stack.

Call stack

When you call a function, the compiler will create a stack frame for the new function call and it will push items on to the stack. The data put on the stack depends on your compiler and whether the code is compiled for the debug or release build; however, in general there will be information about the parameters passed to the function, the return address (the address after the function call), and the automatic variables allocated in the function.
This means that, when you make a function call at runtime, there will be a memory overhead and performance overhead from creating the stack frame before the function runs, and a performance overhead in cleaning up, after the function completes. If a function is inlined, this overhead does not occur because the function call will use the current stack frame rather than a new one. Clearly, inlined functions should be small, both in terms of code and the memory used on the stack. The compiler can ignore the inline specifier and call the function with a separate stack frame.

Specifying calling conventions

When your code uses your own functions, you do not need to pay any attention to *calling conventions* because the compiler will make sure the appropriate convention is used. However, if you are writing library code that can be used by other C++ compilers, or even by other languages, then the calling convention becomes important. Since this book is not about interoperable code we won't go into much depth, but instead will look at two aspects: function naming and stack maintenance.

Using C linkage

When you give a C++ function a name, this is the name that you will use to call the function in your C++ code. However, under the covers, the C++ compiler will *decorate* the name with extra symbols for the return type and parameters so that overloaded functions all have different names. To C++ developers, this is also known as **name mangling**.

If you need to export a function through a shared library (in Windows, a **dynamic linked library**), you must use types and names that other languages can use. To do this, you can mark a function with extern "C". This means that the function has C linkage and the compiler will not use C++ name mangling. Clearly, you should use this only on functions that will be used by external code and you should not use it with functions that have return values and parameters that use C++ custom types.

However, if such a function does return a C++ type, the compiler will only issue a warning. The reason is that C is a flexible language and a C programmer will be able to work out how to turn the C++ type into something usable, but it is poor practice to abuse them like this!

The extern "C" linkage can also be used with global variables, and you can use it on a single item or (using braces) on many items.

Specifying how the stack Is maintained

Visual C++ supports six calling conventions that you can use on a function. The __clrcall specifier means that the function should be called as a .NET function and allows you to write code that has mixed native code and managed code. C++/CLR (Microsoft's language extensions to C++ to write .NET code) is beyond the scope of this book. The other five are used to indicate how parameters are passed to a function (on the stack or using CPU registers) and whose responsibility it is to maintain the stack. We will cover just three: __cdecl, __stdcall, and __thiscall.

You will rarely explicitly use __thiscall; it is the calling convention used for functions defined as members of custom types, and indicates that the function has a hidden parameter that is a pointer to the object that can be accessed through the this keyword in the function. More details will be given in the next chapter, but it is important to realize that such member functions have a different calling convention, especially when you need to initialize function pointers.

By default, C++ global functions will use the __cdecl calling convention. The stack is maintained by the calling code, so in the calling code each call to a __cdecl function is followed by code to clean up the stack. This makes each function call a little larger, but it is needed for variable argument lists to be used. The __stdcall calling convention is used by most of the Windows SDK functions and it indicates that the called function cleans up the stack so there is no need for such code to be generated in the calling code. Clearly, it is important that the compiler knows that a function uses __stdcall because, otherwise, it will generate code to clean up a stack frame that has already been cleaned up by the function. You will usually see Windows functions marked with WINAPI, which is a typedef for __stdcall.

Using recursion

In most cases the memory overhead of a call stack is unimportant. However, when you use recursion it is possible to build up a long chain of stack frames. As the name suggests, recursion is when a function calls itself. A simple example is a function that calculates a factorial:

```
int factorial(int n)
{
if (n > 1) return n ∗ factorial(n − 1);
return 1;
}
```

If you call this for 4, the following calls are made:

```
factorial(4) returns 4 * factorial(3)
factorial(3) returns 3 * factorial(2)
factorial(2) returns 2 * factorial(1)
factorial(1) returns 1
```

The important point is that in the recursive function there must be at least one way to leave the function without recursion. In this case, it will be when factorial is called with a parameter of 1. In practice, a function like this should be marked as inline to avoid creating any stack frames at all.

Overloading functions

You can have several functions with the same name, but where the parameter list is different (the number of parameters and/or the type of the parameters). This is *overloading* the function name. When such a function is called, the compiler will attempt to find the function that best fits the parameters provided. If there is not a suitable function, the compiler will attempt to convert the parameters to see if a function with those types exists. The compiler will start with trivial conversions (for example, an array name to a pointer, a type to a const type), and if this fails the compiler will try to promote the type (for example, bool to int). If that fails, the compiler will try standard conversions (for example, a reference to a type). If such conversions results in more than one possible candidate, then the compiler will issue an error that the function call is ambiguous.

Functions and scope

The compiler will also take the scope of the function into account when looking for a suitable function. You cannot define a function within a function, but you can provide a function prototype within the scope of a function and the compiler will attempt (if necessary through conversions) to call a function with such a prototype first. Consider this code:

```
void f(int i) { /*does something*/ }
void f(double d) { /*does something*/ }
int main()
{
void f(double d);
f(1);
return 0;
}
```

In this code, the function f is overloaded with one version that takes an int and the other with a double. Normally, if you call f(1) then the compiler will call the first version of the function. However, in main there is a prototype for the version that takes a double, and an int can be converted to a double with no loss of information. The prototype is in the same scope as the function call, so in this code the compiler will call the version that takes a double. This technique essentially *hides* the version with an int parameter.

Deleted functions

There is a more formal way to hide functions than using the scope. C++ will attempt to explicitly convert built-in types. For example:

```
void f(int i);
```

You can call this with an int, or anything that can be converted to an int:

```
f(1);
f('c');
f(1.0); // warning of conversion
```

In the second case, a char is an integer, so it is promoted to an int and the function is called. In the third case, the compiler will issue a warning that the conversion can cause a loss of data, but it is a warning and so the code will compile. If you want to prevent this implicit conversion you can *delete* the functions that you do not want callers to use. To do this, provide a prototype and use the syntax = delete:

```
void f(double) = delete;
void g()
{
f(1); // compiles
f(1.0); // C2280: attempting to reference a deleted function
}
```

Now, when the code attempts to call the function with a char or a double (or float, which will be implicitly converted to a double), the compiler will issue an error.

Passing by value and passing by reference

By default, the compiler will pass parameters by value, that is, a copy is made. If you pass a custom type, then its *copy constructor* is called to create a new object. If you pass a pointer to an object of a built-in type or custom type, then the *pointer* will be passed by value, that is, a new pointer is created on the function stack for the parameter and it is initialized with the memory address passed to the function. This means that, in the function, you can change the pointer to point to other memory (this is useful if you want to use pointer arithmetic on that pointer). The data that the pointer points to will be passed by a reference, that is, the data remains where it is, outside of the function, but the function can use the pointer to change the data. Similarly, if you use a reference on a parameter then it means that the object is passed by the reference. Clearly, if you use const on a pointer or reference parameter then this will affect whether the function can change the data pointed to or referenced.

In some cases, you may want to return several values from a function, and you may choose to use the return value of the function to indicate if the function executed correctly. One way to do this is to make one of the parameters an *out* parameter, that is, it is either a pointer or a reference to an object or container that the function will alter:

```
// don't allow any more than 100 items
bool get_items(int count, vector<int>& values)
{
if (count > 100) return false;
for (int i = 0; i < count; ++i)
{
```

```
values.push_back(i);
}
return true;
}
```

To call this function, you must create a vector object and pass it to the function:

```
vector<int> items {};
get_items(10, items);
for(int i : items) cout << i << ' ';
cout << endl
```

Because the values parameter is a reference it means that when get_values calls push_back to insert a value in the values container it is actually inserting that value into the items container.

If an out parameter is passed via a pointer it is important to look at the pointer declaration. A single * means that the variable is a pointer, two means that it is a pointer to a pointer. The following function returns an int through an out parameter:

```
bool get_datum(/*out*/ int *pi);
```

The code is called like this:

```
int value = 0;
if (get_datum(&value)) { cout << "value is " << value << endl; }
else { cout << "cannot get the value" << endl;}
```

This pattern of returning a value indicating success is frequently used, particularly with code that accesses data across process or machine boundaries. The function return value can be used to give detailed information about why the call failed (no network access?, invalid security credentials?, and so on), and indicates that the data in the out parameters should be discarded.

If the out parameter has a double * then it means the return value is itself a pointer, either to a single value or to an array:

```
bool get_data(/*in/out*/ int *psize, /*out*/ int **pi);
```

In this case, you pass in the size of the buffer you want using the first parameter and on return you receive the actual size of the buffer via this parameter (it is in/out) and a pointer to the buffer in the second parameter:

```
int size = 10;
int *buffer = nullptr;
if (get_data(&size, &buffer))
{
```

```
for (int i = 0; i < size; ++i)
{
cout << buffer[i] << endl;
}
delete [] buffer;
}
```

Any function that returns a memory buffer must document who has the responsibility of deallocating the memory. In most cases, it is usually the caller, as assumed in this example code.

Designing functions

Often functions will act upon global data, or data passed in by the caller. It is important that when the function completes, it leaves this data in a consistent state. Equally so, it is important that, the function can make assumptions about the data before it accesses it.

Pre- and post-conditions

A function will typically alter some data: values passed into the function, data returned by the function, or some global data. It is important when designing a function that you determine what data will be accessed and changed and that these rules are documented.

A function will have pre-conditions, assumptions about the data that it will use. For example, if a function is passed a filename, with the intention that the function will extract some data from the file, whose responsibility is it to check that the file exists? You can make it the responsibility of the function, and so the first few lines will check that the name is a valid path to a file and call operating system functions to check that the file exists. However, if you have several functions that will perform actions on the file, you will be replicating this checking code in each function and it may be better to put that responsibility on the calling code. Clearly such actions can be expensive, so it is important to avoid both the calling code and the function to perform the checks.

Chapter 10, *Diagnostics and Debugging*, will describe how to add debugging code, called

asserts, that you can place in your functions to check the values of the parameters to make sure that the calling code is following the pre-condition rules you have set. Asserts are defined using conditional compilation and so will only appear in **debug builds** (that is, C++ code compiled with debugging information). **Release builds** (completed code that will be delivered to the end user) will conditionally compile asserts away; this makes the code faster, and if your testing is thorough enough, you can be assured that pre-conditions are met.

You should also document the post-conditions of your function. That is, assumptions about the data returned by the function (through the function return value, out parameters, or parameters passed by a reference). Post-conditions are the assumptions that the calling code will make. For example, you may return a signed integer where the function is meant to return a positive value, but a negative value is used to indicate an error. Often functions that return pointers will return nullptr if the function fails. In both cases, the calling code knows that it needs to check the return value and only use it if it is either positive or not nullptr.

Using invariants

You should be careful to document how a function uses data external to the function. If the intention of the function is to change external data, you should document what the function will do. If you don't explicitly document what the function does to external data, then you must ensure that when the function finishes such data is left untouched. The reason is that the calling code will only assume what you have said in the documentation and the sideeffects of changing global data may cause problems. Sometimes it is necessary to store the state of global data and return the item back to that state before the function returns.
We have already seen an example of this in Chapter 3, *Exploring C++ Types*, with the cout object. The cout object is global to your application, and it can be changed through manipulators to make it interpret numeric values in certain ways. If you change it in a function (say, by inserting the hex manipulator), then this change will remain when the cout object is used outside the function.
Chapter 3, *Exploring C++ Types*, shows how to address such an issue. In that chapter, you created a function called read16 that reads 16 bytes from a file and prints the values out to the console both in hexadecimal form and interpreted as an ASCII character:

```
int read16(ifstream& stm)
{
if (stm.eof()) return -1;
int flags = cout.flags();
cout << hex;
string line;
// code that changes the line variable
cout.setf(flags);
return line.length();
}
```

This code stores the state of the cout object in a temporary variable, flags. The read16 function can change the cout object in any way necessary, but because we have the stored state it means that the object can be restored to its original state before returning.

Function pointers

When an application is run, the functions it will call will exist in memory somewhere. This means that you can get the address of a function. C++ allows you to use the function call operator (a pair of parentheses enclosing the parameters ()) to call a function through a function pointer.

Remember the parentheses!

First, a simple example of how function pointers can cause difficult to notice bugs in your code. A global function called get_status performs various validation actions to determine if the state of the system is valid. The function returns a value of zero to mean that the system state is valid and values over zero are error codes:

```
// values over zero are error codes
int get_status()
{
int status = 0;
// code that checks the state of data is valid
return status;
}
```

The code could be called like this:

```
if (get_status > 0)
{
cout << "system state is invalid" << endl;
}
```

This is an error because the developer has missed off the (), so the compiler does not treat this as a function call. Instead, it treats this as a test of the memory address of the function, and since the function will never be located at a memory address of zero, the comparison will always be true and the message will be printed even if the system state is valid.

Declaring function pointers

The last section highlights how easy it is to get the address of a function: you just use the name of the function without the parentheses:

```
void *pv = get_status;
```

The pointer pv is only of mild interest; you now know where in memory the function is stored, but to print this address you still need to cast it to an integer. To make the pointer useful, you need to be able to declare a pointer through which the function can be called. To look at how to do this, let's go back to the function prototype:

```
int get_status()
```

The function pointer must be able to call the function passing no parameters and expecting a return value of an integer. The function pointer is declared like this:

```
int (*fn)() = get_status;
```

The * indicates that the variable fn is a pointer; however, this binds to the left, so without the parentheses surrounding *fn the compiler would interpret this to mean that the declaration is for an int* pointer. The rest of the declaration indicates how this function pointer is called: taking no parameters and returning an int.

Calling through a function pointer is simple: you give the name of the pointer where you would normally give the name of the function:

```
int error_value = fn();
```

Note again how important the parentheses are; they indicates that the function at the address held in the function pointer, fn, is called.

Function pointers can make code look rather cluttered, especially when you use them to point to templated functions, so often code will define an alias:

```
using pf1 = int(*)();
typedef int(*pf2)();
```

These two lines declare aliases for the type of the function pointer needed to call the get_status function. Both are valid, but the using version is more readable since it is clear that pf1 is the alias being defined. To see why, consider this alias:

```
typedef bool(*MyPtr)(MyType*, MyType*);
```

The type alias is called MyPtr and it is to a function that returns a bool and takes two MyType pointers. This is much clearer with using:

```
using MyPtr = bool(*)(MyType*, MyType*);
```

The tell-tale sign here is the (*), which indicates that the type is a function pointer because you are using the parenthesis to break the associatively of the *. You can then read outwards to see the prototype of the function: to the left to see the return type, and to the right to get the parameter list.

Once you have declared an alias, you can create a pointer to a function and call it:

```
using two_ints = void (*)(int, int);
void do_something(int l, int r){/* some code */}
void caller()
{
two_ints fn = do_something;
fn(42, 99);
}
```

Notice that, because the two_ints alias is declared as a pointer, you do not use a * when declaring a variable of this type.

Using function pointers

A function pointer is merely a pointer. This means that you can use it as a variable; you can return it from a function, or pass it as a parameter. For example, you may have some code that performs some lengthy routine and you want to provide some feedback during the routine. To make this flexible, you could define your function to take a **callback pointer** and periodically in the routine call the function to indicate progress:

```
using callback = void(*)(const string&);
void big_routine(int loop_count, const callback progress)
{
for (int i = 0; i < loop_count; ++i)
{
```

```
if (i % 100 == 0)
{
string msg("loop ");
msg += to_string(i);
progress(msg);
}
// routine
}
}
```

Here big_routine has a function pointer parameter called progress. The function has a loop that will be called many times and every one hundredth loop it calls the callback function, passing a string that gives information about the progress.

Note that the string class defines a += operator that can be used to append a string to the end of the string in the variable and the <string> header file defines a function called to_string that is overloaded for each of the built-in types to return a string formatted with the value of the function parameter.

This function declares the function pointer as const merely so that the compiler knows that the function pointer should not be changed to a pointer to another function in this function.

The code can be called like this:

```
void monitor(const string& msg)
{
cout << msg << endl;
}
int main()
{
big_routine(1000, monitor);
return 0;
}
```

The monitor function has the same prototype as described by the callback function pointer (if, for example, the function parameter was string& and not const string&, then the code will not compile). The big_routine function is then called, passing a pointer to the monitor function as the second parameter.

If you pass callback functions to library code, you must pay attention to the calling convention of the function pointer. For example, if you pass a function pointer to a Windows function, such as EnumWindows, it must point to a function declared with the __stdcall calling convention.

The C++ standards uses another technique to call functions defined at runtime, which is, functors. It will be covered shortly.

Templated functions

When you write library code, you often have to write several functions that differ only between the types that are passed to the function; the routine action is the same, it's just the types that have changed. C++ provides *templates* to allow you to write more generic code; you write the routine using a *generic type* and at compile time the compiler will generate a function with the appropriate types. The templated function is marked as such using the template keyword and a list of parameters in angle brackets (<>) that give placeholders for the types that will be used. It is important to understand that these template parameters are types and refer to the types of the parameters (and return a value of the function) that will be replaced with the actual types used by calling the functions. They are not parameters of the function, and you do not (normally) provide them when you call the function.

It is best to explain template functions with an example. A simple maximum function can be written like this:

```
int maximum(int lhs, int rhs)
{
return (lhs > rhs) ? lhs : rhs;
}
```

You can call this with other integer types, and smaller types (short, char, bool, and so on) will be promoted to an int, and values of larger types (long long) will be truncated. Similarly, variables of unsigned types will be converted to the signed int which could cause problems. Consider this call of the function:

```
unsigned int s1 = 0xffffffff, s2 = 0x7fffffff;
unsigned int result = maximum(s1, s2);
```

What is the value of the result variable: s1 or s2? It is s2. The reason is that both values are converted to signed int and when converted to a signed type s1 will be a value of -1 and s2 will be a value of 2147483647.

To handle unsigned types, you need to *overload* the function and write a version for signed and unsigned integers:

```
int maximum(int lhs, int rhs)
{
return (lhs > rhs) ? lhs : rhs;
}
unsigned maximum(unsigned lhs, unsigned rhs)
{
return (lhs > rhs) ? lhs : rhs;
}
```

The routine is the same, but the types have changed. There is another issue--what if the caller mixes types? Does the following expression make any sense:

```
int i = maximum(true, 100.99);
```

This code will compile because a bool and a double can be converted to an int and the first overload will be called. Since such a call is nonsense, it would be much better if the compiler caught this error.

Defining templates

Returning back to the two versions of the maximum function, the routine is the same for both; all that has changed is the type. If you had a generic type, let's call it T, where T could be any type that implements an operator>, the routine could be described by this pseudocode:

```
T maximum(T lhs, T rhs)
{
return (lhs > rhs) ? lhs : rhs;
}
```

This will not compile because we have not defined the type T. Templates allow you to tell the compiler that the code uses a type and will be determined from the parameter passed to the function. The following code will compile:

```
template<typename T>
T maximum(T lhs, T rhs)
{
return (lhs > rhs) ? lhs : rhs;
}
```

The template declaration specifies the type that will be used using the typename identifier. The type T is a placeholder; you can use any name you like as long as it is not a name used elsewhere at the same scope, and of course, it must be used in the parameter list of the function. You can use class instead of typename, but the meaning is the same.

You can call this function, passing values of any type, and the compiler will create the code for that type, calling the operator> for that type.

It is important to realize that, the first time the compiler comes across a templated function, it will create a version of the function for the specified type. If you call the templated function for several different types, the compiler will create, or instantiate, a *specialized* function for each of these types.

The definition of this template indicates that only one type will be used, so you can only call it with two parameters of the same type:

```
int i = maximum(1, 100);
double d = maximum(1.0, 100.0);
bool b = maximum(true, false);
```

All of these will compile and the first two will give the expected results. The last line will assign b to a value of true because bool is an integer and true has a value of 1+ and false has a value of 0. This may not be what you would want, so we will return to this issue later. Note that, since the template says that both parameters must be the same type, the following will not compile:

```
int i = maximum(true, 100.99);
```

The reason is that the template parameter list only gives a single type. If you want to define a function with parameters of different types, then you will have to provide extra parameters to the template:

```
template<typename T, typename U>
T maximum(T lhs, U rhs)
{
return (lhs > rhs) ? lhs : rhs;
}
```

This is done to illustrate how templates work; it really does not make sense to define a maximum function that takes two different types.

This version is written for two different types, the template declaration mentions two types, and these are used for the two parameters. But notice that the function returns T, the type of the first parameter. The function can be called like this:
```
cout << maximum(false, 100.99) << endl; // 1
cout << maximum(100.99, false) << endl; // 100.99
```
The output from the first is 1 (or if you use the bool alpha manipulator, true) and the result of the second line is 100.99. The reason is not immediately obvious. In both cases, the comparison will return 100.99 from the function, but because the type of the return value is T, the returned value type will be the type of the first parameter. In the first case, 100.99 is first converted to a bool, and since 100.99 is not zero, the value returned is true (or 1). In the second case, the first parameter is a double, so the function returns a double and this means that 100.99 is returned. If the template version of maximum is changed to return U (the type of the second parameter) then the values returned by the preceding code are reversed: the first line returns 100.99 and the second returns 1.
Note that when you *call* the template function, you do not have to give the types of the template parameters because the compiler will deduce them. It is important to point out that this applies only to the parameters. The return type is not determined by the type of the variable the caller assigns to the function value because the function can be called without using the return value.
Although the compiler will deduce the template parameters from how you call the function, you can explicitly provide the types in the called function to call a specific version of the function and (if necessary) get the compiler to perform implicit conversions:
```
// call template<typename T> maximum(T,T);
int i = maximum<int>(false, 100.99);
```
This code will call the version of maximum that has two int parameters and returns an int, so the return value is 100, that is, 100.99 converted to an int.

Using template parameter values

The templates defined so far have had types as the parameters of the template, but you can also provide integer values. The following is a rather contrived example to illustrate the point:
```
template<int size, typename T>
T* init(T t)
{
T* arr = new T[size];
```

```
for (int i = 0; i < size; ++i) arr[i] = t;
return arr;
}
```
There are two template parameters. The second parameter provides the name of a type where T is a placeholder used for the type of the parameter of the function. The first parameter looks like a function parameter because it is used in a similar way. The parameter size can be used in the function as a local (read-only) variable. The function parameter is T and so the compiler can deduce the second template parameter from the function call, but it cannot deduce the first parameter, so you *must* provide a value in the call. Here is an example of calling this template function for an int for T and a value of 10 for size:
```
int *i10 = init<10>(42);
for (int i = 0; i < 10; ++i) cout << i10[i] << ' ';
cout << endl;
delete [] i10;
```
The first line calls the function with 10 as the template parameter and 42 as the function parameter. Since 42 is an int, the init function will create an int array with ten members

and each one is initialized to a value of 42. The compiler deduced int as the second parameter, but this code could have called the function with init<10,int>(42) to explicitly indicate that you require an int array.
The non-type parameters must be constant at compile time: the value can be integral (including an enumeration), but not a floating point. You can use arrays of integers, but these will be available through the template parameter as a pointer.
Although in most cases the compiler cannot deduce the value parameter, it can if the value is defined as the size of an array. This can be used to make it appear that a function can determine the size of a built-in array, but of course, it can't because the compiler will create a version of the function for each size needed. For example:

```
template<typename T, int N> void print_array(T (&arr)[N])
{
for (int i = 0; i < N; ++i)
{
cout << arr[i] << endl;
}
}
```

Here, there are two template parameters: one is the type of the array, and the other is the size of the array. The parameter of the function looks a little odd, but it is just a built-in array being passed by a reference. If the parentheses are not used then the parameter is T& arr[N], that is, an N-sized built-in array of references to objects of type T, which is not what we want. We want an N-sized built-in array objects of type T. This function is called like this:

```
int squares[] = { 1, 4, 9, 16, 25 };
print_array(squares);
```

The interesting thing about the preceding code is that the compiler sees that there are five items in the initializer list. The built-in array has five items, thus calls the function like this:

```
print_array<int,5>(squares);
```

As mentioned, the compiler will instantiate this function for every combination of T and N that your code calls. If the template function has a large amount of code, then this may be an issue. One way around this is to use a helper function:

```
template<typename T> void print_array(T* arr, int size)
{
for (int i = 0; i < size; ++i)
{
cout << arr[i] << endl;
}
}
template<typename T, int N> inline void print_array(T (&arr)[N])
{
print_array(arr, N);
}
```

This does two things. First, there is a version of print_array that takes a pointer and the number of items that the pointer points to. This means that the size parameter is determined at runtime, so versions of this function are only instantiated at compile time for the types of the arrays used, not for both type and array size. The second thing to note is that the function that is templated with the size of the array is declared as inline and it calls the first version of the function. Although there will be a version of this for each combination of type and array size, the instantiation will be inline rather than a complete function.

Specialized templates

In some cases, you may have a routine that works for most types (and a candidate for a templated function), but you may identify that some types need a different routine. To handle this, you can write a specialized template function, that is, a function that will be used for a specific type and the compiler will use this code when a caller uses types that fit this specialization. As an example, here is a fairly pointless function; it returns the size of a type:

```
template <typename T> int number_of_bytes(T t)
{
return sizeof(T);
}
```

This works for most built-in types, but if you call it with a pointer, you will get the size of the pointer, not what the pointer points to. So, `number_of_bytes("x")` will return 4 (on a 32-bit system) rather than 2 for the size of the `char` array. You may decide that you want a specialization for `char*` pointers that uses the C function, `strlen`, to count the number of characters in the string until the `NUL` character. To do this, you need a similar prototype to the templated function, replacing the template parameter with the actual type, and since the template parameter is not needed you miss this out. Since this function is for a specific type, you need to add the specialized type to the function name:

```
template<> int number_of_bytes<const char *>(const char *str)
{
return strlen(str) + 1;
}
```

Now when you call `number_of_bytes("x")` the specialization will be called and it will return a value of 2.

Earlier, we defined a templated function to return a maximum of two parameters of the same type:

```
template<typename T>
T maximum(T lhs, T rhs)
{
return (lhs > rhs) ? lhs : rhs;
}
```

Using specialization, you can write versions for types that are not compared using the `>` operator. Since it makes no sense to find the maximum of two Booleans, you can delete the specialization for `bool`:

```
template<> bool maximum<bool>(bool lhs, bool rhs) = delete;
```

This now means that, if the code calls `maximum` with `bool` parameters, the compiler will generate an error.

