

## CHAPTER 1

**Essential C++**

**T**his chapter covers the fundamental requirements necessary to allow the reader to get up and running building quantitative models using the C++ programming language. This introduction is in no way intended to be an in-depth treatment of the C++ programming language but more an overview of the basics required to build your own efficient and adaptable programs. Once the key concepts have been developed, object-oriented principles are introduced and many of the advantages of building quantitative systems using such programming approaches are outlined. It is assumed that the reader will have some prerequisite knowledge of a low-level programming language and the necessary computation skills to effectively grasp and apply the material presented here.

**1.1 A BRIEF HISTORY OF C AND C++**

C is a *procedural*<sup>1</sup> programming language developed at Bell Laboratories between 1969 and 1973 for the UNIX operating system. Early versions of C were known as K&R C after the publication of the book *The C Programming Language* written by Brian Kernighan and Dennis Ritchie in 1978. However, as the language developed and became more standardised, a version known as ANSI<sup>2</sup> C became more prominent. Although C is no longer the choice of many developers, there is still a huge amount of *legacy* software coded in it that is actively maintained. Indeed, C has greatly influenced other programming languages, in particular C++ which began purely as an extension of C.

<sup>1</sup> *Procedural* programming is a form of *imperative programming* in which a program is built from one or more procedures i.e. subroutines or functions.

<sup>2</sup> Founded in 1918, the *American National Standards Institute* (ANSI) is a private, non-profit membership organisation that facilitates the development of *American National Standards* (ANS) by accrediting the procedures of the *Standards Developing Organizations* (SDOs). These groups work cooperatively to develop voluntary national consensus standards.

Often described as a *superset* of the C language, C++ uses an entirely different set of programming concepts designed around the *Object-Oriented Programming* (OOP) paradigm. Solving a computer problem with OOP involves the design of so-called *classes* that are abstractions of physical objects containing the state, members, capabilities and methods of the object. C++ was initially developed by Bjarne Stroustrup in 1979 whilst at Bell Laboratories as an enhancement to C; originally known as C with Classes. The language was renamed C++ in the early 80s and by 1998, C++ was standardised as ANSI/ISO<sup>3</sup> C++. During this time several new features were added to the language, including virtual functions, operator overloading, multiple inheritance and exception handling. The ANSI/ISO standard is based on two main components: the *core language* and the *C++ Standard Library* that incorporates the C Standard Library with a number of modifications optimised for use with the C++ language. The C++ Standard Library also includes most of the *Standard Template Library* (STL); a set of tools, such as *containers* and *iterators* that provide array-like functionality, as well as *algorithms* designed specifically for sorting and searching tasks. C++11 is the most recent *complete* overhaul of the C++ programming language approved by ANSI/ISO on 12 August 2011, replacing C++03, and superseded by C++14 on 18 August 2014. The naming convention follows the tradition of naming language versions by the year of the specification's publication, although it was formerly known as C++0x to take into account many publication delays. C++14 is the informal name for the most recent revision of the C++ ANSI/ISO standard, intended to be a small extension over C++11, featuring mainly bug fixes and small syntax improvements.

## 1.2 A BASIC C++ PROGRAM

Without doubt the best method of learning a programming language is to actually start by writing and analysing programs. Source 1.1 implements a basic C++ program that simply outputs a string of text, once the program has been compiled and executed, to the console window. Although the program looks very simple it nevertheless contains many of the fundamental components that every C++ program generally requires.