Variadic templates

A variadic template is when there is a variable number of template parameters. The syntax is similar to variable arguments to a function; you use ellipses, but you use them on the left of the argument in the parameter list, which declares it a *parameter pack*:

```
template<typename T, typename... Arguments>
void func(T t, Arguments... args);
```

The `Arguments` template parameter is zero or more types, which are the types of the corresponding number of arguments, `args`, of the function. In this example, the function has at least one parameter, of type `T`, but you can have any number of fixed parameters, including none at all.

Within the function, you need to unpack the parameter pack to get access to the parameters passed by the caller. You can determine how many items there are in the parameter pack using the special operator, `sizeof...` (note the ellipses are part of the name); unlike the `sizeof` operator, this is the item count and not the size in bytes. To unpack the parameter pack, you need to use the ellipses on the right of the name of the parameter pack (for example, `args...`). The compiler will expand the parameter pack at this point, replacing

the symbol with the contents of the parameter pack.
However, you will not know at design time how many parameters there are or what types they are, so there are some strategies to address this. The first uses recursion:

```
template<typename T> void print(T t)
{
cout << t << endl;
}
template<typename T, typename... Arguments>
void print(T first, Arguments ... next)
{
print(first);
print(next...);
}
```

The variadic templated `print` function can be called with one or more parameters of any type that can be handled by the `ostream` class:

```
print(1, 2.0, "hello", bool);
```

When this is called, the parameter list is split into two: the first parameter (`1`) in the first parameter, `first`, and the other three are put in the parameter pack, `next`. The function body then calls the first version of `print` which, prints the `first` parameter to the console. The next line in the variadic function then expands the parameter pack in a call to `print`, that is, this calls itself recursively. In this call, the `first` parameter will be `2.0`, and the rest will be put in the parameter pack. This continues until the parameter pack has been expanded so much that there are no more parameters.
Another way to unpack the parameter pack is to use an initializer list. In this case, the compiler will create an array with each parameter:

```
template<typename... Arguments>
void print(Arguments ... args)
{
int arr [sizeof...(args)] = { args... };
for (auto i : arr) cout << i << endl;
}
```

The array, `arr`, is created with the size of the parameter pack and the unpack syntax used with the initializer braces will fill the array with the parameters. Although this will work with any number of parameters, all the parameters have to be the same type of the array, `arr`.
One trick is to use the comma operator:

```
template<typename... Arguments>
void print(Arguments ... args)
{
int dummy[sizeof...(args)] = { (print(args), 0)... };
}
```

This creates a dummy array called `dummy`. This array is not used, other than in the expansion of the parameter pack. The array is created in the size of the `args` parameter pack and the ellipsis expands the parameter pack using the *expression* between the parentheses. The expression uses the comma operator, which will return the right side of the comma. Since this is an integer, it means that each entry of `dummy` has a value of zero. The interesting part is the left side of the comma operator. Here the version of `print` with a single templated parameter is called with each item in the `args` parameter pack.

Overloaded operators

Earlier we said that function names should not contain punctuation. That is not strictly true because, if you are writing an operator, you *only* use punctuation in the function name. An

operator is used in an expression acting on one or more operands. A unary operator has one operand, a binary operator has two operands, and an operator returns the result of the operation. Clearly this describes a function: a return type, a name, and one or more parameters.
C++ provides the keyword operator to indicate that the function is not used with the function call syntax, but instead is called using the syntax associated with the operator (usually, a unary operator the first parameter is on the right of the operator, and for a binary operator the first parameter is on the left and the second is on the right, but there are exceptions to this).
In general, you will provide the operators as part of a custom type (so the operators act upon variables of that type) but in some cases, you can declare operators at a global scope. Both are valid. If you are writing a custom type (classes, as explained in the next chapter), then it makes sense to encapsulate the code for an operator as part of the custom type. In this section, we will concentrate on the other way to define an operator: as a global function.
You can provide your own versions of the following unary operators:

```
! & + - * ++ -- ~
```

You can also provide your own versions of the following binary operators:

```
!= == < <= > >= && ||
% %= + += - -= * *= / /= & &= | |= ^ ^= << <<= = >> =>>
-> ->* ,
```

You can also write versions of the function call operator (), array subscript [], conversion operators, the cast operator (), and new and delete. You cannot redefine the ., .*, ::, ?:, # or ## operators, nor the "named" operators, sizeof, alignof or typeid.

When defining the operator, you write a function where the function name is operator*x* and *x* is the operator symbol (note that there is no space). For example, if you define a struct that has two members defining a Cartesian point, you may want to compare two points for equality. The struct can be defined like this:

```
struct point
{
int x;
int y;
};
```

Comparing two point objects is easy. They are the same if x and y of one object are equal to the corresponding values in the other object. If you define the == operator, then you should also define the != operator using the same logic because != should give the exact opposite result of the == operator. This is how these operators can be defined:

```
bool operator==(const point& lhs, const point& rhs)
{
return (lhs.x == rhs.x) && (lhs.y == rhs.y);
}
bool operator!=(const point& lhs, const point& rhs)
{
return !(lhs == rhs);
}
```

The two parameters are the two operands of the operator. The first one is the operand on the left-hand side and the second parameter is the operand on the right-hand side of the operator. These are passed as references so that a copy is not made, and they are marked as const because the operator will not alter the objects. Once defined, you can use the point type like this:

```
point p1{ 1,1 };
point p2{ 1,1 };
cout << boolalpha;
cout << (p1 == p2) << endl; // true
cout << (p1 != p2) << endl; // false
```

You could have defined a pair of functions called equals and not_equals and use these instead:

```
cout << equals(p1,p2) << endl; // true
cout << not_equals(p1,p2) << endl; // false
```

However, defining operators makes the code more readable because you use the type like the built-in types. Operator overloading is often referred to as *syntactic sugar*, syntax that makes the code easier to read--but this trivializes an important technique. For example, smart pointers are a technique that involves class **destructors** to manage resource lifetime, and are only useful because you can call the objects of such classes as if they are pointers. You can do this because the smart pointer class implements the -> and * operators. Another example is **functors**, or function objects, where the class implements the () operator so that objects can be accessed as if they are functions.

When you write a custom type, you should ask yourself if overloading an operator for your type makes sense. If the type is a numeric type, for example, a complex number or a matrix - then it makes sense to implement arithmetic operators, but would it make sense to implement the logical operators since the type does not have a logical aspect? There is a temptation to redefine the *meaning* of operators to cover your specific operation, but this will make your code less readable.

In general, a unary operator is implemented as a global function that takes a single parameter. The postfix increment and decrement operators are an exception to allow for a different implementation from prefix operators. Prefix operators will have a reference to the object as a parameter (which the operator will increment or decrement) and return a reference to this changed object. The postfix operator, however, has to return the value of the object before the increment or decrement. Thus, the operator function has two parameters: a reference to an object that will be changed and an integer (which will always be a value of 1); it will return a copy of the original object.

A binary operator will have two parameters and return an object or a reference to an object. For example, for the struct we defined previously, we could define an insertion operator for ostream objects:

```
struct point
{
int x;
int y;
};
ostream& operator<<(ostream& os, const point& pt)
{
os << "(" << pt.x << "," << pt.y << ")";
return os;
}
```

This means that you can now insert a point object to the cout object to print it on the console:

```
point pt{1, 1};
cout << "point object is " << pt << endl;
```

Function objects

A function object, or **functor**, is a custom type that implements the function call operator: (operator()). This means that a function operator can be called in a way that looks like it is a function. Since we haven't covered classes yet, in this section we will just explore the function objects types that are provided by the Standard Library and how to use them.

The <functional> header file contains various types that can be used as function objects. The following table lists these:

Purpose Types

Arithmetic divides, minus, modulus, multiplies, negate, plus
Bitwise bit_and, bit_not, bit_or, bit_xor
Comparison equal_to, greater, greater_equal, less, less_equals, not_equal_to
Logical logical_and, logical_not, logical_or
These are all binary function classes, other than bit_not, logical_not, and negate, which are unary. Binary function objects act on two values and return a result, unary function objects act on a single value and return a result. For example, you could calculate the modulus of two numbers with this code:

```
modulus<int> fn;
cout << fn(10, 2) << endl;
```

This declares a function object called fn that will perform modulus. The object is used in the second line, which calls the operator() function on the object with two parameters, so the following line is equivalent to the preceding line:

```
cout << fn.operator()(10, 2) << endl;
```

The result is that the value of 0 is printed on the console. The operator() function merely performs the modulus on the two parameters, in this case 10 % 2. This does not look too exciting. The <algorithm> header contains functions that work on function objects. Most take predicates, that is, logical function objects, but one, transform, takes a function object that performs an action:

```
// #include <algorithm>
// #include <functional>
vector<int> v1 { 1, 2, 3, 4, 5 };
vector<int> v2(v1.size());
fill(v2.begin(), v2.end(), 2);
vector<int> result(v1.size());
transform(v1.begin(), v1.end(), v2.begin(),
result.begin(), modulus<int>());
for (int i : result)
{
cout << i << ' ';
}
cout << endl;
```

This code will perform five modulus calculations on the values in the two vectors. Conceptually, it does this:

```
result = v1 % v2;
```

That is, each item in result is the modulus of the corresponding item in v1 and v2. In the code, the first line creates a vector with the five values. We will calculate the modulus of these values with 2, so the second line declares an empty vector but with the same capacity as the first vector. This second vector is filled by calling the fill function. The first parameter is the address of the first item in the vector and the end function returns the address after the *last* item in the vector. The final item in the function call is the value that will be placed in the vector in every item starting with the item pointed to by the first parameter up to, but excluding, the item pointed to by the second parameter.
At this point, the second vector will contain five items and each one will be 2. Next, a vector is created for the results; and again, it is the same size as the first array. Finally, the calculation is performed by the transform function, shown here again:

```
transform(v1.begin(), v1.end(),
v2.begin(), result.begin(), modulus<int>());
```

The first two parameters give the iterators of the first vector and from this the number of items can be calculated. Since all three vectors are the same size, you only need the begin iterator for v2 and result.
The last parameter is the function object. This is a temporary object and only exists during

this statement; it has no name. The syntax used here is an explicit call to the constructor of the class; it is templated so you need to give the template parameter. The transform function will call the operator(int,int) function on this function object for each item in v1 as the first parameter and the corresponding item in v2 as the second parameter and it will store the result in the corresponding position in result.

Since transform takes any binary function object as the second parameter, you can pass an instance of plus<int> to add a value of 2 to every item in v1, or pass an instance of multiplies<int> to multiply every item in v1 by 2.

One situation where function objects are useful is when performing multiple comparisons using a predicate. A predicate is a function object that compares values and returns a Boolean. The <functional> header contains several classes to allow you to compare items. Let's see how many items in the result container are zero. To do this, we use the count_if function. This will iterate over a container, apply the predicate to every item, and count how many times the predicate returns a value of true. There are several ways to do this. The first defines a predicate function:

```
bool equals_zero(int a)
{
return (a == 0);
}
```

A pointer to this can then be passed to the count_if function:

```
int zeros = count_if(
result.begin(), result.end(), equals_zero);
```

The first two parameters indicate the range of values to check. The last parameter is a pointer to the function that is used as the predicate. Of course, if you are checking for different values you can make this more generic:

```
template<typename T, T value>
inline bool equals(T a)
{
return a == value;
}
```

Call it like this:

```
int zeros = count_if(
result.begin(), result.end(), equals<int, 0>);
```

The problem with this code is that we are defining the operation in a place other than where it is used. The equals function could be defined in another file; however, with a predicate it is more readable to have the code that does the checking defined close to the code that needs the predicate.

The <functional> header also defines classes that can be used as function objects. For example, equal_to<int>, which compares two values. However, the count_if function expects a unary function object, to which it will pass a single value (see the equals_zero function, described previously). equal_to<int> is a binary function object, comparing two values. We need to provide the second operand and to do this we use the helper function called bind2nd:

```
int zeros = count_if(
result.begin(), result.end(), bind2nd(equal_to<int>(), 0));
```

The bind2nd will *bind* the parameter 0 to the function object created from equal_to<int>. Using a function object like this brings the definition of the predicate closer to the function call that will use it, but the syntax looks rather messy. C++11 provides a mechanism to get the compiler to determine the function objects that are required and bind parameters to them. These are called lambda expressions.

Introducing lambda expressions

A lambda expression is used to create an anonymous function object at the location where the function object will be used. This makes your code much more readable because you can see what will be executed. On first sight, a lambda expression looks like a function definition in-place as a function parameter:

```
auto less_than_10 = [](int a) {return a < 10; };
bool b = less_than_10(4);
```

So that we don't have the complication of a function that uses a predicate, in this code we have assigned a variable to the lambda expression. This is not normally how you would use it, but it makes the description clearer. The square brackets at the beginning of the lambda expression are called the **capture list**. This expression does not capture variables, so the brackets are empty. You can use variables declared outside of the lambda expression and these have to be *captured*. The capture list indicates whether all such variables will be captured by a reference (use [&]) or by a value (use [=]). You can also name the variables that will be captured (if there are more than one, use a comma-separated list) and if they are captured by a value, you use just their names. If they are captured by a reference, use a & on their names.

You could make the preceding lambda expression more generic by introducing a variable declared outside of the expression called limit:

```
int limit = 99;
auto less_than = [limit](int a) {return a < limit; };
```

If you compare a lambda expression to a global function, the capture list is a bit like identifying the global variables that the global function can access.

After the caption list, you give the parameter list in parentheses. Again, if you compare a lambda to a function, the lambda parameter list is equivalent to the function parameter list. If the lambda expression does not have any parameters, then you can miss out the parentheses altogether.

The body for the lambda is given in a pair of braces. This can contain anything that can be found in a function. The lambda body can declare local variables, and it can even declare static variables, which looks bizarre, but is legal:

```
auto incr = [] { static int i; return ++i; };
incr();
incr();
cout << incr() << endl; // 3
```

The return value of the lambda is deduced from the item that is returned. A lambda expression does not have to return a value, in which case the expression will return void:

```
auto swap = [](int& a, int& b) { int x = a; a = b; b = x; };
int i = 10, j = 20;
cout << i << " " << j << endl;
swap(i, j);
cout << i << " " << j << endl;
```

The power of lambda expressions is that you can use them in cases when a function object or a predicate is needed:

```
vector<int> v { 1, 2, 3, 4, 5 };
int less_than_3 = count_if(
v.begin(), v.end(),
[](int a) { return a < 3; });
cout << "There are " << less_than_3 << " items less than 3" << endl;
```

Here we declare a vector and initialize it with some values. The count_if function is used to count how many items in the container are less than 3. So, the first two parameters are used to give the range of items to check, and the third parameter is a lambda expression that performs the comparison. The count_if function will call this expression for every item in the range that is passed in via the a parameter of the lambda. The count_if function keeps a running count of how many times the lambda returns true.

Using functions in C++

The example in this chapter uses the techniques you have learned in this chapter to list all the files in a folder, and subfolders, in order of file size, giving a listing of the filenames and their sizes. The example is the equivalent of typing the following at the command line:

```
dir /b /s /os /a-d folder
```

Here, folder is the folder you are listing. The /s option recurses, /a-d removes folders from the list, and /os orders by size. The problem is that without the /b option we get information about each folder, but using it removes the file size in the list. We want a list of filenames (and their paths), their size, ordered by the smallest first.

Start by creating a new folder for this chapter (Chapter_05) under the Beginning_C++ folder. In Visual C++ create a new C++ source file and save it as files.cpp under this new folder. The example will use basic output and strings. It will take a single command line parameter; if more command-line parameters are passed, we just use the first one. Add the following to files.cpp:

```
#include <iostream>
#include <string>
using namespace std;
int main(int argc, char* argv[])
{
```

```
if (argc < 2) return 1;
return 0;
}
```

The example will use the Windows functions, FindFirstFile and FindNextFile, to get information about files that meet a file specification. These return data in a WIN32_FIND_DATAA structure, which has information about the filename, the file size, and file attributes. The functions also return information about folders too, so it means we can test for subfolders and recurse. The WIN32_FIND_DATAA structure gives the file size as a 64-bit number in two parts: the upper and lower 32 bits. We will create our own structure to hold this information. At the top of the file, after the C++ include files, add the following:

```
using namespace std;
#include <windows.h>
struct file_size {
unsigned int high;
unsigned int low;
};
```

The first line is the Windows SDK header file so that you can access the Windows functions, and the structure is used to hold the information about a file's size. We want to compare files by their sizes. The WIN32_FIND_DATAA structure provides the size in two unsigned long members (one with the upper 4 bytes and the other with the lower 4 bytes). We could store this as a 64-bit number, but instead, so that we have an excuse to write some operators, we store the size in our file_size structure. The example will print out file sizes and will compare file sizes, so we will write an operator to insert a file_size object into an output steam; since we want to order the files by size, we need an operator to determine if one file_size object is greater than the other.

The code will use Windows functions to get information about the files, in particular their name and size. This information will be stored in a vector, so at the top of the file add these two highlighted lines:

```
#include <string>
#include <vector>
#include <tuple>
```

The tuple class is needed so that we can store both a string (the filename) and a file_size object as each item in the vector. To make the code more readable add the following alias after the structure definition:

```
using file_info = tuple<string, file_size>;
```

Then just above the main function add the skeleton code for the function that will get the file in a folder:

```
void files_in_folder(
const char *folderPath, vector<file_info>& files)
{
}
```

This function takes a reference to a vector and a folder path. The code will go through each item in the specified folder. If it is a file, it will store the details in the vector; otherwise, if the item is a folder it will call itself to get the files in that subfolder. Add a call to this function at the bottom of the main function:

```
vector<file_info> files;
files_in_folder(argv[1], files);
```

The code has already checked that there is at least one command line argument, and we use this as the folder to examine. The main function should print out the file information, so we declare a vector on the stack and pass this by reference to the files_in_folder function.
This code does nothing so far, but you can compile the code to make sure that there are no typos (remember to use the /EHsc parameter).
Most of the work is carried out in the files_in_folder function. As a start, add the following code to this function:

```
string folder(folderPath);
folder += "*";
WIN32_FIND_DATAA findfiledata {};
void* hFind = FindFirstFileA(folder.c_str(), &findfiledata);
if (hFind != INVALID_HANDLE_VALUE)
{
do
{
} while (FindNextFileA(hFind, &findfiledata));
FindClose(hFind);
}
```

We will use the ASCII version of the functions (hence the trailing A on the structure and function names). The FindFirstFileA function takes a search path, and in this case, we use the name of a folder suffixed with a *, meaning *everything in this folder*. Notice that the Windows function wants a const char* parameter, so we use the c_str function on the string object.

If the function call succeeds and it finds an item that meets this criterion, then the function fills in the WIN32_FIND_DATAA structure passed by the reference and it also returns an opaque pointer which will be used to make subsequent calls on this search (you do not need to know what it points to). The code checks to see if the call was successful, and if so, it repeatedly calls FindNextFileA to get the next item until this function returns 0, indicating there are no more items. The opaque pointer is passed to FindNextFileA so that it knows which search is being checked. When the search is complete, the code calls FindClose to release whatever resources Windows allocates for the search.
The search will return both file and folder items; to handle each differently, we can test the dwFileAttributes member of the WIN32_FIND_DATAA structure. Add the following code in the do loop:

```
string findItem(folderPath);
findItem += "";
findItem += findfiledata.cFileName;
if ((findfiledata.dwFileAttributes & FILE_ATTRIBUTE_DIRECTORY) != 0)
{
// this is a folder so recurse
}
else
{
// this is a file so store information
}
```

The WIN32_FIND_DATAA structure contains just the relative name of the item in the folder, so the first few lines create an absolute path. The following lines test to see if the item is a folder (directory) or a file. If the item is a file, then we simply add it to the vector passed to the function. Add the following to the else clause:

```
file_size fs{};
fs.high = findfiledata.nFileSizeHigh;
fs.low = findfiledata.nFileSizeLow;
files.push_back(make_tuple(findItem, fs));
```

The first three lines initialize a file_size structure with the size data, and the last line adds a tuple with the name of the file and its size to the vector. So that you can see the results of a simple call to this function, add the following to the bottom of the main function:

```
for (auto file : files)
{
cout << setw(16) << get<1>(file) << " "
<< get<0>(file) << endl;
}
```

This iterates through the items in the files vector. Each item is a tuple<string, file_size> object and to get the string item, you can use the Standard Library function, get, using 0 as the function template parameter, and to get the file_size object you call get with 1 as the function template parameter. The code calls the setw manipulator to make sure that the file sizes are always printed in a column 16 characters wide. To use this, you need to add an include for <iomanip> at the top of the file. Notice that get<1> will return a file_size object and this is inserted into cout. As it stands, this code will not compile because there is no operator to do this. We need to write one.

After the definition of the structure, add the following code:

```
ostream& operator<<(ostream& os, const file_size fs)
{
int flags = os.flags();
unsigned long long ll = fs.low +
((unsigned long long)fs.high << 32);
os << hex << ll;
os.setf(flags);
return os;
}
```

This operator will alter the ostream object, so we store the initial state at the beginning of the function and restore the object to this state at the end. Since the file size is a 64-bit number, we convert the constituent parts of the file_size object and then print it out as a hexadecimal number.

Now you can compile and run this application. For example:

files C:windows

This will list the names and sizes of the files in the windows folder.

There are two more things that need to be done--recurse subfolders and sort the data. Both are straightforward to implement. In the files_in_folder function, add the following code to the code block of the if statement:

```
// this is a folder so recurse
string folder(findfiledata.cFileName);
// ignore . and .. directories
if (folder != "." && folder != "..")
{
```

```
files_in_folder(findItem.c_str(), files);
}
```

The search will return the . (current) folder and .. (parent) folder, so we need to check for these and ignore them. The next action is to recursively call the files_in_folder function to obtain the files in the subfolder. If you wish, you can compile and test the application, but this time it is best to test the code using the Beginning_C++ folder because recursively listing the Windows folder will produce a lot of files.

The code returns the list of files as they were obtained, but we want to see them in order of file size. To do this we can use the sort function in the <algorithm> header, so add an include to this after the include for <tuple>. In the main function, after the call to files_in_folder, add this code:

```
files_in_folder(argv[1], files);
sort(files.begin(), files.end(),
[](const file_info& lhs, const file_info& rhs) {
return get<1>(rhs) > get<1>(lhs);
} );
```

The first two parameters of the sort function indicate the range of items to check. The third item is a predicate, and the function will pass two items from the vector to the predicate. You have to return a value of true if the two parameters are in order (the first is smaller than the second).

The predicate is provided by a lambda expression. There are no captured variables so the expression starts with [] and this is followed by the parameter list of the items being compared by the sort algorithm (passed by const reference, because they will not be changed). The actual comparison is carried out between the braces. Since we want to list the files in ascending order, we have to ensure that the second of the two is bigger than the first. In this code, we use the > operator on the two file_size objects. So that this code will compile, we need to define this operator. After the insertion operator add the following:

```
bool operator>(const file_size& lhs, const file_size& rhs)
{
if (lhs.high > rhs.high) return true;
if (lhs.high == rhs.high) {
if (lhs.low > rhs.low) return true;
}
return false;
}
```

You can now compile the example and run it. You should find that the files in the specified folder and subfolders are listed in order of the size of the files.

C++ allows you to create your own types. These custom types can have operators and can be converted to other types; indeed, they can be used like built-in types with the behavior that you define. This facility uses a language feature called classes. The advantage of being able to define your own types is that you can encapsulate data in objects of your chosen type, and use the type to manage the lifetime of that data. You can also define the actions that can be performed on that data. In other words, you are able to define custom types that have state and behavior, which is the basis of object-orientated programming.

Writing classes

When you use built-in types, the data is directly available to whatever code has access to that data. C++ provides a mechanism (const) to prevent write access, but any code can use const_cast to cast away const-ness. Your data could be complex, such as a pointer to a file mapped into memory with the intention that your code will change a few bytes and then write the file back to disk. Such raw pointers are dangerous because other code with access to the pointer could change part of the buffer that should not be changed. What is needed is a mechanism to encapsulate the data into a type that knows what bytes to change, and only allow that type to access the data. This is the basic idea behind classes.

Reviewing structures

We have already seen one mechanism in C++ to encapsulate data: struct. A structure allows you to declare data members that are built-in types, pointers, or references. When you create a variable from that struct, you are creating an **instance** of the structure, also known as an **object**. You can create variables that are references to this object or pointers that point to the object. You can even pass the object by value to a function where the compiler will make a copy of the object (it will call the *copy constructor* for the struct). We have seen that with a struct any code that has access to an instance (even through a pointer or reference) can access the members of the object (although this can be changed). Used like this, a struct can be thought of as **aggregate** types containing the state.
The members of an instance of a struct can be initialized by accessing them directly with the dot operator or using the -> operator through a pointer to the object. We have also seen that you can initialize an instance of a struct with an initializer list (in braces). This is quite restrictive because the initializer list has to match the data members in the struct. In Chapter 4, *Working with Memory, Arrays, and Pointers*, you saw that you can have a pointer as a member of a struct, but you have to explicitly take appropriate action to release the memory pointed to by the pointer; if you don't, then this could result in a memory leak.
A struct is one of the class types that you can use in C++; the other two are union and class. Custom types defined as struct or class can have behaviors as well as state, and C++ allows you to define some special functions to control how instances are created and destroyed, copied, and converted. Furthermore, you can define operators on a struct or class type so that you can use the operators on instances in a similar way to using the operators on built-in types. There is a difference between struct and class which we will address later, but in general the rest of the chapter will be about classes and when a class is mentioned you can usually assume the same applies to a struct as well.

Defining classes

A class is defined in a statement, and it will define its members in a block with multiple statements enclosed by braces {}. As it's a statement, you have to place a semicolon after the last brace. A class can be defined in a header file (as are many of the **C++ Standard Library** classes), but you have to take steps to ensure that such files are included only once in a source file. Chapter 1, *Starting with C++*, describes how to do this with #pragma once, conditional compilation, and precompiled header files. There are, however, some rules about specific items in a class that must be defined in a source file, which will be covered later.

If you peruse the C++ Standard Library, you will see that classes contain member functions and, in an attempt to put all the code for a class into a single header file, this makes the code difficult to read and difficult to understand. This may be justifiable for a library file maintained by a legion of expert C++ programmers, but for your own projects readability should be a key design goal. For this reason, a C++ class can be declared in a C++ header file, including its member functions, and the actual implementation of the functions can be placed in a source file. This makes the header files easier to maintain and more reusable.