### SOURCE 1.1: A BASIC C++ PROGRAM

```
// main.cpp
#include <windows.h>
#include <iostream>
```

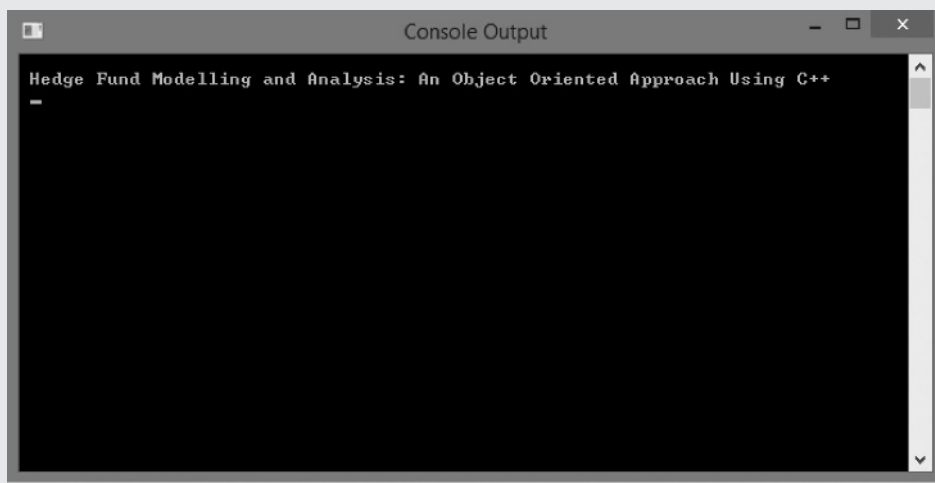
<sup>3</sup>The *International Organisation for Standardisation* (ISO) is an international standard-setting body made up of representatives from a range of *National Standards Organisations* (NSOs).

```
using std::cout;
using std::cin;

int main()
{
    SetConsoleTitle(L"Console Output"); // Set title of console
    window

    cout << "\n " << "Hedge Fund Modelling and Analysis: An Object
    Oriented Approach Using C++";

    cin.get(); // Pause console window
    return 0; // Return null integer and exit
}
```



Statements beginning with a hash symbol (#) indicate *directives* to the *preprocessor* that initialise when the compiler is first invoked, in this case, to inform the compiler that certain functions from the C++ Standard Library must be included. `#include <windows.h>` gives the program access to certain functions in the library, such as `SetConsoleTitle()` whilst `#include <iostream>` enables console input and output (I/O). Typical objects in the `iostream` library include `cin` and `cout` which are explicitly included through the `using` statement at the top of the program. Writing `using std::cout` at the top of the program avoids the need to keep retyping `std` through the *scope resolution operator* (`::`) every time `cout` is used. For example, if we had not specified `using std::cout` we would have to explicitly write `std` in front of each usage throughout the program, that is:

```
std::cout << "\n " << "Hedge Fund Modelling and Analysis: An Object
Oriented Approach Using C++";
std::cin.get();
```

Although in this case there are only two occasions where we need `std`, you can imagine how this could quickly clog up code for very large programs. Note also that all C++ statements must end with a semi-colon (;).

A commonly identified problem with the C language is the issue of running out of names for definitions and functions when programs reach very large sizes eventually resulting in name clashes. Standard C++ has a mechanism to prevent such a clash through the use of the `namespace` keyword. Each set of C++ definitions in a library or program is *wrapped* into a namespace, and if some other definition has an identical name, but is in a different namespace, then there is *no* conflict. All Standard C++ libraries are wrapped in a single namespace called `std` and invoked with the `using` keyword:

```
using namespace std;
```

Whether to use `using namespace std` or explicitly state their use through `using std::cout`, for example, is purely a preference of programming style. The main reason we do not invoke `using namespace std` in our programs is that this leaves us the opportunity of defining our own namespaces if we wish and it is generally good practice to have only one namespace invocation in each program.

The `main()` function is the point at which all C++ programs start their execution even if there are several other functions declared in the same program. For this reason, it is an essential requirement that all C++ programs have a `main()` function within the body at some point in the program. Once the text is output to the console window, `cin.get()` is used to cause the program to pause so that the user can read the output and then close and exit the window by pressing any key. Technically, in C or C++ the `main()` function must return a value because it is declared as `int` i.e. the main function should return an integer data type. The `int` value that `main()` returns is usually the value that will be passed back to the operating system; in this case it is 0 i.e. `return 0` which indicates that the program ran successfully. It is not necessary to state `return 0` explicitly, because the compiler invokes this automatically when `main()` terminates, but it is good practice to include a return type for all functions (including `main()`).