Defining class behavior

A class can define functions that can only be called through an instance of the class; such a function is often called a **method**. An object will have state; this is provided by the data members defined by the class and initialized when the object is created. The methods on an object define the behavior of the object, usually acting upon the state of the object. When you design a class, you should think of the methods in this way: they describe the object doing something.

```
class cartesian_vector
{
public:
double x;
double y;
// other methods
double get_magnitude() { return std::sqrt((x * x) + (y * y)); }
};
```

This class has two data members, x and y, which represent the direction of a twodimensional vector resolved in the Cartesian x and y directions. The public keyword means that any members defined after this specifier are accessible by code defined outside of the class. By default, all the members of a class are private unless you indicate otherwise. Such access specifiers will be covered in more depth in the next chapter, but private means that the member can only be accessed by other members of the class.
This is the difference between a struct and a class: by default, members of a struct are public and by default, members of a class are private.

This class has a method called get_magnituide that will return the length of the Cartesian vector. This function acts upon the two data members of the class and returns a value. This is a type of **accessor** method; it gives access to the state of the object. Such a method is typical on a class, but there is no requirement that methods return values. Like functions, a method can also take parameters. The get_magnituide method can be called like this:

```
cartesian_vector vec { 3.0, 4.0 };
double len = vec.get_magnitude(); // returns 5.0
```

Here a cartesian_vector object is created on the stack and list initializer syntax is used to initialize it to a value representing a vector of (3,4). The length of this vector is 5, which is the value returned by calling get_magnitude on the object.

Using the this pointer

The methods in a class have a special calling convention, which in Visual C++ is called __thiscall. The reason is that every method in a class has a hidden parameter called this, which is a pointer of the class type to the current instance:

```
class cartesian_vector
{
public:
double x;
double y;
// other methods
double get_magnitude()
{
return std::sqrt((this->x * this->x) + (this->y * this->y));
}
};
```

Here, the get_magnitude method returns the length of the cartesian_vector object. The members of the object are accessed through the -> operator. As shown previously, the members of the class can be accessed without the this pointer, but it does make it explicit that the items are members of the class.
You could define a method on the cartesian_vector type that allows you to change its state:

```
class cartesian_vector
{
public:
```

```
double x;
double y;
```

```
reset(double x, double y) { this->x = x; this->y = y; }
// other methods
};
```
The parameters of the reset method have the same names as the data members of the class; however, since we use the this pointer the compiler knows that this is not ambiguous.
You can dereference the this pointer with the * operator to get access to the object. This is useful when a member function must return a reference to the current object (as some operators will, as we will see later) and you can do this by returning *this. A method in a class can also pass the this pointer to an external function, which means that it is passing the current object by reference through a typed pointer.

Using the scope resolution operator

You can define a method inline in the class statement, but you can also separate the declaration and implementation, so the method is declared in the class statement but it is defined elsewhere. When defining a method out of the class statement, you need to provide the method with the name of the type using the scope resolution operator. For example, using the previous cartesian_vector example:
```
class cartesian_vector
{
public:
double x;
double y;
// other methods
double magnitude();
};
double cartesian_vector::magnitude()
{
return sqrt((this->x * this->x) + (this->y * this->y));
}
```
The method is defined outside the class definition; it is, however, still the class method, so it has a this pointer that can be used to access the object's members. Typically, the class will be declared in a header file with prototypes for the methods and the actual methods will be implemented in a separate source file. In this case, using the this pointer to access the class members (methods and data members) make it obvious, when you take a cursory look at a source file, that the functions are methods of a class.

Defining class state

Your class can have built-in types as data members, or custom types. These data members can be declared in the class (and created when an instance of the class is constructed), or they can be pointers to objects created in the free store or references to objects created elsewhere. Bear in mind that if you have a pointer to an item created in the free store, you need to know whose responsibility it is to deallocate the memory that the pointer points to.
If you have a reference (or pointer) to an object created on a stack frame somewhere, you need to make sure that the objects of your class do not live longer than that stack frame.
When you declare data members as public it means that external code can read and write to the data members. You can decide that you would prefer to only give read-only access, in which case you can make the members private and provide read access through accessors:
```
class cartesian_vector
{
double x;
double y;
public:
double get_x() { return this->x; }
double get_y() { return this->y; }
// other methods
};
```

When you make the data members private it means that you cannot use the initializer list syntax to initialize an object, but we will address this later. You may decide to use an accessor to give write access to a data member and use this to check the value.

```
void cartesian_vector::set_x(double d)
{
if (d > -100 && d < 100) this->x = d;
}
```

This is for a type where the range of values must be between (but not including) -100 and 100.

Creating objects

You can create objects on the stack or in the free store. Using the previous example, this is as follows:

```
cartesian_vector vec { 10, 10 };
cartesian_vector *pvec = new cartesian_vector { 5, 5 };
// use pvec
delete pvec
```

This is **direct initialization** of the object and assumes that the data members of cartesian_vector are public. The vec object is created on the stack and initialized with an initializer list. In the second line, an object is created in the free store and initialized with an initializer list. The object on the free store must be freed at some point and this is carried out by deleting the pointer. The new operator will allocate enough memory in the free store for the data members of the class and for any of the infrastructure the class needs (as described in the next chapter).

A new feature of C++11 is to allow direct initialization to provide default values in the class:

```
class point
{
public:
int x = 0;
int y = 0;
};
```

This means that if you create an instance of point without any other initialization values, it will be initialized so that x and y are both zero. If the data member is a built-in array, then you can provide direct initialization with an initialization list in the class:

```
class car
{
public:
double tire_pressures[4] { 25.0, 25.0, 25.0, 25.0 };
};
```

The C++ Standard Library containers can be initialized with an initialize list, so, in this class for tire_pressures, instead of declaring the type to be double[4] we could use vector<double> or array<double,4>, and initialize it in the same way.

Construction of objects

C++ allows you to define special methods to perform the initialization of the object. These are called **constructors**. In C++11, you will get three such functions generated for you by default, but you can provide your own versions if you wish. These three constructors, along with three other related functions, are as follows:

Default constructor: This is called to create an object with the *default* value.
Copy constructor: This is used to create a new object based on the value of an existing object.
Move constructor: This is used to create a new object using the data moved from an existing object.
Destructor: This is called to clean up the resources used by an object.
Copy assignment: This copies the data from one existing object into another existing object.
Move assignment: This moves the data from one existing object into another existing object.

The compiler-created versions of these functions will be implicitly `public`; however, you may decide to prevent copying or assigning by defining your own versions, and making them `private`, or you can delete them using the `=delete` syntax. You can also provide your own constructors that will take any parameters you decide you need to initialize a new object.

A constructor is a member function that has the same name as the type, but does not return a value, so you cannot return a value if the construction fails, which potentially means that the caller will receive a partially constructed object. The only way to handle this situation is to throw an exception (explained in Chapter 10, *Diagnostics and Debugging*).

Defining constructors

The default constructor is used when an object is created without a value and hence the object will have to be initialized with a default value. The `point` declared previously could be implemented like this:

```
class point
{
double x; double y;
public:
point() { x = 0; y = 0; }
};
```

This explicitly initializes the items to a value of zero. If you want to create an instance with the default values, you do not include parentheses.

```
point p; // default constructor called
```

It is important to be aware of this syntax because it is easy to write the following by mistake:

```
point p(); // compiles, but is a function prototype!
```

This will compile because the compiler will think you are providing a function prototype as a forward declaration. However, you'll get an error when you attempt to use the symbol `p` as a variable. You can also call the default constructor using initialize list syntax with empty braces:

```
point p {}; // calls default constructor
```

Although it does not matter in this case, where the data members are built-in types, initializing data members in the body of the constructor like this involves a call to the assignment operator of the member type. A more efficient way is to use direct initialization with a **member list**.

The following is a constructor that takes two parameters, which illustrates a member list:

```
point(double x, double y) : x(x), y(y) {}
```

The identifiers outside the parentheses are the names of class members, and the items inside the parentheses are expressions used to initialize that member (in this case, a constructor parameter). This example uses `x` and `y` for the parameter names. You don't have to do this; this is only given here as an illustration that the compiler will distinguish between the parameters and data members. You can also use braced initializer syntax in the member list of a constructor:

```
point(double x, double y) : x{x}, y{y} {}
```

You call this constructor when you create an object like this:

```
point p(10.0, 10.0);
```

You can also create an array of objects:

```
point arr[4];
```

This creates four point objects, which can be accessed by indexing the arr array. Note that when you create an array of objects the *default* constructor is called on the items; there is no way to call any other constructor, and so you have to initialize each one separately.

You can also provide default values for constructor parameters. In the following code, the car class has values for the four tires (the first two are the front tires) and for the spare tire. There is one constructor that has mandatory values that will be used for the front and back tires, and an optional value for the spare. If a value is not provided for the spare tire pressure, then a default value will be used:

```
class car
{
array<double, 4> tire_pressures;;
double spare;
public:
car(double front, double back, double s = 25.0)
: tire_pressures{front, front, back, back}, spare{s} {}
};
```

This constructor can be called with either two values or three values:

```
car commuter_car(25, 27);
car sports_car(26, 28, 28);
```

Delegating constructors

A constructor may call another constructor using the same member list syntax:

```
class car
{
// data members
public:
car(double front, double back, double s = 25.0)
: tire_pressures{front, front, back, back}, spare{s} {}
car(double all) : car(all, all) {}
};
```

Here, the constructor that takes one value delegates to the constructor that takes three parameters (in this case using the default value for the spare).

Copy constructor

A copy constructor is used when you pass an object by value (or return by value) or if you explicitly construct an object based on another object. The last two lines of the following both create a point object from another point object, and in both cases the copy constructor is called:

```
point p1(10, 10);
```

```
point p2(p1);
point p3 = p1;
```

The last line looks like it involves the assignment operator, but it actually calls the copy constructor. The copy constructor could be implemented like this:

```
class point
{
int x = 0;int y = 0;
public:
point(const point& rhs) : x(rhs.x), y(rhs.y) {}
};
```

The initialization accesses the private data members on another object (rhs). This is acceptable because the constructor parameter is the same type as the object being created. The copy operation may not be as simple as this. For example, if the class contains a data member that is a pointer, you will most likely want to copy the data that the pointer points to, and this will involve creating a new memory buffer in the new object.

Converting between types

You can also perform conversions. In math, you can define a vector that represents direction, so that the line drawn between two points is a vector. In our code we have already defined a point class and a cartesian_vector class. You could decide to have a constructor that creates a vector between the origin and a point, in which case you are converting a point object to a cartesian_vector object:

```
class cartesian_vector
{
double x; double y;
public:
cartesian_vector(const point& p) : x(p.x), y(p.y) {}
};
```

There is a problem here, which we will address in a moment. The conversions can be called like this:

```
point p(10, 10);
cartesian_vector v1(p);
cartesian_vector v2 { p };
cartesian_vector v3 = p;
```

Making friends

The problem with the code above is that the cartesian_vector class accesses private members of the point class. Since we have written both classes, we are happy to bend the rules, and so we make the cartesian_vector class a friend of the point class:

```
class cartesian_vector; // forward decalartion
class point
{
double x; double y;
public:
point(double x, double y) : x(x), y(y){}
friend class cartesian_point;
};
```

Since the cartesian_vector class is declared after the point class, we have to provide a forward declaration that essentially tells the compiler that the name cartesian_vector is about to be used and it will be declared elsewhere. The important line starts with friend. This indicates that the code for the entire class, cartesian_vector, can have access to the private members (data and methods) of the point class.

You can also declare friend functions. For example, you could declare an operator such that a point object can be inserted into the cout object, so it can be printed to the console. You cannot change the ostream class, but you can define a global method:

```
ostream& operator<<(ostream& stm, const point& pt)
{
stm << "(" << pt.x << "," << pt.y << ")";
return stm;
}
```

This function accesses the private members of point so you have to make the function a friend of the point class with:

```
friend ostream& operator<<(ostream&, const point&);
```

Such friend declarations have to be declared in the point class, but it is irrelevant whether it is put in the public or private section.

Marking constructors as explicit

In some cases, you do not want to allow the implicit conversion between one type that is passed as a parameter of the constructor of another type. To do this, you need to mark the constructor with the explicit specifier. This now means that the only way to call the constructor is using the parentheses syntax: *explicitly* calling the constructor. In the following code, you cannot implicitly convert a double to an object of mytype:

```
class mytype
{
public:
explicit mytype(double x);
};
```

Now you have to *explicitly* call the constructor if you want to create an object with a double parameter:

```
mytype t1 = 10.0; // will not compile, cannot convert
mytype t2(10.0); // OK
```

Destructing objects

When an object is destroyed, a special method called the destructor is called. This method has the name of the class prefixed with a ~ symbol and it does not return a value.
If the object is an automatic variable, on the stack, then it will be destroyed when the variable goes out of scope. When an object is passed by value, a copy is made on the called function's stack and the object will be destroyed when the called function completes. Furthermore, it does not matter how the function completes, whether an explicit call to return or reaching the final brace, or if an exception is thrown; in all of these cases, the destructor is called. If there are multiple objects in a function, the destructors are called in the reverse order to the construction of the objects in the same scope. If you create an array of objects, then the default constructor is called for each object in the array on the statement that declares the array, and all the objects will be destroyed--and the destructor on each one is called, when the array goes out of scope.

Here are some examples, for a class mytype:

```
void f(mytype t) // copy created
{
// use t
} // t destroyed
void g()
{
mytype t1;
f(t1);
if (true)
{
mytype t2;
} // t2 destroyed
mytype arr[4];
} // 4 objects in arr destroyed in reverse order to creation
// t1 destroyed
```

An interesting action occurs when you return an object. The following annotation is what you would expect:

```
mytype get_object()
{
mytype t; // default constructor creates t
return t; // copy constructor creates a temporary
} // t destroyed
void h()
{
```

```
test tt = get_object(); // copy constructor creates tt
} // temporary destroyed, tt destroyed
```

In fact, the process is more streamlined. In a debug build, the compiler will see that the temporary object created on the return of the get_object function is the object that will be used as the variable tt, and so there is no extra copy on the return value of the get_object function. The function actually looks like this:

```
void h()
{
mytype tt = get_object();
} // tt destroyed
```

However, the compiler is able to optimize the code further. In a release build (with optimizations enabled), the temporary will not be created and the object tt in the calling function will be the actual object t created in get_object.

An object will be destroyed when you explicitly delete a pointer to an object allocated on the free store. In this case, the call to the destructor is deterministic: it is called when your code calls delete. Again, with the same class mytype, this is as follows:

```
mytype *get_object()
{
return new mytype; // default constructor called
}
void f()
{
mytype *p = get_object();
// use p
delete p; // object destroyed
}
```

There will be times when you want to use the deterministic aspect of deleting an object (with the possible danger of forgetting to call delete) and there will be times when you prefer to have the reassurance that an object is to be destroyed at an appropriate time (with the potential that it may be much later in time).

If a data member in a class is a custom type with a destructor, then when the containing object is destroyed the destructors on the contained objects are called too. Nonetheless, note that this is only if the *object* is a class member. If a class member is a pointer to an object in the free store, then you have to explicitly delete the pointer in the containing object's destructor. However, you need to know where the object the pointer points to is because if it is not in the free store, or if the object is used by other objects, calling delete will cause problems.

Assigning objects

The assignment operator is called when an *already created* object is assigned to the value of another one. By default, you will get a copy assignment operator that will copy all the data members. This is not necessarily what you want, particularly if the object has a data member that is a pointer, in which case your intention is more likely to do a deep copy and copy the data pointed to rather than the value of the pointer (in the latter case, *two* objects will point to the same data).

If you define a copy constructor, you will still get the default copy assignment operator; however, it makes sense that if you regard it important to write your own copy constructor, you should also provide a custom copy assignment operator. (Similarly, if you define a copy assignment operator, you will get the default copy constructor unless you define it.)

The copy assignment operator is typically a public member of the class and it takes a const reference to the object that will be used to provide the values for the assignment. The semantics of the assignment operator are that you can chain them, so, for example, this code calls the assignment operator on two of the objects:

```
buffer a, b, c; // default constructors called
// do something with them
a = b = c; // make them all the same value
a.operator=(b.operator=(c)); // make them all the same value
```

The last two lines do the same thing, but clearly the first is more readable. To enable these semantics, the assignment operator must return a reference to the object that has been assigned. So, the class buffer will have the following method:

```
class buffer
{
// data members
public:
buffer(const buffer&); // copy constructor
buffer& operator=(const buffer&); // copy assignment
};
```

Although the copy constructor and copy assignment methods appear to do similar things, there is a key difference. A copy constructor creates a new object that did not exist before the call. The calling code is aware that if the construction fails, then an exception will be raised. With assignment, both objects already exist, so you are copying the value from one object to another. This should be treated as an atomic action and all the copy should be performed; it is not acceptable for the assignment to fail halfway through, resulting in an object that is a bit of both objects. Furthermore, in construction, an object only exists after the construction is successful, so a copy construction cannot happen on an object itself, but it is perfectly legal (if pointless) for code to assign an object to itself. The copy assignment needs to check for this situation and take appropriate action.
There are various strategies to do this, and a common one is called the copy-and-swap idiom because it uses the Standard Library swap function that is marked as noexcept, and will not throw an exception. The idiom involves creating a temporary copy of the object on the right-hand side of the assignment and then swapping its data members with the data members of the object on the left-hand side.

Move semantics

C++11 provides move semantics through a move constructor and a move assignment operator, which are called when a temporary object is used either to create another object or to be assigned to an existing object. In both cases, because the temporary object will not live beyond the statement, the contents of the temporary can be moved to the other object, leaving the temporary object in an invalid state. The compiler will create these functions for you through the default action of moving the data from the temporary to the newly created (or the assigned to) object.
You can write your own versions, and to indicate move semantics these have a parameter that is an rvalue reference (&&).
If you want the compiler to provide you with a default version of any of these methods, you can provide the prototype in the class declaration suffixed with =default. In most cases, this is self-documenting rather than being a requirement, but if you are writing a POD class you must use the default versions of these functions, otherwise is_pod will not return true.
If you want to use only move and never to use copy (for example, a file handle class), then you can *delete* the copy functions:

```
class mytype
{
int *p;
public:
mytype(const mytype&) = delete; // copy constructor
mytype& operator= (const mytype&) = delete; // copy assignment
mytype&(mytype&&); // move constructor
mytype& operator=(mytype&&); // move assignment
};
```

This class has a pointer data member and allows move semantics, in which case the move constructor will be called with a reference to a temporary object. Since the object is temporary, it will not survive after the move constructor call. This means that the new object can *move* the state of the temporary object into itself:

```
mytype::mytype(mytype&& tmp)
{
this->p = tmp.p;
tmp.p = nullptr;
}
```

The move constructor assigns the temporary object's pointer to nullptr, so that any destructor defined for the class does not attempt to delete the pointer.

Declaring static members

You can declare a member of a class--a data member or a method--static. This is similar in some ways to how you use the static keyword on automatic variables and functions declared at file scope, but there are some important, and different, properties to this keyword when used on a class member.

Defining static members

When you use static on a class member it means that the item is associated with the class and not with a specific instance. In the case, of data members, this means that there is one data item shared by all instances of the class. Likewise, a static method is not attached to an object, it is not __thiscall and has no this pointer.

A static method is part of the namespace of a class, so it can create objects for the class and have access to their private members. A static method has the __cdecl calling convention by default, but you can declare it as __stdcall if you wish. This means that, you can write a method within the class that can be used to initialize C-like pointers, which are used by many libraries. Note that the static function cannot call nonstatic methods on the class because a nonstatic method will need a this pointer, but a nonstatic method can call a static method.

A nonstatic method is called through an object, either using the dot operator (for a class instance) or the -> operator for an object pointer. A static method does not need an associated object, but it can be called through one. This gives two ways to call a static method, through an object or through the class name:

```
class mytype
{
public:
static void f(){}
void g(){ f(); }
};
```

Here, the class defines a static method called f and a nonstatic method called g. The nonstatic method g can call the static method, but the static method f cannot call the nonstatic method. Since the static method f is public, code outside the class can call it:

```
mytype c;
c.g(); // call the nonstatic method
c.f(); // can also call the static method thru an object
mytype::f(); // call static method without an object
```

Although the static function can be called through an object, you do not have to create any objects at all to call it.

Static data members need a bit more work because when you use static it indicates that the data member is not part of an object, and usually data members are allocated when an object is created. You have to define static data members outside of the class:

```
class mytype
{
public:
static int i;
static void incr() { i++; }
};
// in a source file
int mytype::i = 42;
```

The data member is defined outside of the class at file scope. It is named using the class name, but note that it also has to be defined using the type. In this case the data member is initialized with a value; if you do not do this, then on the first use of the variable it will have the default value of the type (in this case, zero). If you choose to declare the class in a header file (which is common), the definition of the static data members must be in a source file.
You can also declare a variable in a method that is static. In this case, the value is maintained across method calls, in all objects, so it has the same effect as a static class member, but you do not have the issue of defining the variable outside of the class.

Using static and global objects

A static variable in a global function will be created at some point before the function is first called. Similarly, a static object that is a member of a class will be initialized at some point before it is first accessed.

Static and global objects are constructed before the main function is called, and destroyed after the main function finishes. The order of this initialization has some issues. The C++ standard says that the initialization of static and global objects defined in a source file will occur before any function or object defined in that source file is used, and if there are several global objects in a source file, they will be initialized in the order that they are *defined*. The issue is if you have several source files with static objects in each. There is no guarantee on the order in which these objects will be initialized. It becomes a problem if one static object depends on another static object because you cannot guarantee that the dependent object will be created after the object it depends upon.

Named constructors

This is one application for public static methods. The idea is that since the static method is a member of the class it means that it has access to the private members of an instance of the class, so such a method can create an object, perform some additional initialization, and then return the object to the caller. This is a **factory method**. The point class used so far has been constructed using Cartesian points, but we could also create a point based on polar co-ordinates, where the (x, y) Cartesian co-ordinates can be calculated as:

```
x = r * cos(theta)
y = r * sin(theta)
```

Here r is the length of the vector to the point and theta is the angle of this vector counterclockwise to the x axis. The point class already has a constructor that takes two double values, so we cannot use this to pass polar co-ordinates; instead, we can use a static method as a *named constructor*:

```
class point
{
double x; double y;
public:
point(double x, double y) : x(x), y(y){}
static point polar(double r, double th)
{
```

```
return point(r * cos(th), r * sin(th));
}
};
```

The method can be called like this:

```
const double pi = 3.141529;
const double root2 = sqrt(2);
point p11 = point::polar(root2, pi/4);
```

The object p11 is the point with the Cartesian co-ordinates of (1,1). In this example the polar method calls a public constructor, but it has access to private members, so the same method could be written (less efficiently) as:

```
point point::polar(double r, double th)
{
point pt;
pt.x = r * cos(th);
pt.y = r * sin(th);
return pt;
}
```

Nested classes

You can define a class within a class. If the nested class is declared as public, then you can create objects in the container class and return them to external code. Typically, however, you will want to declare a class that is used by the class and should be private. The following declares a public nested class:

```
class outer
{
public:
class inner
{
public:
void f();
};
inner g() { return inner(); }
};
void outer::inner::f()
{
// do something
}
```

Notice how the name of the nested class is prefixed with the name of the containing class.

Accessing const objects

You have seen many examples so far of using const, and perhaps the most frequent is when it is applied to a reference as a function parameter to indicate to the compiler that the function only has read-only access to the object. Such a const reference is used so that objects are passed by reference to avoid the overhead of the copying that would occur if the object were passed by value. Methods on a class can access the object data members and, potentially, can change them, so if you pass an object through a const reference the compiler will only allow the reference to call methods that do not change the object. The point class defined earlier had two accessors to access the data in the class:

```
class point
{
double x; double y;
public:
double get_x() { return x; }
double get_y() { return y: }
};
```

If you define a function that takes a const reference to this and you attempt to call these accessors, you will get an error from the compiler:

```
void print_point(const point& p)
{
cout << "(" << p.get_x() << "," << p.get_y() << ")" << endl;
}
```

The error from the compiler is a bit obscure:

cannot convert 'this' pointer from 'const point' to 'point &'

This message is the compiler complaining that the object is const, it is immutable, and it does not know whether these methods will preserve the state of the object. The solution is simple--add the const keyword to methods that do not change the object state, like this:

```
double get_x() const { return x; }
double get_y() const { return y: }
```

This effectively means that the this pointer is const. The const keyword is part of the function prototype, so the method can be overloaded on this. You can have one method that is called when it is called on a const object and another called on a non-const object. This enables you to implement a copy-on-write pattern where, for example, a const method would return read-only access to the data and the non-const method would return a *copy* of the data that is writeable.

Of course, a method marked with const must not alter the data members, not even temporarily. So, such a method can only call const methods. There may be rare cases when a data member is designed to be changed through a const object; in this case the declaration of the member is marked with the mutable keyword.

Using objects with pointers

Objects can be created on the free store and accessed through a typed pointer. This gives more flexibility because it is efficient to pass pointers to functions, and you can explicitly determine the lifetime of the object because an object is created with the call to new and destroyed by the call to delete.

Getting pointers to object members

If you need to get access to the address of a class data member through an instance (assuming the data member is public), you simply use the & operator:

```
struct point { double x; double y; };
point p { 10.0, 10.0 };
int *pp = &p.x;
```

In this case struct is used to declare point so that the members are public by default.

The second line uses an initialization list to construct a point object with two values, and then the final line gets a pointer to one of the data members. Of course, the pointer cannot be used after the object has been destroyed. Data members are allocated in memory (in this case on the stack), so the address operator merely gets a pointer to that memory.

Function pointers are a different case. There will only be one copy of the method in memory, regardless of how many instances of the class are created, but because methods are called using the __thiscall calling convention (with a hidden this parameter) you have to have a function pointer that can be initialized with a pointer to an object to provide the this pointer. Consider this class:

```
class cartesian_vector
{
public:
// other items
double get_magnitude() const
{
```

```
return std::sqrt((this->x * this->x) + (this->y * this->y));
}
};
```
We can define a function pointer to the get_magnitude method like this:
```
double (cartesian_vector::*fn)() const = nullptr;
fn = &cartesian_vector::get_magnitude;
```
The first line declares a function pointer. This is similar to the C function pointer declarations except that there is an inclusion of the class name in the pointer type. This is needed so that the compiler knows that it has to provide a this pointer in any call through this pointer. The second line obtains a pointer to the method. Notice that no object is involved. You are not getting a function pointer to a method on an object; you are getting a pointer to a method on a class that must be called through an object. To call the method through this pointer, you need to use the pointer to the member operator .* on an object:
```
cartesian_vector vec(1.0, 1.0);
double mag = (vec.*fn)();
```
The first line creates an object and the second line calls the method. The pointer to the member operator says that the function pointer on the *right* is called with the object on the *left*. The address of the object on the left is used for the this pointer when the method is called. As this is a method, we need to provide a parameter list, which in this case is empty (if you have parameters, they would be in the pair of parentheses on the right of this statement). If you have an object pointer, then the syntax is similar, but you use the ->* pointer to the member operator:
```
cartesian_vector *pvec = new cartesian_vector(1.0, 1.0);
double mag = (pvec->*fn)();
delete pvec;
```

Operator overloading

One of behaviors of a type is the operations you can apply to it. C++ allows you to overload the C++ operators as part of a class so that it's clear that the operator is acting upon the type. This means that for a unary operator the member method should have no parameters and for a binary operator you need only one parameter, since the current object will be on the left of the operator, and hence the method parameter is the item on the right. The following table summarizes how to implement unary and binary operators, and four exceptions:

Expression	**Name**	**Member method**	**Non-member function**
+a/-a	Prefix unary	operator()	operator(a)
a, b	Binary	operator(b)	operator(a,b)
a+/a-	Postfix unary	operator(0)	operator(a,0)
a=b	Assignment	operator=(b)	
a(b)	Function call	operator()(b)	
a[b]	Indexing	operator[](b)	
a->	Pointer access	operator->()	

Here the ■ symbol is used to indicate any of the acceptable unary or binary operators except for the four operators mentioned in the table.

There are no strict rules over what an operator should return, but it helps if an operator on a custom type behaves like operators on a built-in type. There also has to be some consistency. If you implement the + operator to add two objects together, then the same plus action should be used for the += operator. Also, you could argue that the plus action will also determine what the minus action should be like, and hence the - and -= operators. Similarly, if you want to define the < operator, then you should define <=. >, >=, ==, and != too.

The Standard Library's algorithms (for example, sort) will only expect the < operator to be defined on a custom type.

The table shows that you can implement almost all the operators as either a member of the custom type class or as a global function (with the exception of the four listed that have to be member methods). In general, it is best to implement the operator as part of the class because it maintains encapsulation: the member function has access to the non-public members of the class.

An example of a unary operator is the unary negative operator. This usually does not alter an object but returns a new object that is the *negative* of the object. For our point class, this means making both co-ordinates negative, which is equivalent to a mirror of the Cartesian point in a line *y* = -*x*:

```
// inline in point
point operator-() const
{
return point(-this->x, -this->y);
}
```

The operator is declared as const because it's clear the operator does not change the object and hence it's safe to be called on a const object. The operator can be called like this:

```
point p1(-1,1);
point p2 = -p1; // p2 is (1,-1)
```

To understand why we have implemented the operator like this, review what the unary operator would do when applied to a built-in type. The second statement here, int i, j=0; i = -j;, will only alter i and will not alter j, so the member operator- should not affect the value of the object.

The binary negative operator has a different meaning. First, it has two operands, and, second, in this example, the result is a different type to the operands because the result is a vector that indicates a direction by taking one point away from another. Assuming that the cartesian_vector is already defined with a constructor that has two parameters, then we can write:

```
cartesian_vector point::operator-(point& rhs) const
{
return cartesian_vector(this->x - rhs.x, this->y - rhs.y);
}
```

The increment and decrement operators have a special syntax because they are unary operators that can be prefixed or postfixed, and they alter the object they are applied to. The major difference between the two operators is that the postfixed operator returns the value of the object *before* the increment/decrement action, so a temporary has to be created. For this reason, the prefix operator almost always has better performance than the postfix operator. In a class definition, to distinguish between the two, the prefix operator has no parameters and the postfix operator has a dummy parameter (in the preceding table, 0 is given). For a class mytype, this is as follows:

```
class mytype
{
public:
```

```
mytype& operator++()
{
// do actual increment
return *this;
}
mytype operator++(int)
{
mytype tmp(*this);
operator++(); // call the prefix code
return tmp;
}
};
```

The actual increment code is implemented by the prefix operator, and this logic is used by the postfix operator through an explicit call to the method.

Defining function classes

A functor is a class that implements the () operator. This means that you can call an object using the same syntax as a function. Consider this:

```
class factor
{
double f = 1.0;
public:
factor(double d) : f(d) {}
double operator()(double x) const { return f * x; }
};
```

This code can be called like this:

```
factor threeTimes(3); // create the functor object
double ten = 10.0;
double d1 = threeTimes(ten); // calls operator(double)
double d2 = threeTimes(d1); // calls operator(double)
```

This code shows that the functor object not only provides some behavior (in this case, performing an action on the parameter) but it also can have a state. The preceding two lines are called through the operator() method on an object:

```
double d2 = threeTimes.operator()(d1);
```

Look at the syntax. The functor object is called as if it is a function declared like this:

```
double multiply_by_3(double d)
{
return 3 * d;
}
```

Imagine that you want to pass a pointer to a function--perhaps you are want the function's behavior to be altered by external code. To be able to use either a functor or a method pointer, you need to overload your function:

```
void print_value(double d, factor& fn);
void print_value(double d, double(*fn)(double));
```

The first takes a reference to a functor object. The second has a C-type function pointer (to which you can pass a pointer to multiply_by_3) and is quite unreadable. In both cases the fn parameter is called in the same way in the implementation code, but you need to declare two functions because they are different types. Now, consider the magic of function templates:

```
template<typename Fn>
void print_value(double d, Fn& fn)
{
double ret = fn(d);
cout << ret << endl;
}
```

This is generic code; the Fn type can be a C function pointer or a functor class, and the compiler will generate the appropriate code.

This code can be called by either passing a function pointer to a global function, which will have the __cdecl calling convention, or a functor object where the operator() operator will be called, which has a __thiscall calling convention.

This is a mere implementation detail, but it does mean that you can write a generic function that can take either a C-like function pointer or a functor object as a parameter. The C++ Standard Library uses this magic, which means that the algorithms it provides can be called either with a *global function* or a *functor*, or a *lambda expression*.

The Standard Library algorithms use three type of functional classes, generators, and unary and binary functions; that is, functions with zero, one or two parameters. In addition, the Standard Library calls a function object (unary or binary) that returns a bool **predicate**. The documentation will tell you if a predicate, unary, or binary function is needed. Older versions of the Standard Library needed to know the types of the return value and parameters (if any) of the function object to work, and, for this reason, functor classes had to be based upon the standard classes, unary_function and binary_function (through inheritance, explained in the next chapter). In C++11, this requirement has been removed, so there is no requirement to use these classes.
In some cases, you will want to use a binary functor when a unary functor is required. For example, the Standard Library defines the greater class that, when used as a function object, takes two parameters and a bool to determine whether the first parameter is greater than the second one, using the operator> defined by the type of both parameters. This will be used for functions that need a binary functor, and hence the function will compare two values; for example:

```
template<typename Fn>
int compare_vals(vector<double> d1, vector<double> d2, Fn compare)
{
if (d1.size() > d2.size()) return -1; // error
int c = 0;
for (size_t i = 0; i < d1.size(); ++i)
{
if (compare(d1[i], d2[i])) c++;
}
return c;
}
```

This takes two collections and compares corresponding items using the functor passed as the last parameter. It can be called like this:

```
vector<double> d1{ 1.0, 2.0, 3.0, 4.0 };
vector<double> d2{ 1.0, 1.0, 2.0, 5.0 };
int c = compare_vals(d1, d2, greater<double>());
```

The greater functor class is defined in the <functional> header and compares two numbers using the operator> defined for the type. What if you wanted to compare the items in a container with a fixed value; that is, when the operator()(double, double) method on the functor is called, one parameter always has a fixed value? One option is to define a stateful functor class (as shown previously) so that the fixed value is a member of the functor object. Another way to do this is to fill another vector with the fixed value and continue to compare two vectors (this can get quite expensive for large vectors).

Another way is to reuse the functor class, but to *bind* a value to one of its parameters. A version of the compare_vals function can be written like this, to take just one vector:

```
template<typename Fn>
int compare_vals(vector<double> d, Fn compare)
{
int c = 0;
for (size_t i = 0; i < d.size(); ++i)
{
if (compare(d[i]) c++;
}
return c;
}
```

The code is written to call the functor parameter on just one value because it is assumed that the functor object contains the other value to compare. This is carried out by binding the functor class to the parameter:

```
using namespace::std::placeholders;
int c = compare_vals(d1, bind(greater<double>(), _1, 2.0));
```

The bind function is variadic. The first parameter is the functor object and it is followed by the parameters that will be passed to the operator() method of the functor. The compare_vals function is passed a **binder** object that binds the functor to values. In the compare_vals function, the call to the functor in compare(d[i]) is actually a call to the operator() method of the binder object, and this method forwards the parameter d[i] and the bound value to the operator() method of the functor.
In the call to bind, if an actual value is provided (here, 2.0), then that value is passed to the functor at that position in the call to the functor (here, 2,0 is passed to the second parameter). If a symbol preceded by an underscore is used, then it is a **placeholder**. There are 20 such symbols (_1 to _20) defined in the std::placeholders namespace. The placeholder means "use the value passed in this position to the binder object operator() method call to the functor call operator() method indicated by the placeholder." Thus, the placeholder in this call means "pass the first parameter from invoking the binder and pass it to the first parameter of the greater functor operator()."
The previous code compares each item in the vector with 2.0 and will keep a count of those that are greater than 2.0. You could invoke it this way:

```
int c = compare(d1, bind(greater<double>(), 2.0, _1));
```

The parameter list is swapped, and this means that 2.0 is compared with each item in the vector and the function will keep a count of how many times 2.0 is greater than the item.
The bind function, and placeholders, are new to C++11. In prior versions you could use the bind1st and bind2nd functions to bind a value to either the first or second parameter of the functor.

Defining conversion operators

We have already seen that a constructor can be used to convert from another type to your custom type if your custom type has a constructor that takes the type you are converting. You can also perform the conversion in the other direction: converting the object into another type. To do this, you provide an operator without a return type with the name of the type to convert to. In this case, you need a space between the operator keyword and the name:

```
class mytype
{
int i;
public:
mytype(int i) : i(i) {}
explicit mytype(string s) : i(s.size()) {}
operator int () const { return i; }
};
```

This code can convert an int or a string to mytype; in the latter case, only through an explicit mention of the constructor. The last line allows you to convert an object back to an int:

```
string s = "hello";
mytype t = mytype(s); // explicit conversion
int i = t; // implicit conversion
```

You can make such conversion operators explicit so that they will be called only when an explicit cast is used. In many cases, you will want to leave off this keyword because implicit conversions are useful when you want to wrap a resource in a class and use the destructor to do automatic resource management for you.
Another example of using a conversion operator is returning values from a stateful functor. The idea here is that the operator() will perform some action and the result is maintained by the functor. The issue is how do you obtain this state of the functor, especially when they are often created as temporary objects? A conversion operator can provide this functionality.

For example, when you calculate an average, you do it in two stages: the first stage is to accumulate the values and then the second stage is to calculate the average by dividing it by the number of items. The following functor class does this with the division performed as part of the conversion to a double:

```
class averager
{
double total;
int count;
public:
averager() : total(0), count(0) {}
void operator()(double d) { total += d; count += 1; }
operator double() const
{
return (count != 0) ? (total / count) :
numeric_limits<double>::signaling_NaN();
}
};
```

This can be called like this:

```
vector<double> vals { 100.0, 20.0, 30.0 };
double avg = for_each(vals.begin(), vals.end(), averager());
```

The for_each function calls the functor for every item in the vector, and the operator() simply sums the items passed to it and maintains a count. The interesting part is that after the for_each function has iterated over all of the items in the vector it returns the functor, and so there is an implicit conversion to a double, which calls the conversion operator that calculates the average.

Managing resources

We have already seen one sort of resource that requires careful management: memory. You allocate memory with new, and when you have finished with the memory you must deallocate the memory with delete. A failure to deallocate the memory will cause a memory leak. Memory is, perhaps, the most fundamental of system resources, but most operating systems have many others: file handles, handles to graphic objects, synchronization objects, threads, and processes. Sometimes possession of such a resource is exclusive and will prevent other code from accessing the resource accessed through the resource. Thus, it is important that such resources are freed at some point, and, usually, that they are freed in a timely manner.

Classes help here with a mechanism called **Resource Acquisition Is Initialization** (RAII) invented by Bjarne Stroustrup, the author of C++. Put simply, the resource is allocated in the constructor of an object and freed in the destructor, so it means that the lifetime of the resource is the lifetime of the object. Typically, such wrapper objects are allocated on the stack, and this means that you are guaranteed that the resource will be freed when the object goes out of scope *regardless of how this happens*.

So, if objects are declared in the code block for a looping statement (while, for), then at the end of each loop the destructor for each will be called (in reverse order of creation) and the object will be created again when the loop is repeated. This occurs whether the loop is repeated because the end of the code block has been reached or if the loop is repeated through a call to continue. Another way to leave a code block is through a call to break, a goto, or if the code calls return to leave the function. If the code raises an exception (see Chapter 10, *Diagnostics and Debugging*), the destructor will be called as the object goes out of scope, so if the code is guarded by a try block, the destructor of objects declared in the block will be called before the catch clause is called. If there is no guard block, then the destructor will be called before the function stack is destroyed and the exception propagated.

Writing wrapper classes

There are several issues that you must address when writing a class to wrap a resource. The constructor will be used, either to obtain the resource using some library function (usually accessed through some kind of opaque handle) or will take the resource as a parameter. This resource is stored as a data member so other methods on the class can use it. The resource will be released in the destructor using whatever function your library provides to do this. This is the bare minimum. In addition, you have to think how the object will be used. Often such wrapper classes are most convenient if you can use instances as if they are the resource handle. This means that you maintain the same style of programming to access the resource, but you just don't have to worry too much about releasing the resource.

You should think about whether you want to be able convert between your wrapper class and the resource handle. If you do allow this, it means that you may have to think about cloning the resource, so that you do not have two copies of the handle--one that is managed by the class and the other copy that could be released by external code. You also need to think about whether you want to allow the object to be copied or assigned, and if so, then you will need to appropriately implement the copy constructor, a move constructor, and the copy and move assignment operators.

Using smart pointers

The C++ Standard Library provides several classes to wrap resources accessed through pointers. To prevent memory leaks, you have to ensure that memory allocated on the free store is freed at some point. The idea of a smart pointer is that you treat an instance as if it is the pointer, so you use the * operator to dereference to get access to the object it points to or use the -> operator to access a member of the wrapped object. The smart pointer class will manage the lifetime of the pointer it wraps and will release the resource appropriately.

The Standard Library has three smart pointer classes: unique_ptr, shared_ptr, and weak_ptr. Each handles how the resource is released in a different way, and how or whether you can copy a pointer.

Managing exclusive ownership

The unique_ptr class is constructed with a pointer to the object it will maintain. This class provides the operator * to give access to the object, dereferencing the wrapped pointer. It also provides the -> operator, so that if the pointer is for a class, you can access the members through the wrapped pointer.

The following allocates an object on the free store and manually maintains its lifetime:

```
void f1()
{
int* p = new int;
*p = 42;
cout << *p << endl;
delete p;
}
```

In this case, you get a pointer to the memory on the free store allocated for an int. To access the memory--either to write to it or read from it--you dereference the pointer with the * operator. When you are finished with the pointer, you must call delete to deallocate the memory and return it to the free store. Now consider the same code, but with a smart pointer:

```
void f2()
{
unique_ptr<int> p(new int);
*p = 42;
cout << *p << endl;
delete p.release();
}
```

The two main differences are that the smart pointer object is constructed explicitly by calling the constructor that takes a pointer of the type that is used as the template parameter. This pattern reinforces the idea that the resource should only be managed by the smart pointer.
The second change is that the memory is deallocated by calling the release method on the smart pointer object to take ownership of the wrapped pointer, so that we can delete the pointer explicitly.
Think of the release method releasing the pointer from the ownership of the smart pointer. After this call, the smart pointer no longer wraps the resource. The unique_ptr class also has a method get that will give access to the wrapped pointer, but the smart pointer object will still retain ownership; *do not delete the pointer obtained this way*!
Note that a unique_ptr object wraps a pointer, and just the pointer. This means that the object is the same size in memory as the pointer it wraps. So far, the smart pointer has added very little, so let's look at another way to deallocate the resource:
```
void f3()
{
unique_ptr<int> p(new int);
*p = 42;
cout << *p << endl;
p.reset();
}
```
This is *deterministic* releasing of the resource, and means that the resource is released just when you want it to happen, which is similar to the situation with the pointer. The code here is not releasing the resource itself; it is allowing the smart pointer to do it, using a **deleter**. The default deleter for unique_ptr is a functor class called default_delete, which calls the delete operator on the wrapped pointer. If you intend to use deterministic destruction, reset is the preferred method. You can provide your own deleter by passing the type of a custom functor class as the second parameter to the unique_ptr template:
```
template<typename T> struct my_deleter
{
void operator()(T* ptr)
{
cout << "deleted the object!" << endl;
delete ptr;
}
};
```

In your code, you will specify that you want the custom deleter, like this:
```
unique_ptr<int, my_deleter<int> > p(new int);
```
You may need to carry out an additional clean up before deleting the pointer, or the pointer could be a obtained by a mechanism other than new, so you can use a custom deleter to ensure that the appropriate releasing function is called. Note that the deleter is part of the smart pointer class, so if you have two different smart pointers using two different deleter this way, the smart pointer types are different even if they wrap the same type of resource.
When you use a custom deleter, the size of a unique_ptr object may be larger than the pointer wrapped. If the deleter is a functor object, each smart pointer object will need memory for this, but if you use a lambda expression, no more extra space will be required.
Of course, you are most likely to allow the smart pointer to manage the resource lifetime for you, and to do this you simply allow the smart pointer object to go out of scope:
```
void f4()
```

```
{
unique_ptr<int> p(new int);
*p = 42;
cout << *p << endl;
} // memory is deleted
```
Since the pointer created is a single object, it means that you can call the new operator on an appropriate constructor to pass in initialization parameters. The constructor of unique_ptr is passed a pointer to an already constructed object, and the class manages the lifetime of the object after that. Although a unique_ptr object can be created directly by calling its constructor, you cannot call the copy constructor, so you cannot use initialization syntax during construction. Instead, the Standard Library provides a function called make_unique. This has several overloads, and for this reason it is the preferred way to create smart pointers based on this class:
```
void f5()
{
unique_ptr<int> p = make_unique<int>();
*p = 42;
cout << *p << endl;
} // memory is deleted
```

This code will call the default constructor on the wrapped type (int), but you can provide parameters that will be passed to the appropriate constructor of the type. For example, for a struct that has a constructor with two parameters, the following may be used:
```
void f6()
{
unique_ptr<point> p = make_unique<point>(1.0, 1.0);
p->x = 42;
cout << p->x << "," << p->y << endl;
} // memory is deleted
```
The make_unique function calls the constructor that assigns the members with non-default values. The -> operator returns a pointer and the compiler will access the object members through this pointer.
There is also a specialization of unique_ptr and make_unique for arrays. The default deleter for this version of unique_ptr will call delete[] on the pointer, and thus it will delete every object in the array (and call each object's destructor). The class implements an indexer operator ([]) so you can access each item in the array. However, note that there are no range checks, so, like a built-in array variable, you can access beyond the end of the array. There are no dereferencing operators (* or ->), so a unique_ptr object based on an array can only be accessed with array syntax.
The make_unique function has an overload that allows you to pass the size of the array to create, but you have to initialize each object individually:
```
unique_ptr<point[]> points = make_unique<point[]>(4);
points[1].x = 10.0;
points[1].y = -10.0;
```
This creates an array with four point objects initially set to the default value, and the following lines initialize the second point to a value of (10.0, -10.0). It is almost always better to use vector or array than unique_ptr to manage arrays of objects.
Earlier versions of the C++ Standard Library had a smart pointer class called auto_ptr. This was a first attempt, and worked in most cases, but also had some limitations; for example, auto_ptr objects could not be stored in Standard Library containers. C++11 introduces rvalue references and other language features such as move semantics, and, through these, unique_ptr objects can be stored in containers. The auto_ptr class is still available through the <new> header, but only so that older code can still compile.

The important point about the unique_ptr class is that it ensures that there is a single copy of the pointer. This is important because the class destructor will release the resource, so if you *could* copy a unique_ptr object it would mean more than one destructor will attempt to release the resource. Objects of unique_ptr have *exclusive ownership*; an instance always owns what it points to.
You cannot copy assign unique_ptr smart pointers (the copy assignment operator and copy constructor are deleted), but you can *move* them by transferring ownership of the resource from the source pointer to the destination pointer. So, a function can return a unique_ptr because the ownership is transferred through move semantics to the variable being assigned to the value of the function. If the smart pointer is put into a container, there is another move.

Sharing ownership

There are occasions when you will need to share a pointer: you may create several objects and pass a pointer to a single object to each of them so they can call this object. Ordinarily, when an object has a pointer to another object, that pointer represents a resource that should be destroyed during the destruction of the containing object. If a pointer is shared, it means that when one of the objects deletes the pointer, the pointers in all of the other objects will be invalid (this is called a **dangling pointer** because it no longer points to an object).
You need a mechanism where several objects can hold a pointer that will remain valid until *all* the objects using that pointer have indicated they will no longer need to use it.
C++11 provides this facility with the shared_ptr class. This class maintains a **reference count** on the resource, and each copy of the shared_ptr for that resource will increment the reference count. When one instance of shared_ptr for that resource is destroyed, it will decrement the reference count. The reference count is shared, so it means that a non-zero value signifies that at least one shared_ptr exists accessing the resource. When the last shared_ptr object decrements the reference count to zero, it is safe to release the resource. This means that the reference count must be managed in an atomic way to handle multithreaded code.
Since the reference count is shared, it means that each shared_ptr object holds a pointer to a shared buffer called the **control block**, and this means it holds the raw pointer and a pointer to the control block, and so each shared_ptr object will hold more data than a unique_ptr. The control block is used for more than just the reference count.

A shared_ptr object can be created to use a custom deleter (passed as a constructor parameter), and the deleter is stored in the control block. This is important because it means that the custom deleter is not part of the type of the smart pointer, so several shared_ptr objects wrapping the same resource type but using different deleters are still the same type and can be put in a container for that type.
You can create a shared_ptr object from another shared_ptr object, and this will initialize the new object with the raw pointer and the pointer to the control block, *and* increment the reference count.
```
point* p = new point(1.0, 1.0);
shared_ptr<point> sp1(p); // Important, do not use p after this!
shared_ptr<point> sp2(sp1);
p = nullptr;
sp2->x = 2.0;
sp1->y = 2.0;
sp1.reset(); // get rid of one shared pointer
```
Here, the first shared pointer is created using a raw pointer. This is not the recommended way to use shared_ptr. The second shared pointer is created using the first smart pointer, so now there are two shared pointers to the same resource (p is assigned to nullptr to

prevent its further use). After this, either sp1 or sp2 can be used to access the *same* resource. At the end of this code, one shared pointer is reset to nullptr; this means that sp1 no longer has a reference count on the resource, and you cannot use it to access the resource. However, you can still use sp2 to access the resource until it goes out of scope, or you call reset.

In this code, the smart pointers were created from a separate raw pointer. Since the shared pointers now have taken over the lifetime management of the resource it is important to no longer use the raw pointer, and in this case it is assigned to nullptr. It is better to avoid the use of raw pointers, and the Standard Library enables this with a function called make_shared, which can be used like this:

```
shared_ptr<point> sp1 = make_shared<point>(1.0,1.0);
```

The function will create the specified object using a call to new, and since it takes a variable number of parameters, you can use it to call any constructor on the wrapped class.

You can create a shared_ptr object from a unique_ptr object, which means that the pointer is *moved* to the new object and the reference counting control block created. Since the resource will now be shared, it means that there is no longer exclusive ownership on the resource, so the pointer in the unique_ptr object will be made a nullptr. This means that you can have a factory function that returns a pointer to an object wrapped in a unique_ptr object, and the calling code can determine if it will use a unique_ptr object to get exclusive access to the resource or a shared_ptr object to share it.

There is little point in using shared_ptr for arrays of objects; there are much better ways to store collections of objects (vector or array). In any case, there is an indexing operator ([]) and the default deleter calls delete, not delete[].

Handling dangling pointers

Earlier in this book we made the point that, when you delete a resource, you should set the pointer to nullptr and you should check a pointer before using it to see if it is nullptr. This is so that you do not call a pointer to memory for an object that has been deleted: a dangling pointer.

There are situations when a dangling pointer can occur by design. For example, a *parent* object may create *child* objects that have a **back pointer** to the parent so that the child has access to the parent. (An example of this is a window that creates child controls; it is often useful for the child controls to have access to the parent window.) The problem with using a shared pointer in this situation is that the parent will have a reference count on each child control and each child control has a reference count on the parent, and this creates a circular dependency.

Another example is if you have a container of observer objects with the intention of being able to inform each of these observer objects when an event occurs by calling a method on each one. Maintaining this list can be complicated, particularly if an observer object can be deleted, and hence you have to provide a means to remove the object from the container (where there will be a shared_ptr reference count) before you can completely delete the object. It becomes easier if your code can simply add a pointer to the object to the container in a way that does not maintain a reference count, but allows you to check when the pointer is used if the pointer is dangling or points to an existing object.

Such a pointer is called a **weak pointer** and the C++11 Standard Library provides a class called weak_ptr. You cannot use a weak_ptr object directly and there is no dereference operator. Instead, you create a weak_ptr object from a shared_ptr object and, when you want to access the resource, you create a shared_ptr object from the weak_ptr object. This means that a weak_ptr object has the same raw pointer, and access to the same control block as the shared_ptr object, but it does not take part in reference counting.