### 1.3 VARIABLES

A variable is a name associated with a portion of memory used to *store* and *manipulate* the data associated with that variable. The compiler sets aside a specific amount of memory space to store the data *assigned* to the variable and associates the variable name with that memory *address*. As the name implies, variables can be changed within a program as and when required. When new data is assigned to the same variable, the old data is *overwritten* and restored in the same memory address. The data stored in a

**TABLE 1.1** Reserved C++ keywords

asm, auto, bool, break, case, catch, char, class, const, const_cast, continue, default, delete, do, double, dynamic_cast, else, enum, explicit, export, extern, false, float, for, friend, goto, if, inline, int, long, mutable, namespace, new, operator, private, protected, public, register, reinterpret_cast, return, short, signed, sizeof, static, static_cast, struct, switch, template, this, throw, true, try, typedef, typeid, typename, union, unsigned, using, virtual, void, volatile, wchar_t, while
---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

variable is only *temporary* and only exists as long as the variable itself exists (defined by the *scope* of the variable). If the data stored in a variable is required beyond its existence then it must be written to a *permanent* storage device, such as a disk or file.

A variable name can be any length and composed of lower and upper case letters, numbers and the underscore ( `_` ) character, but keep in mind that variables are *case-sensitive*. In practice, a programmer will usually develop their own variable naming convention but bear in mind that C++ reserves certain keywords for variable names so try not to clash with these. Table 1.1 shows a list of reserved C++ keywords.

There are several built-in *data types* provided by C++ along with specific *type modifiers* to further quantify the data. A complete list of all the data types and their associated modifiers are described in Table 1.2.

In Table 1.2, other than `char` (which has a size of exactly one byte), none of the fundamental types has a standard size (only a minimum size, at most). This does not mean that these types are of an undetermined size, but that there is no *standard size* across all compilers and machines; each compiler implementation can specify the sizes that best fit the architecture where the program is going to be executing. This rather generic size specification of data types allows the C++ language a lot of flexibility in adapting to work optimally on all kinds of platforms, both present and future.

### 1.3.1 Characters and Strings

When using the `char` data type, we use single quotes, for example:

```
char Stock = 'MSFT';
```

Certain characters, such as single ( `' '` ) and double ( `" "` ) quotes have special meaning in C++ and have to be treated with care. In addition, C++ reserves special characters for formatting text and other processing tasks known as *character escape sequences* (or *backslash character constants*) as shown in Table 1.3.

A more versatile data type than `char` is `string` which can be a combination of characters, numbers, spaces and symbols of any length. C++ does not have a built-in data type to hold *strings* instead it is defined in the C++ Standard Library through the inclusion of the header file `<string>`. An example of using `string` variables is shown in Source 1.2.

**TABLE 1.2** Common C++ data types

Name	Description	Size (Bytes)	Range
char	Character	1	-128 to 127
unsigned char		1	0 to 255 (ASCII characters)
signed char		1	-128 to 127 (ASCII characters)
int	Integer number	4	-2,147,483,648 to 2,147,483,647
unsigned int		4	0 to 4,294,967,295
signed int		4	-2,147,483,648 to 2,147,483,647
short int		2	-32,768 to 32,767
unsigned short int		2	0 to 65,535
signed short int		2	-32,768 to 32,767
long int		4	Same as int
unsigned long int		4	Same as unsigned int
signed long int		4	Same as signed int
float	Floating point number	4	3.4E-38 to 3.4E+38
double	Double precision floating point number	8	1.7E-308 to 1.7E+308
long double		10	3.4E-4932 to 1.1E+4932
bool	Boolean value	1	True or False
string	As required		Any length
wchar_t	Wide character	2	0 to 65,535

**TABLE 1.3** Character escape sequences

Sequence	Output
\n	New line
\t	Tab
\b	Back space
\?	Question mark
\f	Page feed
\a	Alert (beep)
\\	Backslash
\'	Single quote
\"	Double quote

**SOURCE 1.2: STRING VARIABLES**

```
// main.cpp
#include <windows.h>
#include <iostream>
#include <string>
using std::cout;
using std::cin;
using std::string;

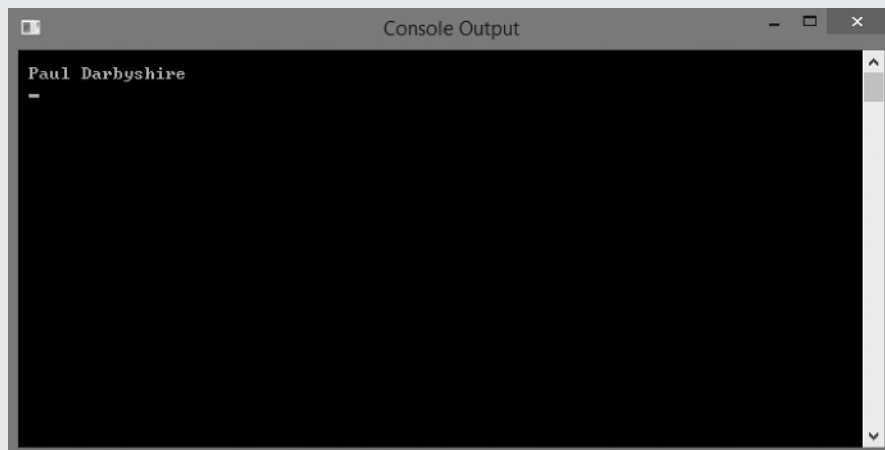
int main()
{
    SetConsoleTitle(L"Console Output"); // Set title of console
    window

    //declare two string variables
    string strFirstName = "Paul";
    string strLastName = "Darbyshire";

    //concatenate the two strings
    string strFullName = strFirstName + " " + strLastName;

    cout << "\n " << strFullName;

    cin.get(); // Pause console window
    return 0; // Return null integer and exit
}
```



In Source 1.2, two `string` variables are declared and initialised and then joined together to form another string. The `(+)` symbol is used for joining (or *concatenating*) two variables together, and in this context the `(+)` symbol is often referred to as the *concatenation operator*.

### 1.3.2 Variable Declarations

Before a variable can be used in a program it must first be *declared* as shown in Source 1.3. Declaring the variable and its data type allows the compiler to set aside the appropriate amount of memory for storage and subsequent manipulation.

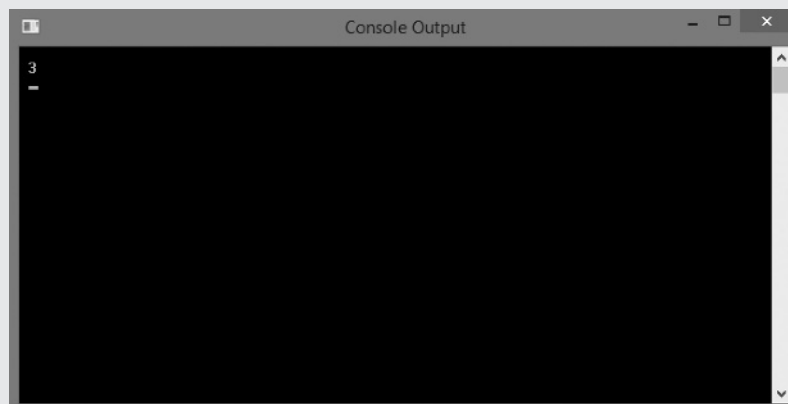
#### SOURCE 1.3: DECLARING VARIABLES

```
// ...

// Declare variables
int x, y;
int result;
// Assign values
x = 4;
y = 2;
x = x + 1;
//Do something
result = x - y;

cout << "\n " << result << "\n ";

// ...
```





It is possible to declare more than one variable of the *same* type in the same declaration statement. It is also possible to assign initial values to variables whilst they are being declared through the process of *initialisation*, for example:

```
int x, y = 4, z = 3;
```

There is another useful method of initialising a variable known as *constructor initialisation*:

```
int x(0);
```

### 1.3.3 Type Casting

One way to force an expression to produce a result that is of a different type to the variables declared in the expression is to use a construct called `cast` (i.e. *type casting*). Source 1.4 shows an example of declaring two variables as `int` and dividing them to produce an `int` and double division through type casting.