Once created, the weak_ptr object will enable you to test whether the wrapper pointer is to an existing resource or to a resource that has been destroyed. There are two ways to do this: either call the member function expired or attempt to create a shared_ptr from the weak_ptr. If you are maintaining a collection of weak_ptr objects, you may decide to periodically iterate through the collection, call expired on each one, and if the method returns true, remove that object from the collection. Since the weak_ptr object has access to the control block created by the original shared_ptr object, it can test to see if the reference count is zero.

The second way to test to see if a weak_ptr object is dangling is to create a shared_ptr object from it. There are two options. You can create the shared_ptr object by passing the weak pointer to its constructor and if the pointer has expired, the constructor will throw a bad_weak_ptr exception. The other way is to call the lock method on the weak pointer and if the weak pointer has expired, then the shared_ptr object will be assigned to nullptr and you can test for this. These three ways are shown here:

```
shared_ptr<point> sp1 = make_shared<point>(1.0,1.0);
weak_ptr<point> wp(sp1);
// code that may call sp1.reset() or may not
if (!wp.expired()) { /* can use the resource */}
shared_ptr<point> sp2 = wp.lock();
if (sp2 != nullptr) { /* can use the resource */}
try
{
shared_ptr<point> sp3(wp);
// use the pointer
}
catch(bad_weak_ptr& e)
{
// dangling weak pointer
}
```

Since a weak pointer does not alter the reference count on a resource it means that you can use it for a back pointer to break the cyclic dependency (although, often it makes sense to use a raw pointer instead because a child object cannot exist without its parent object).

Templates

Classes can be templated, which means that you can write generic code and the compiler will generate a class with the types that your code uses. The parameters can be types, constant integer values, or variadic versions (zero or more parameters, as provided by the code using the class). For example:

```
template <int N, typename T> class simple_array
{
T data[N];
public:
const T* begin() const { return data; }
const T* end() const { return data + N; }
int size() const { return N; }
T& operator[](int idx)
{
if (idx < 0 || idx >= N)
throw range_error("Range 0 to " + to_string(N));
return data[idx];
}
};
```

Here is a very simple array class that defines the basic iterator functions and the indexing operator, so that you can call it like this:

```
simple_array<4, int> four;
four[0] = 10; four[1] = 20; four[2] = 30; four[3] = 40;
```

```
for(int i : four) cout << i << " "; // 10 20 30 40
cout << endl;
four[4] = -99; // throws a range_error exception
```

If you choose to define a function out of the class declaration, then you need to give the template and its parameters as part of the class name:

```
template<int N, typename T>
T& simple_array<N,T>::operator[](int idx)
{
if (idx < 0 || idx >= N)
throw range_error("Range 0 to " + to_string(N));
return data[idx];
}
```

You can also have default values for template parameters:

```
template<int N, typename T=int> class simple_array
{
// same as before
};
```

If you think you should have a specific implementation for a template parameter, then you can provide the code for that version as a specialization of the template:

```
template<int N> class simple_array<N, char>
{
char data[N];
public:
simple_array<N, char>(const char* str)
{
strncpy(data, str, N);
}
int size() const { return N; }
char& operator[](int idx)
{
if (idx < 0 || idx >= N)
throw range_error("Range 0 to " + to_string(N));
return data[idx];
}
operator const char*() const { return data; }
};
```

Note that, with a specialization, you do not get any code from the fully templated class; you have to implement all the methods you want to provide, and, as illustrated here, methods that are relevant to the specialization but not available on the fully templated class. This example is a **partial specialization**, meaning that it is specialized on just one parameter (T, the type of the data). This class will be used for declared variables of the type simple_array<n, char>, where n is an integer. You are free to have a fully specialized template, which, in this case, will be a specialization for a fixed size and a specified type:

```
template<> class simple_array<256, char>
{
char data[256];
public:
// etc
};
```

It is probably not useful in this case, but the idea is that there will be special code for variables that need 256 chars.

Using classes

The **Resource Acquisition Is Initialization** technique is useful for managing resources provided by other libraries, such as the C Runtime Library or the Windows SDK. It

simplifies your code because you do not have to think about where a resource handle will go out of scope and provide clean-up code at every point. If the clean-up code is complicated, it is typical in C code to see it put at the end of a function and every exit point in the function will have a goto jump to that code. This results in messy code. In this example, we will wrap the C files functions with a class, so that the lifetime of the file handle is maintained automatically.
The C runtime _findfirst and _findnext functions allow you to search for a file or directory that matches a pattern (including wildcard symbols). The _findfirst function returns an intptr_t, which is relevant to just that search and this is passed to the _findnext function to get subsequent values. This intptr_t is an opaque pointer to resources that the C Runtime maintains for the search, and so when you are finished with the search you must call _findclose to clean up any resources associated with it. To prevent memory leaks, it is important to call _findclose.

Under the Beginning_C++ folder, create a folder called Chapter_06. In Visual C++, create a new C++ source file, save it to the Chapter_06 folder, and call it search.cpp. The application will use the Standard Library console and strings, and it will use the C Runtime file functions, so add these lines to the top of the file:

```
#include <iostream>
#include <string>
#include <io.h>
using namespace std;
```

The application will be called with a file search pattern and it will use the C functions to search for files, so you will need a main function that has parameters. Add the following to the bottom of the file:

```
void usage()
{
cout << "usage: search pattern" << endl;
cout << "pattern is the file or folder to search for "
<< "with or without wildcards * and ?" << endl;
}
int main(int argc, char* argv[])
{
if (argc < 2)
{
usage();
return 1;
}
}
```

The first thing is to create a wrapper class for the search handle that will manage this resource. Above the usage function, add a class called search_handle:

```
class search_handle
{
intptr_t handle;
public:
search_handle() : handle(-1) {}
search_handle(intptr_t p) : handle(p) {}
void operator=(intptr_t p) { handle = p; }
void close()
{ if (handle != -1) _findclose(handle); handle = 0; }
~search_handle() { close(); }
};
```

This class has a separate function to release the handle. This is so that a user of this class can release the wrapper resource as soon as possible. If the object is used in code that could

throw an exception, the close method won't be called directly, but the destructor will be called instead. The wrapper object can be created with a intptr_t value. If this value is -1, then the handle is invalid, so the close method will only call _findclose if the handle does not have this value.
We want objects of this class to have exclusive ownership of the handle, so delete the copy constructor and copy assignment by putting the following in the public part of the class:

```
void operator=(intptr_t p) { handle = p; }
search_handle(search_handle& h) = delete;
void operator=(search_handle& h) = delete;
```

If an object is moved, then any handle in the existing object must be released, so add the following after the lines you just added:

```
search_handle(search_handle&& h) { close(); handle = h.handle; }
void operator=(search_handle&& h) { close(); handle = h.handle; }
```

The wrapper class will be allocated by a call to _findfirst and will be passed to a call to _findnext, so the wrapper class needs two operators: one to convert to an intptr_t, so objects of this class can be used wherever an intptr_t is needed, and the other so that object can be used when a bool is needed. Add these to the public part of the class:

```
operator bool() const { return (handle != -1); }
operator intptr_t() const { return handle; }
```

The conversion to bool allows you to write code like this:

```
search_handle handle = /* initialize it */;
if (!handle) { /* handle is invalid */ }
```

If you have a conversion operator that returns a pointer, then the compiler will call this in preference to the conversion to bool.
You should be able to compile this code (remember to use the /EHsc switch) to confirm that there are no typos.

Next, write a wrapper class to perform the search. Below the search_handle class, add a file_search class:

```
class file_search
{
search_handle handle;
string search;
public:
file_search(const char* str) : search(str) {}
file_search(const string& str) : search(str) {}
};
```

This class is created with the search criteria, and we have the option of passing a C or C++ string. The class has a search_handle data member, and, since the default destructor will call the destructor of member objects, we do not need to provide a destructor ourselves. However, we will add a close method so that a user can explicitly release resources. Furthermore, so that users of the class can determine the search path, we need an accessor. At the bottom of the class, add the following:

```
const char* path() const { return search.c_str(); }
void close() { handle.close(); }
```

We do not want instances of the file_search object to be copied because that would mean two copies of the search handle. You could delete the copy constructor and assignment operator, but there is no need. Try this: in the main function, add this test code (it does not matter where):

```
file_search f1("");
file_search f2 = f1;
```

Compile the code. You'll get an error and an explanation:

```
error C2280: 'file_search::file_search(file_search &)': attempting to
reference a deleted function
note: compiler has generated 'file_search::file_search' here
```

Without a copy constructor, the compiler will generate one (this is the second line). The first line is a bit odd because it is saying that you are trying to call a deleted method that the compiler has generated! In fact, the error is saying that the generated copy constructor is attempting to copy the handle data member and the search_handle copy constructor that has been deleted. Thus you are protected against copying file_search objects without adding any other code. Delete the test lines you just added.

Next add the following lines to the bottom of the main function. This will create a file_search object and print out information to the console.

```
file_search files(argv[1]);
cout << "searching for " << files.path() << endl;
```

Then you need to add code to perform the search. The pattern used here will be a method that has an out parameter and returns a bool. If a call to the method succeeds, then the file found will be returned in the out parameter and the method will return true. If the call fails, then the out parameter is left untouched and the method returns false. In the public section of the file_search class, add this function:

```
bool next(string& ret)
{
_finddata_t find{};
if (!handle)
{
handle = _findfirst(search.c_str(), &find);
if (!handle) return false;
}
else
{
if (-1 == _findnext(handle, &find)) return false;
}
ret = find.name;
return true;
}
```

If this is the first call to this method, then handle will be invalid and so _findfirst is called. This will fill a _finddata_t structure with the results of the search and return an intptr_t value. The search_handle object data member is assigned to this value returned from this function, and if _findfirst returns -1, the method returns false. If the call is successful, then the out parameter (a reference to a string) is initialized using a C string pointer in the _finddata_t structure.

If there are more files that match the pattern, then you can call the next function repeatedly, and on these subsequent calls the _findnext function is called to get the next file. In this case the search_handle object is passed to the function and there is an implicit conversion to intptr_t through the class's conversion operator. If the _findnext function returns -1, it means there are no more files in the search.

At the bottom of the main function, add the following lines to perform the search:

```
string file;
while (files.next(file))
{
cout << file << endl;
}
```

Now you can compile the code and run it with a search criterion. Bear in mind that this is constrained by the facilities of the _findfirst/_findnext functions, so the searches you can do will be quite simple. Try running this at the command line with a parameter to search for the subfolders in the Beginning_C++ folder:

search Beginning_C++Ch*

This will give a list of the subfolders starting with Ch. Since there is no reason for search_handle to be a separate class, move the entire class to the private section of the search_handle, above the declaration of the handle data member. Compile and run the code

So far, you have seen how to modularize code in functions and encapsulate data with code in a class. You have also seen how to write generic code with templates. Classes and encapsulation allow you to combine together code and data as an object. In this chapter, you'll learn how to *reuse* code through inheritance and composition and how to use class inheritance to write object-orientated code.

Inheritance and composition

The classes you have seen so far are complete classes: you can create an instance of the class on the free store or the stack. You can do this because the data members of the class have been defined and so it is possible to calculate how much memory is needed for the object, and you have provided the full functionality of the class. These are called **concrete classes**.

If you have a routine in a class that proves useful and you want to reuse in a new class, you have a few choices. The first is called **composition**. With composition you add an instance of your utility class as a data member of the classes that will use the routine. A simple example is the string class--this provides all the functionality that you want from a string. It will allocate memory according to how many characters have to be stored and deallocate the memory it uses when the string object is destroyed. Your class uses the functionality of a string, but it is not a string itself, hence it has the string as a data member.

The second option is to use **inheritance**. There are many ways to use inheritance, and this chapter will mention some of them. In basic terms, inheritance is when one class *extends* another class the class being extended is called the **base class**, **parent class**, or **superclass**, and the class doing the extending is called a **derived class**, **child class**, or **subclass**.

However, there is an important concept to understand with inheritance: the relationship of the derived class to the base class. It is commonly given in terms of **is-a**. If the derived class is a type of base class, then the relationship is inheritance. An mp3 file is an operating system file, so if you have a os_file class, then you could legitimately derive from it to create an mp3_file class.

The derived class has the functionality and state of the base class (although it may not have complete access to them, as will be explained later), so it can use the functionality of the base class. In this case, it is similar to composition. However, there are significant differences. In general, in composition, the composed object is used by the class and not exposed directly to the client of the class. With inheritance, an object of the derived class is an object of the base class, so usually the client code will see the base class functionality. However, a derived class can hide the functionality of the base class, so client code will not see the hidden base class member, and the derived class can override the base class methods and provide its own version.

There is a lot of disagreement in the C++ community over whether you should use inheritance or composition to reuse code, and there are advantages and disadvantages of each. Neither is perfect and often a compromise is needed.

Inheriting from a class

Consider a class that wraps an operating system. This will provide lots of methods to give access to things such as the creation date, modification date, and the size of the file obtained by calling operating system functions. It could also provide methods to open the file, close the file, map the file into memory, and other useful things. Here are a few such members:

```
class os_file
```

```
{
const string file_name;
int file_handle;
// other data members
public:
long get_size_in_bytes();
// other methods
};
```

An mp3 file is an operating system file, but there are other operating system functions to access its data. We could decide to create an mp3_file class that derives from an os_file so that it has the functionality of the operating system file and extend this with the functionality of an mp3 file:

```
class mp3_file : public os_file
{
long length_in_secs;
// other data members
public:
long get_length_in_seconds();
// other methods
};
```

The first line of the mp3_file class indicates that it uses *public inheritance* (we will explain what public inheritance means later, but it is worth pointing out that this is the most common way to derive from a class). The derived class inherits the data members and the methods, and users of the derived class can use the members of the base class through the derived class, subject to the access specifiers. In this example, if some code has an mp3_file object, it can call the get_length_in_seconds method from the mp3_file class, and it can also call the get_size_in_bytes method from the base because this method is public.

The base class methods will most likely access the base class data members, and this illustrates an important point: the derived object contains the base class data members. Conceptually, in memory, you can think of the derived object as being the base class object data members with the extra data members defined in the derived object. That is, the derived object is an extended version of the base class object. This is illustrated in the following diagram:

In memory, an os_file object has two data members, file_name and file_handle, and an mp3_file object has these two data members and an additional data member, length_in_secs.

The encapsulation principle is important in C++. Although an mp3_file object contains the file_name and file_handle data members, they should only be changed by the base class methods. In this code, this is enforced by making them private to the os_file class.

When a derived object is created, the base object must be created first (with an appropriate constructor), similarly, when a derived object is destroyed, the derived part of the object is destroyed first (through the destructor of the derived class) before the base class destructor is called. Consider the following code snippet, using the members discussed in preceding text:

```
class os_file
{
public:
os_file(const string& name)
: file_name(name), file_handle(open_file(name))
{}
~os_file() { close_file(file_handle); }
};
```

```
class mp3_file : public os_file
{
public:
mp3_file(const string& name) : os_file(name) {}
~mp3_file() { /* clean up mp3 stuff*/ }
};
```

The open_file and close_file functions will be some operating system functions to open and close operating system files.

The derived class no longer has to perform the action of closing the file because the base class destructor, ~os_file, is automatically called after the derived class destructor is called. The mp3_file constructor calls the base class constructor through its constructor member list. If you do not explicitly call a base class constructor, then the compiler will call the default constructor of the base class as the first action of the derived class constructor. If the member list initializes data members, these will be initialized after any base class constructor is called.

Overriding methods and hiding names

The derived class inherits the functionality of the base class (subject to the access level of the methods), so a base class method can be called through an object of the derived class. The derived class can implement a method with the same prototype as the base class method, in which case the base class method is *overridden* by the derived class method and the derived class provides the functionality. A derived class will often override a base class method to provide functionality that is specific to the derived class; however, it can call the base class method by calling the method using the name resolution operator:

```
struct base
{
void f(){ /* do something */ }
void g(){ /* do something */ }
};
struct derived : base
{
void f()
{
base::f();
// do more stuff
}
};
```

Remember that a struct is a class type where members are public by default, and inheritance is public by default.

Here, the base::f and base::g methods will perform some action available to users of instances of this class. The derived class inherits both methods, and since it does not implement the method g when instances of the derived class call the g method, they will actually call the base::g method. The derived class implements its own version of the f method, so when an instance of the derived class calls the f method, they will call derived::f and not the base class version. In this implementation, we have decided that we need some of the functionality of the base class version, so derived::f explicitly calls the base::f method:

```
derived d;
d.f(); // calls derived::f
d.g(); // calls base::g
```

In the previous example, the method calls the base class version first before providing its own implementation. There is no specific convention here. Class libraries are sometimes implemented specifically for you to derive from a base class and use the class library code.

The documentation of the class library will say whether you are expected to replace the base class implementation or if you are expected to add to the base class implementation, and if so, whether you will call the base class method before or after your code.
In this example, the derived class provides a method with the exact prototype as the method on the base class to override it. In fact, adding any method with the same name as a method in the base class hides that base class method from the client code that uses the derived instance. So, consider that the derived class is implemented like this:

```
struct derived : base
{
void f(int i)
{
base::f();
// do more stuff with i
}
};
```

In this case, the base::f method is hidden from the code that creates a derived object, even though the method has a different prototype:

```
derived d;
d.f(42); // OK
d.f(); // won't compile, derived::f(int) hides base::f
```

The base class method with the same name is hidden, so the last line will not compile. You can, however, call the function explicitly by providing the base class name:

```
derived d;
d.derived::f(42); // same call as above
d.base::f(); // call base class method
derived *p = &d; // get an object pointer
p->base::f(); // call base class method
delete p;
```

At first sight, this syntax looks a little odd but once you know that the . and -> operators give access to a member, and the symbol after the operator is the name of the member, in this case, explicitly specified using the class name and scope resolution operator.
In general, the code shown so far is called **implementation inheritance**, where a class inherits the implementation from a base class.

Using pointers and references

In C++, you can get a pointer to where an object (a built-in type or a custom type) resides in memory using the & operator. The pointer is typed, so the code using the pointer assumes that the pointer points to the memory layout of the object of the type. Similarly, you can obtain a reference to an object, and the reference is an *alias* for the object, that is, operations on the reference occur on the object. A pointer (or a reference) to an instance of a derived class can be implicitly converted to a pointer (or a reference) to a base class object. This means that you can write a function that acts upon base class objects, using the behavior of the base class objects, and as long as the parameter is a pointer or reference to a base class, you can pass any derived class object to the function. The function does not know about, nor does it care about, the derived class functionality.
You should think about the derived object as being a base class object and accept that it can be used as a base class object. Clearly, a base class pointer will only have access to the members on the base class:
If the derived class hides a member of the base class, it means that a pointer to the derived class will call the derived version through the member name, but the base class pointer will only see the base class member, not the derived version.
If you have a base class pointer, you can cast it to a derived class pointer using static_cast:

```
// bad code
```

```
void print_y(base *pb)
{
// be wary of this
derived *pd = static_cast<derived*>(pb);
cout << "y = " << pd->y << endl;
}
void f()
{
derived d;
print_y(&d); // implicit cast to base*
```

```
}
```

The problem here is how can the print_y function guarantee that the base class pointer passed as the parameter to the specific derived object? It cannot, without discipline from the developers using the function guaranteeing that they will never pass a derived class pointer of a different type. The static_cast operator will return a pointer to a derived object even if the memory does not contain that object. There is a mechanism to perform type checking on the pointer being cast, which we will cover later in this chapter.

Access levels

So far, we have seen two access specifiers for class members: public and private. Members declared in the public section can be accessed by code in the class *and* by code outside the class either on an object or if the member is static, using the class name. Members declared in the private section can only be accessed by other members in the same class. A derived class can access the private members of the base class but not the private members. There is a third type of member access: protected. Members declared in the protected section can be accessed by methods in the same class or by methods in any derived class and by friends, but not by external code:

```
class base
{
protected:
void test();
};
class derived : public base
{
public:
void f() { test(); }
};
```

In this code, the test method can be called by members in the derived class but not by code outside of the class:

```
base b;
b.test(); // won't compile
derived d;
d.f(); // OK
d.test(); // won't compile
```

If you are writing a base class that you intend only ever to be used as a base class (client code should not create instances of it), then it makes sense to make the destructor protected:

```
class base
{
public:
// methods available through the derived object
protected:
~base(){}
};
```

The compiler will not allow you to create objects of this class on the free store and then destroy with delete, because this operator will call the destructor. Similarly, the compiler won't allow you to create objects on the stack because the compiler will call the inaccessible destructor when the object goes out of scope. This destructor will be called through the destructor of the derived class, so you can be assured that a correct cleanup of the base class will occur. This pattern does mean that you always only intend to use pointers to the derived classes to destroy the object with a call to the delete operator.

Changing access level through inheritance

When you override a method in the derived class, the access to the method is defined by the derived class. So if the base class method is protected or public, the access can be changed by the derived class:

```
class base
{
protected:
void f();
public:
void g();
};
class derived : public base
{
public:
void f();
protected:
void g();
};
```

In the preceding example, the base::f method is protected, so only the derived class can access it. The derived class overrides this method (and can call the base class method if the fully qualified name is used) and makes it public. Similarly, the base::g method is public but the derived class overrides this method and makes it protected (and if desired, it could make the method private).

You can also expose a protected base class from a derived class as a public member with a using statement:

```
class base
{
protected:
void f(){ /* code */};
};
class derived: public base
{
public:
using base::f;
};
```

Now, the derived::f method is public without the derived class creating a new method. A better use of this facility is to make a method private so that it is not available to derived classes (or if it was public, through an instance), or make it protected so that external code cannot access the member:

```
class base
{
public:
void f();
};
class derived: public base
{
protected:
using base::f;
};
```

The preceding code can be used like this:

```
base b;
b.f(); // OK
derived d;
d.f(); // won't compile
```

The last line won't compile because the f method is protected. If the intention is to make the method available only in the derived class and not to in any classes that may derive from it, you can use the using statement in the private section of the derived class; this is similar to deleting a base class method:

```
class derived: public base
{
public:
void f() = delete;
void g()
{
base::f(); // call the base class method
}
};
```

The f method cannot be used through the derived class, but the class can call the base class method.

Inheritance access levels

Earlier, you saw that to derive from a class, you provide the base class name and give an inheritance access specifier; the examples so far have used public inheritance, but you can use protected or private inheritance.

This is another difference between class and struct. For a class, if you miss off the inheritance access specifier, the compiler will assume that it is private; for a struct, if you miss off the inheritance access specifier, the compiler will assume that it is public.

The inheritance specifier applies more access restrictions, it will not relax them. The access specifier does not determine the access it has to the base class members, instead it alters the accessibility of those members through the derived class (that is through an instance of the class, or if another class derives from it). If a base class has private members, and a class inherits using public inheritance, the derived class still cannot access the private members; it only has access to public and protected members and objects of the derived class can only access the public members, and a class deriving from this class will only have access to the public and protected members.

If a derived class derives through the *protected inheritance*, it still has the same access to the base class as public and protected members, but the base class public and protected members will now be treated as protected through the derived class, so they can be accessed by a further derived class but are not accessible through an instance. If a class derives through private inheritance then all base class members become private in the derived class; so, although the derived class can access public and protected members, classes that derive from it cannot access any of the base class members.

One way of looking at protected inheritance is if the derived class had a using statement for each of the public members of the base class in the protected part of the class.

Similarly, private inheritance is as if you have deleted each of the public and protected methods of the base class.

In general, most inheritance will be through *public inheritance*. However, *private inheritance* has a use when you want to access some functionality from a base class but do not want its functionality to be available to classes that derive from your class. This is a little like composition, where you are using functionality but do not want that functionality directly exposed.

Multiple inheritance

C++ allows you to inherit from more than one base class. This is a powerful facility when used with interfaces, as we will discover later in this chapter. It can be useful for implementation inheritance, but it can cause some problems. The syntax is simple: you provide a list of classes to inherit from:

```
class base1 { public: void a(); };
class base2 { public: void b(); };
class derived : public base1, public base2
{
public:
// gets a and b
};
```

One way to use multiple inheritances is to build up libraries of classes each providing some functionality, or services. To get these services in your class you can add the class from the library to your base class list. Such a *building block* approach to creating classes through implementation inheritance has issues, as we will see later, and often a better approach is to use composition.

It is important when you consider multiple inheritances that you carefully review that you need the services via inheritance or whether composition is more appropriate. If a class provides a member that you do not want to be used by instances and you decide that you need to delete it, it is a good sign that you should consider composition.

If both classes have a member with the same name, then there is a potential problem. The most obvious case is if the base classes have a data member with the same name:

```
class base1 { public: int x = 1; };
class base2 { public: int x = 2; };
class derived : public base1, public base2 {};
```

In the previous example, both base classes have a data member called x. The derived class inherits from both classes, so does this mean that it gets just one data member called x? No. If it did, then this would mean that the base1 class would be able to alter a data member in the base2 class without knowing that it is affecting another class, and similarly the base2 class will find its data member being altered by the base1 class even though that class is not a friend. Consequently, when you derive from two classes with data members that have the same name, the derived class gets both data members.

This yet again illustrates the importance of maintaining encapsulation. Such data members should be private and only changed by the base class.

The derived class (and the code that uses instances, if the data members are accessible) can distinguish between them using their full names:

```
derived d;
cout << d.base1::x << endl; // the base1 version
cout << d.base2::x << endl; // the base2 version
```

The class can be summed up with the following diagram, illustrating the memory occupied by the three classes: base1, base2, and derived:

If you maintain encapsulation and make data members private and give access only through accessor methods, then derived classes will not have direct access to the data members and will not see this issue. However, the same issue occurs with methods, but the problem occurs even if the methods have different prototypes:

```
class base1 { public: void a(int); };
class base2 { public: void a(); };
class derived : public base1, public base2 {};
```

In this case, the two base classes have a method with the same name, a, but with a different prototype. This causes a problem when using the derived class, even when it may be obvious by the parameters what method should be called:

```
derived d;
d.a(); // should be a from base2, compiler still complains
```

This code will not compile, and the compiler will complain that the method call is ambiguous. Again, the solution to this problem is simple, you just need to specify which base class method to use:

```
derived d;
d.base1::a(42); // the base1 version
d.base2::a(); // the base2 version
```

Multiple inheritances can get even more complicated. The problem occurs if you have two classes that derive from the same base class and then you create another class that derives from both. Does the new class get two copies of the topmost base class members--one through each of its immediate base classes?

At the first level of inheritance, each of the classes (base1 and base2) inherit the data member from the ultimate base class (here, the data members are both called base::x to illustrate that they are inherited from the ultimate base class, base). The most derived class, derived, inherits *two* data members, so which one is base::x? The answer is that only one of them is, base1::x is base::x, because it is the first in the inheritance list. When the base methods change it, the change will be seen in base1 through base1::x. The base2::x member is a separate data member and not affected when base changes base::x. This is perhaps an unexpected result: the most-derived class inherits x from both of its parent classes.

This may not be the behavior you want. This issue is often called *diamond inheritance issue* and it should be apparent from the preceding diagram, where this name comes from. The solution is straightforward, and will be covered later in this chapter.

Object slicing

Earlier in the chapter, you saw that if you use a base class pointer to a derived object only the base class members can be safely accessed. The other members are still there, but they can only be accessed through an appropriate derived class pointer.

However, if you cast a derived class object to a base class object, something else happens: you create a new object, and that object is the base class object, just the base class object. The variable that you have cast to will only have the memory for the base class object, so the result is only the base class object part of the derived object:

```
struct base { /*members*/ };
struct derived : base { /*members*/ };
derived d;
base b1 = d; // slicing through the copy constructor
base b2;
b2 = d; // slicing through assignment
```

Here, the objects b1 and b2 have been created by *slicing off* the extra data in the derived class object d. This code looks a bit perverse, and you are not likely to write it, but the situation is likely to happen if you pass an object by value to a function:

```
void f(base b)
{
// can only access the base members
}
```

If you pass a derived object to this function, the base copy constructor will be called to create a new object, slicing off the derived class data members. In most cases, you do not want this behavior. This issue also has an unexpected behavior if your base class has virtual methods and expects the polymorphic functionality that virtual methods offer (virtual methods are covered later in this chapter). It is almost always a better idea to pass objects by reference.

Introducing polymorphism

Polymorphism comes from the Greek for *many shapes*. So far, you have a basic form of polymorphism. If you use a base class pointer to an object, then you can access the base class behavior, and if you have a derived class pointer, you get the derived class behavior. This is not as trivial as it appears because the derived class can implement its own version of the base class methods, so you can have a different implementation of that behavior.
You can have more than one class derived from a base class:

```
class base { /*members*/ };
class derived1 : public base { /*members*/ };
class derived2 : public base { /*members*/ };
class derived3 : public base { /*members*/ };
```

Since C++ is strongly typed, it means that a pointer to one derived class cannot be used to point to another derived class. So you cannot use a derived1* pointer to access an instance of derived2, it can only point to an object of type derived1. Even if the classes have the same members, they are still different types and their pointers are different. However, all derived classes have something in common, which is the base class. A derived class pointer can be implicitly converted to a base class pointer, so a base* pointer can point to an instance of base, derived1, derived2, or derived3. This means that a generic function taking a base* pointer as a parameter can be passed a pointer to any of these classes. This is the basis of interfaces, as we will see later.
The polymorphic aspect is that through pointers (or references), an instance of a class can be treated as an instance of any of the classes in its inheritance hierarchy.

Virtual methods

A base class pointer or reference giving access to just the base class functionality, and makes sense, but it is restrictive. If you have a car class that provides the interface for a car, a gas pedal, and brake to alter the speed, a steering wheel and reverse gear to alter the directionyou can derive from this class various other car types: a sports car, an SUV, or a family sedan. When you press the gas pedal, you expect the car to have the torque of an SUV, if your car is an SUV, or the speed of a sports car if it's a sports car. Similarly, if you call the accelerate method on a car pointer and that pointer is to a suv, then you expect to get method to reflect the torque of the SUV, and if the car pointer points to a sportscar object, you performance acceleration. Earlier, we said that if you access a derived class instance through a base class pointer, then you will get the implementation of the base class methods. This means that, calling the accelerate method on a car pointer that points to an suv or a sportscar object, you will still get the implementation of car::accelerate and not suv::accelerate or sportscar::accelerate which you would want.
This behavior of calling the derived method through a base class pointer is known as **method dispatching**. The code calling a method through a base class pointer does not know the type of object that the pointer points to, but it still gets the functionality of that object because the method on that object is called. This method dispatching is not applied by default because it involves a little extra cost both in memory and performance.
Methods that can take part in method dispatching are marked with the keyword virtual in the base class, and hence are usually called **virtual methods**. When you call such a method through a base class pointer, the compiler ensures that the method on the actual object's class is called. Since every method has a this pointer as a hidden parameter, the method dispatching mechanism must ensure that the this pointer is appropriate when the method is called. Consider the following example:

```
struct base
```

```
{
void who() { cout << "base "; }
};
struct derived1 : base
{
void who() { cout << "derived1 "; }
};
struct derived2 : base
{
void who() { cout << "derived2 "; }
};
struct derived3 : derived2
{
void who() { cout << "derived3 "; }
```

```
};
void who_is_it(base& r)
{
p.who();
}
int main()
{
derived1 d1;
who_is_it(d1);
derived2 d2;
who_is_it(d2);
derived3 d3;
who_is_it(d3);
cout << endl;
return 0;
}
```
There is a base class and two child classes, derived1 and derived2. There is a further level of inheritance through derived2 to a class called derived3. The base class implements a method called who that prints the class name. This method is implemented appropriately on each of the derived classes so that when this method is called on an object of derived3, the method will print derived3 in the console. The main function creates an instance of each of the derived classes and passes each one by reference to a function called who_is_it that calls the who method. This function has a parameter that is a reference to base, and since this is the base class of all of the classes (for derived3, its immediate base class is derived2). When you run this code, the result will be as follows:
```
base base base
```
This output comes from the three calls to the who_is_it function, passing objects that are instances of the derived1, derived2, and derived3 classes. Since the parameter is a reference to base, it means that the base::who method is called.
Making one simple change will alter this behavior completely:
```
struct base
{
virtual void who() { cout << "base "; }
};
```
All that has changed is the addition of the virtual keyword to the who method in the base class, but the result is significant. When you run this code, the result will be as follows:
```
derived1 derived2 derived3
```

You have not changed the who_is_it function, nor the methods on the derived classes, yet the output of who_is_it is very different compared to what it was earlier. The who_is_it function calls the who method through a reference, but now, rather than calling the

base::who method, the who method on the actual object that the reference aliases is called. The who_is_it function has done nothing additional to make sure that the derived class function is called--it is *exactly* the same as earlier.
The derived3 class is not derived directly from base, instead, it is derived from derived2, which is itself a child class of base. Even so, the method dispatching works on instances of the derived3 class. This illustrates that however far up the inheritance chain virtual is applied, the method dispatching will still work on the inherited method of the derived class.
It is important to point out that the method dispatching is applied *only* to the methods that virtual has been applied to in the base class. Any other methods in the base class not marked with virtual will be called without method dispatching. A derived class will inherit a virtual method and get the method dispatching automatically, it does not have to use the virtual keyword on any methods it overrides, but it is a useful visual indication as to how the method can be called.
With the derived classes implementing virtual methods, you can use a single container to hold pointers to instances of all such classes and invoke their virtual methods without the invocation code knowing the type of the object:

```
derived1 d1;
derived2 d2;
derived3 d3;
base *arr[] = { &d1, &d2, &d3 };
for (auto p : arr) p->who();
cout << endl;
```

Here, the arr built-in array holds pointers to objects of the three types and the ranged for loop iterates through the array and calls the method virtually. This gives the expected result:

```
derived1 derived2 derived3
```

There are three important points about the preceding code:
It is important that a built-in array is used here; there are issues with the Standard Library containers like vector.
It is important that the array holds pointers, not objects. If you have an array of base objects, they will be initialized by slicing the derived objects.
It is also important that the address of a stack object is used. This is because there are issues with destructors.
These three issues are covered in later sections.
For a virtual method to be called using method dispatching, the derived class method must match the same signature as the base class' virtual method in terms of the name, parameters, and return type. If any of these are different (for example, different parameters), then the compiler will think that the derived method is a new function, and so when you call the virtual method through the base pointer, you'll get the base method. This is a rather insidious error because the code will compile, but you will get the wrong behavior.
The one exception to the last paragraph is if two methods differ by return types that are **covariant**, that is, one type can be converted to the other.

Virtual method tables

The behavior of method dispatching via virtual methods is all you need to know, but it is helpful to see how the C++ compiler implements method dispatching because it highlights the overhead of the virtual methods.
When the compiler sees a virtual method on a class, it will create a method pointer table, called the **vtable**, and put a pointer to each of the virtual methods in the class in the table. There will be a single copy of the vtable for the class. The compiler will also add a pointer

to this table, called the **vptr**, in every instance of the class. So, when you mark a method as virtual, there will be a single memory overhead of a vtable being created for that class at runtime, and the memory overhead of an extra data member, the vptr, for every object created from the class. Normally, when client code calls a (non-inline) method, the compiler will place a jump to the function in the client code to the method. When the client code calls a virtual method, the compiler has to dereference the vptr to get to the vtable and then use the appropriate address stored there. Clearly, this involves an extra level of indirection.

There is a separate entry in the vtable for each virtual method in the base class, in the order in which they are declared. When you derive from a base class with virtual methods, the derived class will also have a vptr, but the compiler will make it point to the vtable of the derived class, that is, the compiler will populate the vtable with the addresses of the virtual method implementations in the derived class. If the derived class does not implement a virtual method it inherits, then the pointer in the vtable will be to the base class method. This is illustrated in the following diagram:

On the left-hand side, there are two classes; the base class has two virtual functions, and the derived class implements just one of these. On the right-hand side, there is an illustration of the memory layout. Two objects are shown as a base object and a derived object. Each object has a single vptr followed by the data members of the class, and the data members are arranged in such a way that the base class data members are arranged first, followed by the derived class data members. The vtable pointers contain method pointers to the virtual methods. In the case of the base class, the method pointers point to the methods implemented on the base class. In the case of the derived class, only the second method is implemented in the derived class, so the vtable for this class has a pointer to one virtual method in the base class and the other in the derived class.

This raises the question: what happens if a derived class introduces a new method, not available in the base class, and makes that virtual? This is not inconceivable since the ultimate base class could provide just part of the behavior needed and classes derived from it, provide more of the behavior to be called through virtual method dispatching on subclasses. The implementation is quite simple: the compiler creates a vtable for all of the virtual methods on the class, so if a derived class has extra virtual methods, the pointers for these appear in the vtable after the pointers to the virtual methods inherited from the base class.

When the object is called through a base class pointer, wherever that class is in the inheritance hierarchy, it will only see the vtable entries relevant to it:

Multiple inheritance and virtual method tables

If a class derives from more than one class and the parent classes have virtual methods, then the vtable for the derived class will be a combination of the vtables of its parents arranged in the order in which the parent classes were listed in the derivation list:

If the object is accessed through a base class pointer, the vptr has access to the part of the vtable relevant to that base class.

Virtual methods, construction, and destruction

The derived class portion of the object won't be constructed until the constructor has completed, so if you call a virtual method, the vtable entry will not be set up to call the correct method. Similarly, in a destructor, the derived class portions of the object will have already been destroyed-including their data members, and so the virtual methods on the derived class cannot be called because they may attempt to access data members that no

longer exist. If the virtual method dispatching was allowed in these situations, the result would be unpredictable. You should not call a virtual method in a constructor or a destructor, if you do, the call will resolve to the base class version of the method.
If a class is expected to be called through base class pointers with the virtual method dispatching, then you should make the destructor virtual. We do this because a user may delete a base class pointer, and in this situation, you will want the derived destructor to be called. If the destructor is not virtual and the base class pointer is deleted, then only the base class destructor is called, potentially causing a memory leak.
In general, a base class destructor should be either protected and non-virtual, or public and virtual. If the intention is to use the class through base class pointers, then the destructor should be public and virtual so that the derived class destructor is called, but if the base class is intended to be used to provide services available only through a derived class object, then you should not give direct access to base class objects, and so the destructor should be protected and non-virtual.

Containers and virtual methods

One advantage of the virtual methods is to put objects related by a base class into a container; earlier, we saw a specific case of using a built-in array of base class pointers, but what about the Standard Library containers? As an example, imagine that you have a class hierarchy where there is one base class, base, and three derived classes, derived1, derived2, and derived3, and each class implements a virtual method who, as used earlier. One attempt to put objects in a container may be as follows:

```
derived1 d1;
derived2 d2;
derived3 d3;
vector<base> vec = { d1, d2, d3 };
for (auto b : vec) b.who();
cout << endl;
```

The problem is that the vector holds base objects, and so as the items in the initialization list are put into the container, they are actually used to initialize new base objects. Since the type of vec is vector<base>, the push_back method will slice the object. Thus, the statement that calls the who method on each object will print a string base.
In order to have virtual method dispatching, we need to put the whole object in the container. We can do this either with a pointer or a reference. To use a pointer, you can use the addresses of stack objects as long as the vector does not live longer than the objects in the container. If you use objects created on the heap, then you need to ensure that the objects are deleted appropriately, and you can do this using smart pointers.
You may be tempted to create a container of references:

```
vector<base&> vec;
```

This will result in a slew of errors; unfortunately, none of them fully indicate the issue. The vector must contain types that are copy constructible and assignable. This is not the case with references because they are aliases to actual objects. There is a solution. The <functional> header contains an adapter class called reference_wrapper that has a copy constructor and assignment operator. The class converts a reference of an object to a pointer to that object. Now you can write the following:

```
vector<reference_wrapper<base> > vec = { d1, d2, d3 };
for (auto b : vec) b.get().who();
cout << endl;
```

The downside of using reference_wrapper is that to call the wrapped object (and their virtual methods), you need to call the get method, which will return a *reference* to the wrapped object.

Friends and inheritance

In C++, friendship is not inherited. If a class makes another class (or function) a friend, it means that the friend has access to its private and protected members as if the friend is a member of the class. If you derive from the friend class, the new class is not a friend of the first class, and it has no access to the members of that first class.
In the last chapter, we saw how you can insert an object into an ostream object to print it by writing a global insertion operator and making this a friend of the class.

In the following, the friend function is implemented inline, but it is actually a separate, global function that can be called without an object or name resolution with the class name:

```
class base
{
int x = 0;
public:
friend ostream& operator<<(ostream& stm, const base& b)
{
// thru b we can access the base private/protected members
stm << "base: " << b.x << " ";
return stm;
}
};
```

If we derive from the base class, we will need to implement a friend function to insert the derived object into the stream. Since the function is a *friend*, it will be able to access the private and protected members of the derived class, but it cannot access the private members of the base class. This situation will mean that the insertion operator that is a *friend* of the derived class can only print out part of the object.
If a derived class object is cast to a base class, say, through a pointer or reference when passing by reference, and the object is printed, it will be the base version of the insertion operator that will be called. The insertion operator is a friend function so that it has access to the class' non-public data members, but being a *friend* is not enough to allow it to be a virtual method, so there is no virtual method dispatching.
Although the friend function cannot be called as a virtual method, it can call virtual methods and get the method dispatching:

```
class base
{
int x = 0;
protected:
virtual void output(ostream& stm) const { stm << x << " "; }
public:
friend ostream& operator<<(ostream& stm, const base& b)
{
b.output(stm);
return stm;
}
};
class derived : public base
{
int y = 0;
```

```
protected:
virtual void output(ostream& stm) const
{
base::output(stm);
stm << y << " ";
}
};
```

In this version, there is just one insertion operator and it is defined for the base class. This means that any object that can be converted to the base class can be printed using this operator. The actual work of printing out the object is delegated to a virtual function called output. This function is protected because it is intended only to be used by the class or derived classes. The base class version of this prints out the data members of the base class. The derived class version has two tasks: printing out the data members in the base class and then printing out the data members specific to the derived class. The first task is accomplished by calling the base class version of the method by qualifying the name with the base class name. The second task is simple because it has access to its own data members. If you were to derive another class from derived, then its version of output function will be similar, but it would call derived::output.

Now when an object is inserted into an ostream object like cout, the insertion operator will be called, and the call to the output method will be dispatched to the appropriate derived class.

Override and final

As mentioned earlier, if you type the prototype of a derived virtual method wrong, for example, use the wrong parameter types, the compiler will treat the method as a new method and will compile it. It is perfectly legal for a derived class not to override the method of the base class; this is a feature that you will often want to use. However, if you make a mistake in typing the prototype of a derived virtual method, the base method will be called when you intended your new version to be called. The override specifier is designed to prevent this bug. When the compiler sees this specifier, it knows that you intend to override a virtual method inherited from a base class and it will search the inheritance chain to find a suitable method. If no such method can be found, then the compiler will issue an error:

```
struct base
{
virtual int f(int i);
};
```

```
struct derived: base
{
virtual int f(short i) override;
};
```

Here, derived::f won't compile because there is no method in the inheritance chain with the same signature. The override specifier gets the compiler to perform some useful checks, so it is a good habit to use it on all derived overridden methods.

C++11 also provides a specifier called final, which you can apply to a method to indicate that a derived class cannot override it, or you can apply it to a class to indicate that you cannot derive from it:

```
class complete final { /* code */ };
class extend: public complete{}; // won't compile
```

It is rare that you'll want to use this.

Virtual inheritance

Earlier, we talked about the so-called *diamond* problem with multiple inheritance, where a class inherits from a single ancestor class via two base classes. When a class inherits from another class, it will get the parent class' data members so that an instance of the derived class is treated as being made up of the base class data members and the derived class data members. If the parent classes are derived from the same ancestor class, they will each get the ancestor class' data members resulting in the final derived classes getting copies of the ancestor class' data members from each parent class:

```
struct base { int x = 0; };
struct derived1 : base { /*members*/ };
struct derived2 : base { /*members*/ };
struct most_derived : derived1, derived2 { /*members*/ };
```

When you create an instance of the most_derived class, you have two copies of base in the object: one from each of the derived1 and derived2. This means that the most_derived object will have two copies of the data member x. Clearly, the intention is for the derived class to get just one copy of the ancestor class' data members, so how can this be achieved? The solution to this problem is **virtual inheritance**:

```
struct derived1 : virtual base { /*members*/ };
struct derived2 : virtual base { /*members*/ };
```

Without virtual inheritance, derived classes just call the constructors of their immediate parent. When you use virtual inheritance, the most_derived class has the responsibility to call the constructor of the topmost parent class and if you do not explicitly call the base class constructor, the compiler will automatically call the default constructor:

```
derived1::derived1() : base(){}
derived2::derived2() : base(){}
most_derived::most_derived() : derived1(), derived2(), base(){}
```

In the preceding code, the most_derived constructor calls the base constructor because this is the base class that its parent classes inherit from virtually. The virtual base classes are always created before the non-virtual base classes. In spite of the call to the base constructor in the most_derived constructor, we still have to call the base constructor in the derived classes. If we further derive from most_derived, then that class must call the constructor of base too because that is where the base object will be created. Virtual inheritance is more expensive than single or multiple inheritance.

Abstract classes

A class with virtual methods is still a **concrete class**--you can create instances of the class. You may decide that you want to provide just a part of the functionality, with the intention that a user *has* to derive from the class and add the missing functionality.

One way to do this is to provide a virtual method that has no code. This means that you can call the virtual method in your class, and at runtime, the version of the method in the derived class will be called. However, although this provides a mechanism for you to call derived methods in your code, it does not *force* the implementation of those virtual methods. Instead, the derived class will inherit the empty virtual methods and if it does not override them, the client code will be able to call the empty method. You need a mechanism to *force* a derived class to provide an implementation of those virtual methods.

C++ provides a mechanism called **pure virtual methods** that indicates that the method should be overridden by a derived class. The syntax is simple, you mark the method with = 0:

```
struct abstract_base
{
virtual void f() = 0;
void g()
{
cout << "do something" << endl;
f();
}
};
```

This is the complete class; it is all that this class provides for the definition of the method f. This class will compile even though the method g calls a method that has no implementation. However, the following will not compile:

```
abstract_base b;
```

By declaring a pure virtual function, you make the class abstract, which means that you cannot create instances. You can, however, create pointers or references to the class and call code on them. This function will compile:

```
void call_it(abstract_base& r)
{
r.g();
}
```

This function only knows about the public interface of the class and does not care how that is implemented. We have implemented the method g to call the method f to show that you can call a pure virtual method in the same class. In fact, you can call the pure virtual function outside the class too; this code is just as valid:

```
void call_it2(abstract_base& r)
{
r.f();
}
```

The only way to use an abstract class is to derive from it and implement the pure virtual functions:

```
struct derived1 : abstract_base
{
virtual void f() override { cout << "derived1::f" << endl; }
};
struct derived2 : abstract_base
{
virtual void f() override { cout << "derived2::f" << endl; }
};
```

Here are two classes derived from the abstract class, which both implement the pure virtual function. These are concrete classes and you can create instances of them:

```
derived1 d1;
call_it(d1);
derived2 d2;
call_it(d2);
```

Abstract classes are used to indicate that a specific functionality has to be provided by a derived class, and the = 0 syntax indicates that the method body is not provided by the abstract class. In fact, it is more subtle than this; the class must be derived and the method called on the derived class must be defined on the derived class, but the abstract base class can also provide a body for the method:

```
struct abstract_base
{
virtual int h() = 0 { return 42; }
};
```

Again, this class cannot be instantiated, you *must* derive from it and you *must* implement the method to be able to instantiate an object:

```
struct derived : abstract_base
{
virtual int h() override { return abstract_base::h() * 10; }
};
```

The derived class can call the pure virtual function defined in the abstract class, but when external code calls such a method, it will always result (through method dispatching) in a call to the implementation of the virtual method on the derived class.

Obtaining type information

C++ provides type information, that is, you can get information that is unique to that type and, which identifies it. C++ is a strongly typed language so the compiler will determine type information at compile time and will enforce typing rules when it comes to conversions between variable types. Any type checking that the compiler does, you can do as the developer. As a general rule of thumb if you need to cast using static_cast, const_cast, reinterpret_cast, or C-like casts, then you are making the types do something they shouldn't and hence you should reconsider rewriting your code. The compiler is very good at telling you where there is a misalignment of types, so you should use this as a hint to reassess your code.

A *no casting* rule can be a bit too strict, and often code using casts is simpler to write and easier to read, but such a rule does focus your mind to always question whether a cast is needed.

When you use polymorphism, you will often get a pointer or reference to a type that is different to the type of the object, and this becomes especially true when you move to interface programming where frequently the actual object is unimportant, as it is the behavior that is important. There may be occasions when you need to obtain type information and the compiler is unable to help you at compile time. C++ provides a mechanism to obtain type information called **Runtime Type Information** (**RTTI**) because you can obtain this information at runtime. This information is obtained using the typeid operator on an object:

```
string str = "hello";
const type_info& ti = typeid(str);
cout << ti.name() << endl;
```

The result is the following printed at the command-line:

```
class std::basic_string<char,struct std::char_traits<char>,
class std::allocator<char> >
```

This reflects that the string class is in fact a typedef for the templated class, basic_string, with a char as the character type with character traits described by the specialization of the char_traits class and an allocator object (used to maintain the buffer used by the string), which is a specialization of the allocator class.

The typeid operator returns a const reference to a type_info object, and in this case, we use the name method to return a const char pointer to the name of the type of the object. This is the readable version of the type name. The type name is actually stored in a compact, decorated name, which is obtained via the raw_name method, but if you want to store objects according to their type (in a dictionary object, for example), then a more efficient mechanism is to use the 32-bit integer returned from the hash_code method rather than the decorated name. In all cases, the value returned will be the same for all objects of the same type, but different to objects of another type.

The type_info class has no copy constructor or copy assignment operator, and so objects of this class cannot be put in a container. If you want to put type_info objects in an associative container like a map, then you have two options. First you can put a pointer to the type_info object into a container (a pointer can be obtained from a reference); in which case, if the container is ordered, you need to define a comparison operator. The type_info class has a before method, which can be used to compare two type_info objects.

The second option (in C++11) is to use objects of the type_index class as the key to the associative container, and this class is used to wrap the type_info objects.

The type_info class is intended to be read-only, and the only way to create instances is through the typeid operator. You can, however, call the comparison operators, == and !=, on type_info objects, which means that you can compare at runtime the types of objects. Since you can apply the typeid operator on both variables and types, it means that you can use the operator to perform casts that are safe from slicing or from casting to a completely unrelated type:

```
struct base {};
struct derived { void f(); };
void call_me(base *bp)
```

```
{
derived *dp = (typeid(*bp) == typeid(derived))
? static_cast<derived*>(bp) : nullptr;
if (dp != nullptr) dp->f();
}
int main()
{
derived d;
call_me(&d);
return 0;
}
```

This function can take a pointer for any class that is derived from the base class. The first line uses the conditional operator where the comparison is between the type information for the object pointed to by the function parameter and the type of the class derived. If the pointer is to a derived object, then the cast will work. If the pointer is to an object of another derived type, but not the derived class, then the comparison will fail and the expression evaluates to nullptr. The call_me function will only call the f method if the pointer points to an instance of the derived class.

C++ provides a cast operator that performs runtime, and such type checking at runtime is called dynamic_cast. If the object can be cast to the requested type, then the operation will succeed and return a valid pointer. If the object cannot be accessed through the requested pointer, then the cast fails and the operator returns nullptr. This means that whenever you use dynamic_cast, you should always check the returned pointer before using it. The call_me function can be rewritten as follows:

```
void call_me(base *bp)
{
derived *dp = dynamic_cast<derived*>(bp);
if (dp != nullptr) dp->f();
}
```

This is essentially the same code as earlier; the dynamic_cast operator performs runtime type checking and returns an appropriate pointer.

Note that you cannot downcast, neither to a virtual base class pointer nor to a class derived through protected or private inheritance. The dynamic_cast operator can be used for casts other than downcasts; clearly, it will work for an upcast (to a base class, although it is not necessary), it can be used for casts sideways:

```
struct base1 { void f(); };
struct base2 { void g(); };
struct derived : base1, base2 {};
```

Here there are two base classes, so if you access a derived object through one of the base class pointers, you can use the dynamic_cast operator to cast to a pointer of the other base class:

```
void call_me(base1 *b1)
{
base2 *b2 = dynamic_cast<base2*>(b1);
if (b2 != nullptr) b2->g();
}
```

Smart pointers and virtual methods

If you want to use dynamically created objects, you will want to use smart pointers to manage their lifetime. The good news is that virtual method dispatching works through smart pointers (they are simply wrappers around object pointers), and the bad news is that the class relationships are lost when you use smart pointers. Let's examine why.

For example, the following two classes are related by inheritance:

```
struct base
{
Virtual ~base() {}
virtual void who() = 0;
};
struct derived : base
{
virtual void who() { cout << "derivedn"; }
};
```

This is straightforward: that implement a virtual method, which indicates the type of the object. There is, a virtual destructor because we are going to hand over the lifetime management to a smart pointer object and we want to ensure that the derived class destructor is called appropriately. You can create an object on the heap using make_shared or the constructor of the shared_ptr class:

```
// both of these are acceptable
shared_ptr<base> b_ptr1(new derived);
shared_ptr<base> b_ptr2 = make_shared<derived>();
```

A derived class pointer can be converted to a base class pointer and this is explicit in the first statement: new returns a derived* pointer, which is passed to a shared_ptr<base> constructor that expects a base* pointer. The situation in the second statement is a bit more complicated. The make_shared function returns a temporary shared_ptr<derived> object that is converted to a shared_ptr<base> object. This is carried out by a conversion constructor on the shared_ptr class that calls a **compiler intrinsic** called __is_convertible_to, which determines if one pointer type can be converted to the other. In this case, there is an upcast so the conversion is allowed.
Compiler intrinsic are essentially functions provided by the compiler. In this example __is_convertible_to(derived*, base*) will return true and __is_convertible_to(base*, derived*) will return false. You will rarely need to know about intrinsics unless you are writing libraries.

Since a temporary object is created in the statement using the make_shared function, it is more efficient to use the first statement.
The operator-> on a shared_ptr object will give direct access to the wrapped pointer and hence this means that the following code will perform virtual method dispatching, as expected:

```
shared_ptr<base> b_ptr(new derived);
b_ptr->who(); // prints "derived"
```

The smart pointer will ensure that the derived object is destroyed through the base class pointer when b_ptr goes out of scope, and since we have a virtual destructor, appropriate destruction will occur.
If you have multiple inheritance, you can use dynamic_cast (and RTTI) to cast between pointers to the base classes so that you can select only the behavior that you need. Consider the following code:

```
struct base1
{
Virtual ~base1() {}
virtual void who() = 0;
};
struct base2
{
Virtual ~base2() {}
virtual void what() = 0;
};
struct derived : base1, base2
{
virtual void who() { cout << "derivedn"; }
virtual void what() { cout << "derivedn"; }
};
```
If you have a pointer to either of these base classes, you can convert one to the other:
```
shared_ptr<derived> d_ptr(new derived);
d_ptr->who();
d_ptr->what();
base1 *b1_ptr = d_ptr.get();
b1_ptr->who();
base2 *b2_ptr = dynamic_cast<base2*>(b1_ptr);
b2_ptr->what();
```

The who and what methods can be called on a derived* pointer and hence they can be called on the smart pointer. The following lines obtain a base class pointer so that *specific* behavior is accessed. In this code, we call the get method to get the raw pointer from the smart pointer. The problem with this method is that there is now a pointer to the object that is not protected by the smart pointer lifetime management, so it is possible for code to call delete on either pointer b1_ptr or b2_ptr and cause problems later when the smart pointer attempts to delete the object.
This code works, and there is correct lifetime management of the dynamically-created object in this code, but accessing raw pointers like this is inherently unsafe because there is no guarantee that the raw pointers will not be deleted. The temptation is to use smart pointers:
```
shared_ptr<base1> b1_ptr(d_ptr.get());
```
The problem is that even though the classes base1 and derived are related, the classes shared_ptr<derived> and shared_ptr<base1> are *not* related, and so a different control block will be used for each smart pointer type even though they refer to the *same object*. The shared_ptr class will reference the count using the control block and will delete the object when the reference count falls to zero. Having two unrelated shared_ptr objects and two control blocks to the same object means that they will attempt to manage the lifetime of the derived object independently of each other, and this will ultimately mean one smart pointer deleting the object before the other has finished with it.
There are three messages here: a smart pointer is a lightweight wrapper around a pointer, so you can call virtual methods with method dispatching; however, be cautious about using raw pointers obtained from smart pointers, and bear in mind that although you can have many shared_ptr objects to the same object, they must be of the same type so that only one control block is used.

Interfaces

Pure virtual functions and virtual method dispatching leads to an incredibly powerful way of writing object-orientated code, which is called **interfaces**. An interface is a class that has no functionality; it only has pure virtual functions. The purpose of an interface is to define a behavior. A concrete class that derives from an interface *must* provide an implementation of all of the methods on the interface, and hence this makes the interface a kind of contract. Users of objects that implement an interface have a guarantee that the object that has the interface will implement *all* the methods of the interface. Interface programming decouples behavior from the implementation. Client code is only interested in behavior and they are not interested in the actual class that provides the interface.

For example, an IPrint interface could give access to the behavior of printing a document (setting page size, orientation, number of copies, and telling the printer to print the document). The IScan interface can give access to the behavior of scanning a sheet of paper (resolution, grayscale or color, and adjustments like rotation and cropping). These two interfaces are two different behaviors. Client code will use an IPrint if it wants to print a document or an IScan interface pointer if it wants to scan a document. Such client code does not care whether it is a printer object that implements the IPrint interface or a printer_scanner object that implements both the IPrint and IScan interfaces. Client code that is passed to an IPrint* interface pointer is guaranteed that it can call every method.

In the following code, we have defined the IPrint interface (the define makes it more obvious that we are defining abstract classes as interfaces):

```
#define interface struct
interface IPrint
{
virtual void set_page(/*size, orientation etc*/) = 0;
virtual void print_page(const string &str) = 0;
};
```

A class can implement this interface:

```
class inkjet_printer : public IPrint
{
public:
virtual void set_page(/*size, orientation etc*/) override
{
// set page properties
}
virtual void print_page(const string &str) override
{
cout << str << endl;
}
};
void print_doc(IPrint *printer, vector<string> doc);
```

You can then create the printer object and call the function:

```
inkjet_printer inkjet;
IPrint *printer = &inkjet;
printer->set_page(/*properties*/);
vector<string> doc {"page 1", "page 2", "page 3"};
print_doc(printer, doc);
```

Our inkjet printer is also a scanner, so we can make it implement the IScan interface:

```
interface IScan
{
virtual void set_page(/*resolution etc*/) = 0;
virtual string scan_page() = 0;
};
```

The next version of the inkject_printer class can use multiple inheritance to implement this interface, but note that there is a problem. The class already implements a method called set_page, and since the page properties of the printer will be different from the page properties of the scanner, we want a different method for the IScan interface. We can address this with two different methods and qualifying their names:

```
class inkjet_printer : public IPrint, public IScan
{
public:
virtual void IPrint::set_page(/*etc*/) override { /*etc*/ }
virtual void print_page(const string &str) override
{
cout << str << endl;
}
virtual void IScan::set_page(/*etc*/) override { /*etc*/ }
virtual string scan_page() override
{
static int page_no;
string str("page ");
str += to_string(++page_no);
return str;
}
};
void scan_doc(IScan *scanner, int num_pages);
```
Now, we can get the IScan interface on the inkjet object and call it as a scanner:
```
inkjet_printer inkjet;
IScan *scanner = &inkjet;
scanner->set_page(/*properties*/);
scan_doc(scanner, 5);
```

Since the inkject_printer class derives from both the IPrinter and IScan interfaces, you can obtain one interface pointer and cast to the other through the dynamic_cast operator since this will use RTTI to ensure that the cast is possible. So assuming that you've got an IScanner interface pointer, you can test to see if you can cast this to an IPrint interface pointer:
```
IPrint *printer = dynamic_cast<IPrint*>(scanner);
if (printer != nullptr)
{
printer->set_page(/*properties*/);
vector<string> doc {"page 1", "page 2", "page 3"};
print_doc(printer, doc);
}
```
Effectively, the dynamic_cast operator is being used to request one interface pointer if the behavior represented by another interface is unavailable on the object that pointer points to.
An interface is a contract; once you have defined it, you should *never* change it. This does not constrain you from changing the class. In fact, this is the advantage of using interfaces because the class implementation can change completely, but as long as it continues to implement the interfaces that the client code uses, users of the class can continue to use the class (although a recompile will be needed). There are cases when you will discover that the interface you defined is inadequate. Perhaps there is a parameter which is incorrectly typed that you need to fix, or perhaps you need to add additional functionality.
For example, imagine that you want to tell the printer object to print an entire document rather than a page at a time. The way to do this is to derive from the interface that needs changing and create a new interface; interface inheritance:
```
interface IPrint2 : IPrint
{
virtual void print_doc(const vector<string> &doc) = 0;
};
```
Interface inheritance means that IPrint2 has three methods, set_page, print_page, and print_doc. Since the IPrint2 interface is an IPrint interface, this means that when you implement the IPrint2 interface, you also implement the IPrint interface, so you need to change the class to derive from the IPrint2 interface to add the new functionality:
```
class inkjet_printer : public IPrint2, public IScan
{
public:
virtual void print_doc(const vector<string> &doc) override {
/* code*/
}
```

```
// other methods
};
```
The other two methods on the IPrint2 interface already exist on this class from the implementation of the IPrint interface. Now, a client can obtain both IPrint pointers and IPrint2 pointers from instances of this class. You have extended the class, and yet the older client code will still compile.

Microsoft's **Component Object Model** (**COM**) takes this concept a step further. COM is based upon interface programming, so COM objects are only ever accessed through interface pointers. The extra step is that this code can be loaded into your process using a dynamic loaded library, or in another process on your machine or on another machine, and since you use interface programming, the objects are accessed in *exactly* the same way regardless of their location.

Class relationships

Inheritance appears to be an ideal way to reuse code: you write it once in as generic way possible and then derive a class from the base class and reuse the code, specializing it if necessary. You will find, however, a lot of advice against this. Some people will tell you that inheritance is the worst way possible to reuse code and you should use composition instead. In fact, the situation is somewhere between the two: inheritance offers some benefits, but it should not be treated as the best or only solution.

It is possible to get carried away with designing a class library, and there is a general principle to bear in mind: the more code you write, the more maintenance you (or someone else) will have to do. If you change a class, all the other classes that depend upon it will change.

At the highest level, you should be aware of three main issues to avoid:

Rigidity: It is too hard to change a class because any change will affect too many other classes.

Fragility: When you change your class, it could cause unexpected changes in other classes.

Immobility: It is hard to reuse the class because it is too dependent on other classes.

This occurs when you have tight coupling between classes. In general, you should design your classes to avoid this and interface programming is an excellent way to do this because an interface is simply a behavior and not an instance of a specific class.

Such problems occur when you have *dependency inversion*, that is, higher level code, using components, becomes dependent upon the details of how the lower level components are implemented. If you have code that performs some action and then logs the result if you write that logging to use a specific device (say the `cout` object), then the code is rigidly coupled to, and dependent upon, that logging device and you have no option in the future to change to another device. If you abstract the functionality, typically, through an interface pointer-then you break this dependency enabling the code to be used with other components in the future.

Another principle is that in general you should design your classes to be extendable. Inheritance is quite a brute force mechanism to extend a class because you are creating a whole new type. If the functionality only needs to be refined, then inheritance can be an overkill. A more lightweight form of refining an algorithm is to pass a method pointer (or a functor), or an interface pointer to the method of a class for that method to call at an appropriate time to refine how it works.

For example, most sort algorithms require that you pass a method pointer to perform comparisons of two objects of the type that it is sorting. The sort mechanism is generic and does the work of ordering the objects in the most efficient manner, but it bases this on you telling it how to order the two objects. It is excessive to write a new class for every type since the majority of the algorithm remains the same.

Using mixin classes

The **mixin** technique allows you to provide extensibility to classes without the lifetime issues of composition or the heavyweight aspect of raw inheritance. The idea here is that you have a library with specific functionality that can be added to an object. One way to do this is to apply it as a base class with public methods, and so if the derived class publicly derives from that class, it will also have those methods as public methods. This works fine unless the functionality requires that the derived class performs some functionality too in those methods, in which case the documentation of the library will require that the derived class overrides the method, calls the base class implementation, and adds their own code to the method to complete the implementation (the base class method could be called before, or after the extra derived class code, and the documentation would have to specify this). We have seen this used several times so far in this chapter, and it is a technique used by some older class libraries, for example, Microsoft's **Foundation Classes library** (**MFC**). Visual C++ makes this easier because it generates MFC code with a wizard tool and there are comments about where the developer should add their code.

The problem with this approach is that it requires the developer deriving from the base class implements specific code and follows the rules.

There is a possibility that the developer will write code that compiles and runs, but since it is not written to the desired rules, it has the wrong behavior at runtime.

A mixin class turns this concept on its head. Instead of the developer deriving from a base class provided by the library and extending the functionality provided, the mixin class provided by the library *is derived from a class provided by the developer*. This solves several problems. First, the developer will have to provide specific methods as required by the documentation, otherwise the mixin class (which will use those methods) will not compile. The compiler is enforcing the rules of the class library author to require that the developer using the library provides specific code. Second, the methods on the mixin class can call the base class methods (provided by the developer) exactly where it needs them. The developer using the class library is no longer provided with detailed instructions about how their code is developed, other than that, they have to implement certain methods.

So, how can this be achieved? The class library author does not know about the code that the client developer will write and they do not know about the names of the classes a client developer will write, so they cannot derive from such classes. C++ allows you to provide a type through a template parameter so that the class is instantiated using this type at compile time. With mixin classes, the type passed through a template parameter is the name of a type that will be used as the base class. The developer simply provides a class with the specific methods and then creates a specialization of the mixin class using their class as the template parameter:

```
// Library code
template <typename BASE>
class mixin : public BASE
{
public:
void something()
{
cout << "mixin do something" << endl;
BASE::something();
cout << "mixin something else" << endl;
}
};
// Client code to adapt the mixin class
class impl
{
public:
void something()
{
cout << "impl do something" << endl;
}
};
```

This class is used in this way:
```
mixin<impl> obj;
obj.something();
```
As you can see, the mixin class implements a method called something and it calls a base class method called something. This means that a client developer using the functionality of the mixin class must implement a method with this name and with the same prototype, otherwise the mixin class cannot be used. The client developer writing the impl class does not know how or where their code will be used, just that they have to provide methods with specific names and prototypes. In this case, the mixin::something method calls the base class method in the code between the functionality that it provides, the writer of the impl class does not need to know this. The output of this code is as follows:
```
mixin do something
impl do something
mixin something else
```
This shows that the mixin class can call the impl class where it thinks is appropriate. The impl class only has to provide the functionality; the mixin class determines how it is used. In fact, any class that implements a method with the right name and prototype can be provided as a parameter to the template of the mixin class-even another mixin class!
```
template <typename BASE>
class mixin2 : public BASE
{
public:
void something()
{
cout << "mixin2 do something" << endl;
BASE::something();
cout << "mixin2 something else" << endl;
}
};
```
This can be used like this:
```
mixin2< mixin<impl> > obj;
obj.something();
```
The result will be as follows:
```
mixin2 do something
mixin do something
impl do something
mixin something else
mixin2 something else
```

Note that the mixin and mixin2 classes know nothing about each other, other than the fact that the appropriate methods are implemented.
Since the mixin class cannot be used without the type provided by the template parameter, they are sometimes called abstract subclasses.
This works fine if the base class only has a default constructor. If the implementation requires another constructor, then the mixin must know what constructor to call and must have appropriate parameters. Also, if you chain the mixins, then they get coupled through the constructors. One way to get around this is to use two stage construction, that is, provide a named method (say, init) used to initialize data members in the object after construction. The mixin classes will still be created using their default constructors as earlier, and so there will be no coupling between the classes, that is, the mixin2 class will know nothing about the data members of mixin or those of impl:
```
mixin2< mixin<impl> > obj;
obj.impl::init(/* parameters */); // call impl::init
obj.mixin::init(/* parameters */); // call mixin::init
obj.init(/* parameters */); // call mixin2::init
obj.something();
```

This works because you can call a public base class method as long as you qualify the name of the method. The parameter list in these three init methods can be different. However, this does pose the problem that the client now has to initialize all the base classes in the chain.

This is the approach that Microsoft's **ActiveX Template Library** (**ATL**) (now part of MFC) uses to provide implementation of standard COM interfaces.

Using polymorphism

In the following example, we will create code that simulates a team of C++ developers. The code will use interfaces to decouple the classes so that it is possible to change the services that a class uses without changing that class. In this simulation, we have a manager managing a team, so a property of the manager is their team. Further, every worker, whether a manager or a team member have some common properties and behaviors--they all have a name and a job position and they all do work of some kind.

Create a folder for the chapter and in that folder, create a file called team_builder.cpp, and since this application will use a vector, smart pointers, and files, add the following lines to the top of the file:

```
#include <iostream>
#include <string>
#include <vector>
#include <fstream>
#include <memory>
using namespace std;
```

The application will have command-line parameters, but for the time being, just provide an empty copy of the main function:

```
int main(int argc, const char *argv[])
{
return 0;
}
```

We are going to define interfaces, so before the main function, add the following:

```
#define interface struct
```

This is just syntactic sugar, but it makes the code a bit more readable to show the purpose of the abstract classes. Under this, add the following interfaces:

```
interface IWork
{
virtual const char* get_name() = 0;
virtual const char* get_position() = 0;
virtual void do_work() = 0;
};
interface IManage
{
virtual const vector<unique_ptr<IWork>>& get_team() = 0;
virtual void manage_team() = 0;
};
interface IDevelop
{
virtual void write_code() = 0;
};
```

All the workers will implement the first interface, which gives access to their name and job position and a method that tells them to do some work. The two types of workers we will define are a manager who manages a team by scheduling their time and a developer who writes code. The manager has a vector of IWork* pointers, and since these pointers will be to objects created on the free store, the vector members are smart pointers wrapping these pointers. This is saying that the manager maintains the lifetime of these objects: while the manager object exists; so will their team.

The first action is to create a helper class that does the basic work of a worker. The reason for this will be apparent later in the example. This class will implement the IWork interface:

```
class worker : public IWork
{
string name;
string position;
public:
worker() = delete;
worker(const char *n, const char *p) : name(n), position(p) {}
virtual ~worker() {}
virtual const char* get_name() override
{ return this->name.c_str(); }
virtual const char* get_position() override
{ return this->position.c_str(); }
virtual void do_work() override { cout << "works" << endl; }
};
```

A worker object must be created with a name and job position. We will also have a helper class for a manager:

```
class manager : public worker, public IManage
{
vector<unique_ptr<IWork>> team;
public:
manager() = delete;
manager(const char *n, const char* p) : worker(n, p) {}
const vector<unique_ptr<IWork>>& get_team() { return team; }
virtual void manage_team() override
{ cout << "manages a team" << endl; }
void add_team_member(IWork* worker)
{ team.push_back(unique_ptr<IWork>(worker)); }
virtual void do_work() override { this->manage_team(); }
};
```

Note that the do_work method is implemented in terms of the virtual function, manage_team, which means that a derived class only needs to implement the manage_team method since it will inherit the do_work method from its parent and method dispatching will mean the correct method is called. The rest of the class is straightforward, but note that the constructor calls the base class constructor to initialize the name and job position (a manager is, after all, a worker) and that the manager class has a function to add items to the team as shared in smart pointers.

To test this out, we need to create a manager class that manages developers:

```
class project_manager : public manager
{
public:
project_manager() = delete;
project_manager(const char *n) : manager(n, "Project Manager")
{}
virtual void manage_team() override
{ cout << "manages team of developers" << endl; }
};
```

This overrides calls to the base class constructor passing the name of the project manager and a literal describing the job. The class also overrides manage_team to say what the manager actually does. At this point, you should be able to create a project_manager and add some members to their team (use the worker objects, you'll create developers in a moment). Add the following to the main function:

```
project_manager pm("Agnes");
pm.add_team_member(new worker("Bill", "Developer"));
pm.add_team_member(new worker("Chris", "Developer"));
pm.add_team_member(new worker("Dave", "Developer"));
pm.add_team_member(new worker("Edith", "DBA"));
```

This code will compile, but there will be no output when it runs, so create a method to print out a manager's team:

```
void print_team(IWork *mgr)
{
cout << mgr->get_name() << " is "
<< mgr->get_position() << " and ";
IManage *manager = dynamic_cast<IManage*>(mgr);
if (manager != nullptr)
{
cout << "manages a team of: " << endl;
for (auto team_member : manager->get_team())
{
cout << team_member->get_name() << " "
```

```
<< team_member->get_position() << endl;
}
}
else { cout << "is not a manager" << endl; }
}
```

This function shows how useful interfaces are. You can pass any worker to the function, and it will print out the information relevant to all workers (name and job position). It then asks the object if it is a manager by requesting the IManage interface. The function can only get the manager behavior (in this case, having a team) if the object implements this interface. At the end of the main function, after the last call to the program_manager object, call this function:

```
print_team(&pm)
```

Compile this code (remember to use the /EHsc switch) and run the code. You will get the following output:

```
Agnes is Project Manager and manages a team of:
Bill Developer
Chris Developer
Dave Developer
Edith DBA
```

Now we will add a level of polymorphism, so add the following classes before the print_team function:

```
class cpp_developer : public worker, public IDevelop
{
public:
cpp_developer() = delete;
cpp_developer(const char *n) : worker(n, "C++ Dev") {}
void write_code() { cout << "Writing C++ ..." << endl; }
virtual void do_work() override { this->write_code(); }
};
class database_admin : public worker, public IDevelop
{
public:
database_admin() = delete;
database_admin(const char *n) : worker(n, "DBA") {}
void write_code() { cout << "Writing SQL ..." << endl; }
virtual void do_work() override { this->write_code(); }
};
```

You can change the main function so that rather than using worker objects, you use cpp_developer for Bill, Chris, and Dave and database_admin for Edith:

```
project_manager pm("Agnes");
pm.add_team_member(new cpp_developer("Bill"));
pm.add_team_member(new cpp_developer("Chris"));
pm.add_team_member(new cpp_developer("Dave"));
pm.add_team_member(new database_admin("Edith"));
print_team(&pm);
```

You can now compile and run the code and see that not only can you add different types of objects to the manager's team, but you will also get the appropriate information printed through the IWork interface.

The next task is to add code to serialize and deserialize these objects. Serialization means writing the object's state (and type information) to a stream and deserialization will take that information and create a new object of the appropriate type and with the specified state. To do this, every object must have a constructor that takes an interface pointer to a deserialzer object, and the constructor should call this interface to extract the state of the object being created. Further, such classes should implement a method to serialize and write an object's state to a serializer object. Let's first look at serialization. At the top of the file, add the following interfaces:

```
#define interface struct
interface IWork;
// forward declaration
interface ISerializer {
virtual void write_string(const string& line) = 0;
virtual void write_worker(IWork *worker) = 0;
virtual void write_workers (
const vector<unique_ptr<IWork>>& workers) = 0;
};
interface ISerializable {
virtual void serialize(ISerializer *stm) = 0;
};
```

The forward declaration is needed because the ISerializer interface uses the IWork interface. The first interface, ISerializer, is implemented by an object that provides serialization services. This could be based on a file, a network socket, a database, or whatever you want to use to store the object. The underlying storage mechanism is unimportant to the user of this interface; all that is important is that the interface can store a string and it can store an entire object passed using an IWork interface pointer or a collection of such objects.

The objects that can be serialized must implement the ISerializable interface and this has a single method that takes an interface pointer to the object that will provide the serialization service. After the definition of the interfaces, add the following class:

```
class file_writer : public ISerializer
{
ofstream stm;
public:
file_writer() = delete;
file_writer(const char *file) { stm.open(file, ios::out); }
~file_writer() { close(); }
void close() { stm.close(); }
virtual void write_worker(IWork *worker) override
{
ISerializable *object = dynamic_cast<ISerializable*>(worker);
if (object != nullptr)
{
ISerializer *serializer = dynamic_cast<ISerializer*>(this);
serializer->write_string(typeid(*worker).raw_name());
object->serialize(serializer);
}
}
virtual void write_workers(
const vector<unique_ptr<IWork>>& workers) override
{
write_string("[[");
for (const unique_ptr<IWork>& member : workers)
{
write_worker(member.get());
}
write_string("]]"); // end marker of team
}
virtual void write_string(const string& line) override
{
stm << line << endl;
}
};
```

This class provides the ISerializer interface for a file, so the write_string method uses the ifstream insertion operator to write the string on a single line in the file. The write_worker method writes the worker object to the file. To do this, it first asks the worker object if it can serialize itself by casing the IWork interface an ISerializable interface. If the worker object implements this interface, the serializer can ask the worker object to serialize itself by passing the ISerializer interface pointer to the serialize method on the worker object. It is up to the worker object to determine the information that must be serialized.

The worker object knows nothing about the file_writer class other than the ISerializer interface, and the file_writer class knows nothing about the worker object, other than that it implements the IWork and ISerializable interfaces.

If the worker object is serializable, the first thing that the write_worker method does is obtain type information about the object. The IWork interface will be on a class (project_manager, cpp_developer, or database_admin) and so dereferencing the pointer will give the typeid operator access to the class type information. We store the raw type name in the serializer because it is compact. Once the type information is serialized, we ask the object to serialize itself by calling the serialize method on its ISerializable interface. The worker object will store whatever information it wants.

A manager object will need to serialize their team and they do this by passing a collection of worker objects to the write_workers method. This indicates that the objects being serialized are an array by writing them between two markers, [[and]]. Note that because the container has unique_ptr objects, there is no copy constructor because that would imply shared ownership. So instead, we access the items through the index operator, which will give us a reference to the unique_ptr object within the container.

Now, for every class that can be serialized, you must derive the class from ISerializable and implement the serialize method. The class inheritance tree means that every class for a type of worker derives from the worker class, so we only need this class to inherit from the ISerializable interface:

class worker : public IWork, public ISerializable

The convention is that a class only serializes its own state and delegates to its base class to serialize the base class object. At the top of the inheritance tree is the worker class, so at the bottom of this class, add the following interface method:

```
virtual void serialize(ISerializer *stm) override
{
stm->write_string(name);
stm->write_string(position);
}
```
This simply serializes the name and job position to the serializer. Note that the worker object does not know what the serializer will do with this information and does not know which class provides the ISerializer interface.
At the bottom of the cpp_developer class, add this method:
```
virtual void serialize(ISerializer* stm) override
{ worker::serialize(stm); }
```

The cpp_developer class does not have any additional state, so it delegates the serialization to its parent class. If the developer class had a state, then it would serialize this state after serializing the base object. Add exactly the same code to the bottom of the database_admin class.
The project_manager class also calls its base class, but this is manager, so add the following to the bottom of the project_manager class:
```
virtual void serialize(ISerializer* stm) override
{ manager::serialize(stm); }
```
The manager::serialize is more complicated because this class has state that should be serialized:
```
virtual void serialize(ISerializer* stm) override
{
worker::serialize(stm);
stm->write_workers(this->team);
}
```
The first action is to serialize the base class: a worker object. Then the code serializes the state of the manager object, which means serializing the team data member by passing this collection to the serializer.
To be able to test the serialization, create a method above the main method and move the project_manager code to the new method and add code to serialize the objects:
```
void serialize(const char* file)
{
project_manager pm("Agnes");
pm.add_team_member(new cpp_developer("Bill"));
pm.add_team_member(new cpp_developer("Chris"));
pm.add_team_member(new cpp_developer("Dave"));
pm.add_team_member(new database_admin("Edith"));
print_team(&pm);
cout << endl << "writing to " << file << endl;
file_writer writer(file);
ISerializer* ser = dynamic_cast<ISerializer*>(&writer);
ser->write_worker(&pm);
writer.close();
}
```

The preceding code creates a file_writer object for the specified file, obtains the ISerializer interface on that object, and then serializes the project manager object. If you have other teams, you can serialize them to the file before closing the writer object.
The main function will take two parameters. The first is the name of the file and the second is a character, r or w (to read or write the file). Add the following code to replace the main function:
```
void usage()
{
cout << "usage: team_builder file [r|w]" << endl;
cout << "file is the name of the file to read or write" << endl;
cout << "provide w to file the file (the default)" << endl;
cout << " r to read the file" << endl;
}
```

```
int main(int argc, char* argv[])
{
if (argc < 2)
{
usage();
return 0;
}
bool write = true;
const char *file = argv[1];
if (argc > 2) write = (argv[2][0] == 'w');
cout << (write ? "Write " : "Read ") << file << endl << endl;
if (write) serialize(file);
return 0;
}
```

You can now compile this code and run it, giving the name of a file:

```
team_builder cpp_team.txt w
```

This will create a file called cpp_team.txt containing information about the team; type it at the command-line with **type cpp_team.txt**:

```
.?AVproject_manager@@
Agnes
Project Manager
[[
.?AVcpp_developer@@
Bill
```

```
C++ Dev
.?AVcpp_developer@@
Chris
C++ Dev
.?AVcpp_developer@@
Dave
C++ Dev
.?AVdatabase_admin@@
Edith
DBA
]]
```

This file is not intended to be read by humans, but as you can see, it has one piece of information on each line and each serialized object is preceded by the type of the class. Now you will write the code to deserialize an object. The code needs a class that will read the serialization data and return worker objects. This class is tightly coupled to the serializer class, but it should be accessed through an interface so that it is not coupled to the worker objects. After the declaration of the ISerializable interface, add the following:

```
interface IDeserializer
{
virtual string read_string() = 0;
virtual unique_ptr<IWork> read_worker() = 0;
virtual void read_workers(vector<unique_ptr<IWork>>& team) = 0;
};
```

The first method obtains a serialization string and the other two methods obtain a single object and a collection of objects. Since these worker objects will be created on the free store, these methods use smart pointers. Every class can serialize itself, and so now you will make each serializable class able to deserialize itself. To do this, for every class that implements ISerializable, add a constructor that takes an IDeserializer interface pointer. Start with the worker class; add the following public constructor:

```
worker(IDeserializer *stm)
{
name = stm->read_string();
position = stm->read_string();
}
```

Essentially, this reverses what the serialize method does, it reads the name and position string from the deserializer *in the same order* in which they were passed to the serializer. Since the cpp_developer and database_admin classes have no state, they do not need to do any other deserializing work other than call the base class constructor. For example, add the following public constructor to the cpp_developer class:

```
cpp_developer(IDeserializer* stm) : worker(stm) {}
```

Add a similar constructor to the database_admin class.

Managers have a state, so there is a bit more work to deserialize them. Add the following to the manager class:

```
manager(IDeserializer* stm) : worker(stm)
{ stm->read_workers(this->team); }
```

The initializer list constructs the base class and after this is run, the constructor initializes the team collection with zero or more worker objects by calling read_workers on the IDeserializer interface. Finally, the project_manager class derives from the manager class, but adds no extra state, so add the following constructor:

```
project_manager(IDeserializer* stm) : manager(stm) {}
```

Now, every serializable class can deserialize itself, the next action is to write the deserializer class that will read a file. After the file_writer class, add the following (note that two methods are not implemented inline):

```
class file_reader : public IDeserializer
{
ifstream stm;
public:
file_reader() = delete;
file_reader(const char *file) { stm.open(file, ios::in); }
~file_reader() { close(); }
void close() { stm.close(); }
virtual unique_ptr<IWork> read_worker() override;
virtual void read_workers(
vector<unique_ptr<IWork>>& team) override;
virtual string read_string() override
{
string line;
getline(stm, line);
return line;
}
};
```

The constructor opens the specified file and the destructor closes it. The read_string interface method reads a line from the file and returns it as a string. The main work is carried out in the two interface methods not implemented here. The read_workers method will read a collection of IWork objects and put them into the collection passed by reference. This method will call the read_worker method for every object in the file and put them in the collection, and so the main work of reading the file is carried out in this method. The read_worker method is the only part of the class that has any coupling to the serializable classes, and because of this, it has to be defined below the definition of the worker classes. Above the serialize global function, add the following:

```
unique_ptr<IWork> file_reader::read_worker()
{
}
void file_reader::read_workers(vector<unique_ptr<IWork>>& team)
{
while (true)
{
unique_ptr<IWork> worker = read_worker();
if (!worker) break;
team.push_back(std::move(worker));
}
}
```

The read_workers method will read each object from the file using the read_worker method, which returns each object in a unique_ptr object. We want to put this object into the container, but because there should be exclusive ownership of the pointer, we need to move the ownership into the object in the container. There are two ways to do this. The first way is simply to use the call to read_worker as the parameter to push_back. The read_worker method returns a temporary object, which is an rvalue, so the compiler will use move semantics when creating the object in the container. We do not do this because the read_worker method may return a nullptr (which we want to test for), so instead we create a new unique_ptr object (move semantics will pass ownership to this object), and once we have tested that this object is not a nullptr, we call the Standard Library function, move, to move copy the object into the container.

If the read_worker method reads the end marker of an array, it returns a nullptr and hence the read_workers method loops, reading each worker and putting them in the collection until a nullptr is returned.

Implement the read_worker method like this:

```
unique_ptr<IWork> file_reader::read_worker()
{
string type = read_string();
if (type == "[[") type = read_string();
if (type == "]]") return nullptr;
if (type == typeid(worker).raw_name())
{
return unique_ptr<IWork>(
dynamic_cast<IWork*>(new worker(this)));
}
return nullptr;
}
```

The first line reads the type information of the worker object from the file so that it knows what object to create. Since the file will have markers to indicate the array of team members, the code has to detect these. If the start of the array is detected, the marker string is ignored and the next line is read to get the type of the first object in the team. If the end marker is read, then this is the end of the array so a nullptr is returned.

The code for a worker object is shown here. The if statement tests to check whether the type string is the same as the raw name of the worker class. If it is, then we must create a worker object and request that it deserializes itself by calling the constructor that takes an IDeserializer pointer. The worker object is created on the free store and the dynamic_cast operator is called to obtain the IWork interface pointer, which is then used to initialize a smart pointer object. The constructor for the unique_ptr is explicit, so you have to call it. Now add similar code for all the other serializable classes:

```
if (type == typeid(project_manager).raw_name())
{
return unique_ptr<IWork>(
dynamic_cast<IWork*>(new project_manager(this)));
}
if (type == typeid(cpp_developer).raw_name())
{
return unique_ptr<IWork>(
dynamic_cast<IWork*>(new cpp_developer(this)));
}
```

```
if (type == typeid(database_admin).raw_name())
{
return unique_ptr<IWork>(
dynamic_cast<IWork*>(new database_admin(this)));
}
```

Finally, you need to create a file_reader and deserialize a file. After the serialize function, add the following:

```
void deserialize(const char* file)
{
file_reader reader(file);
while (true)
{
unique_ptr<IWork> worker = reader.read_worker();
if (worker) print_team(worker.get());
else break;
}
reader.close();
}
```

This code simply creates a file_reader object based on the file name and then reads each worker object from the file printing out the object and, if it is a project_manager, prints out their team. Finally, add a line in the main function to call this function:

```
cout << (write ? "Write " : "Read ") << file << endl << endl;
if (write) serialize(file);
else deserialize(file);
```

Now you can compile the code and use it to read in the serialization file with the following:

```
team_builder cpp_team.txt r
```

(Note the r parameter.) The code should print out the objects that you serialized to the file. The previous example has shown that you can write serializable objects that do not know about the mechanism that is used to serialize them. If you want to use a different mechanism than a flat file (for example, an XML file or to a database), you do not need to alter any of the worker classes. Instead, you write an appropriate class that implements the ISerializer interface and the IDeserailizer interface. If you need to create another worker class, all you need to do is alter the read_worker method to deserialize objects of that type.

Strings and numbers

The Standard Library contains various functions and classes to convert between C++ strings and numeric values.

Converting strings to numbers

The C++ standard library contains functions with names like stod and stoi that convert a C++ string object to a numeric value (stod converts to a double and stoi converts to an integer). For example:

```
double d = stod("10.5");
d *= 4;
cout << d << "n"; // 42
```

This will initialize the floating-point variable d with a value of 10.5, which is then used in a calculation and the result is printed on the console. The input string may have characters that cannot be converted. If this is the case then the parsing of the string ends at that point. You can provide a pointer to a size_t variable, which will be initialized to the location of the first character that cannot be converted:

```
string str = "49.5 red balloons";
size_t idx = 0;
```

```
double d = stod(str, &idx);
d *= 2;
string rest = str.substr(idx);
cout << d << rest << "n"; // 99 red balloons
```

In the preceding code, the idx variable will be initialized with a value of 4, indicating that the space between the 5 and r is the first character that cannot be converted to a double.

Converting numbers to strings

The <string> library provides various overloads of the to_string function to convert integer types and floating point types into a string object. This function does not allow you to provide any formatting details, so for an integer you cannot indicate the radix of the string representation (for example, hex), and for floating point conversions, you have no control over options like the number of significant figures. The to_string function is a simple function with limited facilities. A better option is to use the stream classes, as explained in the following section.

Using stream classes

You can print floating point numbers and integers to the console using the cout object (an instance of the ostream class) or to files with an instance of ofstream. Both of these classes will convert numbers to strings using member methods and manipulators to affect the formatting of the output string. Similarly, the cin object (an instance of the istream class) and the ifstream class can read data from formatted streams.

Manipulators are functions that take a reference to a stream object and return that reference. The Standard Library has various global insertion operators whose parameters are a reference to a stream object and a function pointer. The appropriate insertion operator will call the function pointer with the stream object as its parameter. This means that the manipulator will have access to, and can manipulate, the stream it is inserted into. For input streams, there are also extraction operators that have a function parameter which will call the function with the stream object.

The architecture of C++ streams means that there is a buffer between the stream interface that you call in your code and the low-level infrastructure that obtains the data. The C++ Standard Library provides stream classes that have string objects as the buffer. For an output stream, you access the string after items have been inserted in the stream, which means that the string will contain those items formatted according to those insertion operators. Similarly, you can provide a string with formatted data as the buffer for an input stream, and when you use extraction operators to extract the data from the stream you are actually parsing the string and converting parts of the string to numbers.

In addition, stream classes have a locale object and stream objects will call the conversion facet of this locale to convert character sequences from one encoding to another.

Outputting floating point numbers

The <ios> library has manipulators that alter how streams handle numbers. By default, the output stream will print floating-point numbers in a decimal format for numbers in the range 0.001 to 100000, and for numbers outside this range it will use a scientific format with a mantissa and exponent. This mixed format is the default behavior of the defaultfloat manipulator. If you always want to use scientific notation, then you should insert the scientific manipulator into the output stream.

If you want to display floating point numbers using just the decimal format (that is the whole number on the left side of a decimal point and the factional part on the right side), then modify the output stream with the fixed manipulator. The number of decimal places can be altered by calling the precision method:

```
double d = 123456789.987654321;
cout << d << "n";
cout << fixed;
cout << d << "n";
cout.precision(9);
cout << d << "n";
cout << scientific;
cout << d << "n";
```

The output from the preceding code is:

```
1.23457e+08
123456789.987654
123456789.987654328
1.234567900e+08
```

The first line shows that scientific notation is used for large numbers. The second line shows the default behavior of fixed, which is to give the decimal number to 6 decimal places. This is changed in the code by calling the precision method to give 9 decimal places (the same effect can be achieved by inserting the setprecision manipulator in the <iomanip> library in the stream). Finally, the format is switched over to the scientific format with 9 decimal places to the mantissa from calling the precision method. The default is that the exponent is identified by the lowercase e. If you prefer, you can make this uppercase using the uppercase manipulator (and lowercase with nouppercase). Notice that the way that fractional parts are stored means that in fixed formats with 9 decimal places we see that the ninth digit is 8 rather than 1 as expected.
You can also specify whether a + symbol is shown for a positive number; the showpos manipulator will show the symbol, but the default noshowpos manipulator will not show the symbol. The showpoint manipulator will ensure that the decimal point is shown even if the floating-point number is a whole number. The default is noshowpoint, which means that, if there is no fractional part, no decimal point is displayed.

The setw manipulator (defined in the <iomanip> header) can be used with both integer and floating point numbers. In effect, this manipulator defines the minimum width of the space that the next (and only the next) item placed in the stream will occupy when printed on the console:

```
double d = 12.345678;
cout << fixed;
cout << setfill('#');
cout << setw(15) << d << "n";
```

To illustrate the effect of the setw manipulator, this code calls setfill manipulator, which indicates that instead of spaces a hash symbol (#) should be printed. The rest of the code says that the number should be printed using the fixed format (to 6 decimal places by default) in a space 15 characters wide. The result is:

```
######12.345678
```

If the number is negative (or showpos is used), then by default the sign will be with the number; if the internal manipulator (defined in <ios>) is used, then the sign will be leftjustified in the space set for the number:

```
double d = 12.345678;
cout << fixed;
cout << showpos << internal;
cout << setfill('#');
cout << setw(15) << d << "n";
```

The result of the preceding code is as follows:

```
+#####12.345678
```

Notice that the + sign to the right of the spaces is indicated by the pound symbol.
The setw manipulator is typically used to allow you to output tables of data in formatted

columns:

```
vector<pair<string, double>> table
{ { "one",0 },{ "two",0 },{ "three",0 },{ "four",0 } };
double d = 0.1;
for (pair<string,double>& p : table)
{
p.second = d / 17.0;
d += 0.1;
}
cout << fixed << setprecision(6);
```

```
for (pair<string, double> p : table)
{
cout << setw(6) << p.first << setw(10) << p.second << "n";
}
```

This fills a vector of pairs with a string and a number. The vector is initialized with the string values and a zero, then the floating-point number is altered in the for loop (the actual calculation is irrelevant here; the point is to create some numbers with multiple decimal places). The data is printed out in two columns with the numbers printed with 6 decimal places. This means that, including the leading zero and decimal point, each number will take up 8 spaces. The text column is specified as being 6 characters wide and the number column is specified as 10 characters wide. By default, when you specify a column width, the output will be right justified, meaning that each number is preceded by two spaces and the text is padded according to the length of the string. The output looks like this:

```
one 0.005882
two 0.011765
three 0.017647
four 0.023529
```

If you want the items in a column to be left justified, then you can use the left manipulator. This will affect all columns until the right manipulator is used to change the justification to right:

```
cout << fixed << setprecision(6) << left;
```

The output from this will be:

```
one 0.005882
two 0.011765
three 0.017647
four 0.023529
```

If you want different justification for the two columns, then you need to set the justification before printing a value. For example, to left justify the text and right justify the numbers, use this:

```
for (pair<string, double> p : table)
{
cout << setw(6) << left << p.first
<< setw(10) << right << p.second << "n";
}
```

The result of the preceding code is as follows:

```
one 0.005882
two 0.011765
three 0.017647
four 0.023529
```

Outputting integers

Integers can also be printed in columns using the setw and setfill methods. You can insert manipulators to print integers in base 8 (oct), base 10 (dec), and base 16 (hex). (You can also use the setbase manipulator and pass the base you want to use, but the only values allowed are 8, 10, and 16.) The number can be printed with the base indicated (prefixed with 0 for octal or 0x for hex) or without using the showbase and noshowbase manipulators. If you use hex, then the digits above 9 are the letters a to f, and by default these are lowercase. If you prefer these to be uppercase, then you can use the uppercase manipulator (and lowercase with nouppercase).

Outputting time and money

The put_time function in <iomanip> is passed a tm structure initialized with a time and date and a format string. The function returns an instance of the _Timeobj class. As the name suggests, you are not really expected to create variables of this class; instead, the function should be used to insert a time/date with a specific format into a stream. There is an insertion operator that will print a _Timeobj object. The function is used like this:

```
time_t t = time(nullptr);
tm *pt = localtime(&t);
cout << put_time(pt, "time = %X date = %x") << "n";
```

The output from this is:

```
time = 20:08:04 date = 01/02/17
```

The function will use the locale in the stream, so if you imbue a locale into the stream and then call put_time, the time/date will be formatted using the format string and the time/date localization rules for the locale. The format string uses format tokens for strftime:

```
time_t t = time(nullptr);
tm *pt = localtime(&t);
cout << put_time(pt, "month = %B day = %A") << "n";
cout.imbue(locale("french"));
cout << put_time(pt, "month = %B day = %A") << "n";
```

The output of the preceding code is:

```
month = March day = Thursday
month = mars day = jeudi
```

Similarly, the put_money function returns a _Monobj object. Again, this is simply a container for the parameters that you pass to this function and you are not expected to use instances of this class. Instead, you are expected to insert this function into an output stream. The actual work occurs in the insertion operator that obtains the money facet on the current locale, which uses this to format the number to the appropriate number of decimal places and determine the decimal point character; if a thousands separator is used, what character to use, before it is inserted it in the appropriate place.

```
Cout << showbase;
cout.imbue(locale("German"));
cout << "German" << "n";
cout << put_money(109900, false) << "n";
cout << put_money("1099", true) << "n";
cout.imbue(locale("American"));
cout << "American" << "n";
cout << put_money(109900, false) << "n";
cout << put_money("1099", true) << "n";
```

The output of the preceding code is:

```
German
1.099,00 euros
EUR10,99
American
$1,099.00
USD10.99
```

You provide the number in either a double or a string as Euro cents or cents and the put_money function formats the number in Euros or dollars using the appropriate decimal point (, for German, . for American) and the appropriate thousands separator (. for German, , for American). Inserting the showbase manipulator into the output stream means that the put_money function will show the currency symbol, otherwise just the formatted number will be shown. The second parameter to the put_money function specifies whether the currency character (false) or the international symbol (true) is used.

Converting numbers to strings using streams

Stream buffer classes are responsible for obtaining characters and writing characters from the appropriate source (file, console, and so on) and are derived from the abstract class basic_streambuf from <streambuf>. This base class defines two virtual methods, overflow and underflow, which are overridden by the derived classes to write and read characters (respectively) to and from the device associated with the derived class. The stream buffer class does the basic action of getting or putting items into a stream, and since the buffer handles characters, the class is templated with parameters for the character type and character traits.

As the name suggests, if you use a basic_stringbuf the stream buffer will be a string, so the source for read characters and the destination for written characters is that string. If you use this class to provide the buffer for a stream object, it means that you can use the insertion or extraction operators written for streams to write or read formatted data into or out of a string. The basic_stringbuf buffer is extendable, so as you insert items in the stream, the buffer will extend appropriately. There are typedef, where the buffer is a string (stringbuf) or a wstring (wstringbuf).

For example, imagine you have a class that you have defined and you have also defined an insertion operator so that you can use this with the cout object to print the value to the console:

```
struct point
{
double x = 0.0, y = 0.0;
point(){}
point(double _x, double _y) : x(_x), y(_y) {}
};
```

```
ostream& operator<<(ostream& out, const point& p)
{
out << "(" << p.x << "," << p.y << ")";
return out;
```

Using this with the cout object is simple--consider the following piece of code:

```
point p(10.0, -5.0);
cout << p << "n"; // (10,-5)
```

You can use the stringbuf to direct the formatted output to a string rather than the console:

```
stringbuf buffer;
ostream out(&buffer);
out << p;
string str = buffer.str(); // contains (10,-5)
```

Since the stream object handles the formatting it means that you can insert any data type for which there is an insertion operator, and you can use any of the ostream formatting methods and any of the manipulators. The formatted output from all of these methods and manipulators will be inserted into the string object in the buffer.

Another option is to use the basic_ostringstream class in <sstream>. This class is templated on the character type of the strings used as the buffer (so the string version is ostringstream). It is derived from the ostream class, so you can use instances wherever you would use an ostream object. The formatted results can be accessed through the str method:

```
ostringstream os;
os << hex;
os << 42;
cout << "The value is: " << os.str() << "n";
```

This code obtains the value of 42 in hexadecimal (2a); this is achieved by inserting the hex manipulator in the stream and then inserting the integer. The formatted string is obtained by calling the str method.

Reading numbers from strings using streams

The cin object, an instance of the istream class (in the <istream> library), can input characters from the console and convert them to the numeric form you specify. The ifstream class (in the <ifstream> library) will also allow you to input characters from a file and convert them to numeric form. As with outputting streams, you can use the stream classes with a string buffer so that you can convert from a string object to a numeric value. The basic_istringstream class (in the <sstream> library) is derived from the basic_istream class, so you can create stream objects and extract items (numbers and strings) from these objects. The class provides this stream interface on a string object (the typedefs keyword istringstream is based on a string and wistringstream is based on a wstring). When you construct objects of this class you initialize the object with a string containing a number and then you use the >> operator to extract objects for the fundamental built-in types, just as you would extract those items from the console using cin.

It is important to reiterate that the extraction operators treat whitespaces as the separators between items in a stream, and hence they will ignore all leading whitespaces, read the nonwhitespaces
characters up to the next whitespaces and attempt to convert this substring into the appropriate type as follows:

```
istringstream ss("-1.0e-6");
double d;
ss >> d;
```

This will initialize the d variable with the value of -1e-6. As with cin, you have to know the format of the item in the stream; so if, instead of extracting a double from the string in the preceding example, you try to extract an integer, the object will stop extracting characters when it comes to the decimal point. If some of the string is not converted, you can extract the rest into a string object:

```
istringstream ss("-1.0e-6");
int i;
ss >> i;
string str;
ss >> str;
cout << "extracted " << i << " remainder " << str << "n";
```

This will print the following at the console:

```
extracted -1 remainder .0e-6
```

If you have more than one number in the string you can extract these with several calls to the >> operator. The stream also supports some manipulators. For example, if the number in the string is in hex format you can inform the stream that this is the case using the hex manipulator as follows:

```
istringstream ss("0xff");
```

```
int i;
ss >> hex;
ss >> i;
```

This says that the number in the string is in hexadecimal format and the variable i will be initialized with a value of 255. If the string contains non-numeric values, then the stream object will still try to convert the string to the appropriate format. In the following snippet you can test if such an extraction fails by calling the fail function:

```
istringstream ss("Paul was born in 1942");
int year;
ss >> year;
if (ss.fail()) cout << "failed to read number" << "n";
```

If you know that the string contains text, you can extract it into string objects, but bear in mind that whitespace characters are treated as delimiters:

```
istringstream ss("Paul was born in 1942");
string str;
ss >> str >> str >> str >> str;
int year;
ss >> year;
```

Here, there are four words before the number, so the code reads a string four times. If you don't know where in the string the number is but you know there is a number in the string, you can move the internal buffer pointer until it points to a digit:

```
istringstream ss("Paul was born in 1942");
string str;
while (ss.eof() && !(isdigit(ss.peek()))) ss.get();
int year;
ss >> year;
if (!ss.fail()) cout << "the year was " << year << "n";
```

The peek method returns the character at the current position, but does not move the buffer pointer. This code checks to see if this character is a digit, and if not, the internal buffer pointer is moved by calling the get method. (This code tests the eof method to ensure that there is no attempt to read a character after the end of the buffer.) If you know where the number starts then you can call the seekg method to move the internal buffer pointer to a specified position.

The <istream> library has a manipulator called ws that removes whitespace from a stream. Recall earlier that we said that there is no function to remove whitespace from a string. This is true because the ws manipulator removes whitespace from a *stream* and not from a *string*, but since you can use a string as the buffer for a stream it means that you can use this function to remove white space from a string indirectly:

```
string str = " hello ";
cout << "|" << str1 << "|n"; // | hello |
istringstream ss(str);
ss >> ws;
string str1;
ss >> str1;
ut << "|" << str1 << "|n"; // |hello|
```

The ws function essentially iterates through the items in the input stream and returns when a character is not whitespace. If the stream is a file or the console stream then the ws function will read in the characters from those streams; in this case, the buffer is provided by an already-allocated string and so it skips over the whitespaces at the beginning of the string. Note that stream classes treat subsequent whitespaces as being separators between values in the stream, so in this example the stream will read in characters from the buffer until there is a whitespace, and will essentially *left-and right-trim* the string. However, this is not necessarily what you want. If you have a string with several words padded by whitespace, this code will only provide the first word.

The get_money and get_time manipulators in the <iomanip> library allow you to extract money and time from strings using the money and time facets for a locale:

```
tm indpday = {};
string str = "4/7/17";
istringstream ss(str);
ss.imbue(locale("french"));
ss >> get_time(&indpday, "%x");
if (!ss.fail())
{
cout.imbue(locale("american"));
cout << put_time(&indpday, "%x") << "n";
}
```

In the preceding code, the stream is first initialized with a date in the French format (day/month/year) and the date is extracted with get_time using the locale's standard date representation. The date is parsed into a tm structure, which is then printed out in standard date representation for the American locale using put_time. The results is:

```
7/4/2017
```