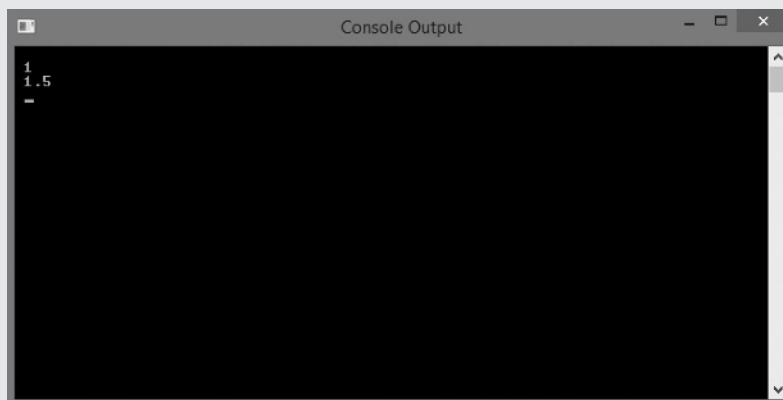
#### SOURCE 1.4: TYPE CASTING

```
// ...

int a = 6, b = 4;

cout << "\n " << a/b << "\n"; // Integer division
cout << " " << (double)a/b << "\n "; // Type casting to double
division

// ...
```



Note that type casting will not change the type of the variables from integer only the type of the result to double.

### 1.3.4 Variable Scope

A variable can have either *global* (i.e. *public*) or *local* (i.e. *private*) scope depending on where it is declared within the program. Any variables declared with global scope should be prefixed with the keyword `const`. An example is shown in Source 1.5.

#### SOURCE 1.5: VARIABLE SCOPE

```
// ...

// GLOBAL variable
int globalN = 144;

int main()
{
    SetConsoleTitle(L"Console Output"); // Set title of console
    window

    // LOCAL variable
    int localN = 72;

    cout << "\n " << "# of data points (LOCAL) = " << localN;
    cout << "\n " << "# of data points (GLOBAL) = " << globalN;

    cin.get(); // Pause console window
    return 0; // Return null integer and exit
}
```



In Source 1.5, you can see that the variable `globalN` has been declared globally and initialised to the value 144. Global variables can be accessed from anywhere in the program once they have been declared. Local variables, on the other hand, such as `localN` can only be used within the block enclosed by the braces (`{}`) in which it is declared.

### 1.3.5 Constants

*Constants* are fixed values assigned to variables that cannot be changed once they have been declared and initialised. We have already used *literal constants* when a variable was declared and initialised in Source 1.2:

```
string FirstName = "Paul";
```

Or, as in Source 1.5:

```
int localN = 72;
```

With *symbolic constants* the `const` keyword is used in front of the declaration and initialisation, for example:

```
const double Volatility = 0.18;
```

*Enumerated constants* are an alternative way of creating a series of integer constants. Suppose you wanted to assign an integer value of 0 to 6 to the days of the week starting at Sunday. This could be achieved using a list of symbolic constants written as:

```
const int Sun = 0;  
const int Mon = 1;  
const int Tue = 2; etc.
```

However, with enumeration it is possible to write:

```
enum WeekDays  
{  
    Sun,  
    Mon,  
    Tue,  
    Wed,  
    Thu,  
    Fri,  
    Sat  
};
```

If each week day is not explicitly initialised, they are *automatically* assigned the values 0, 1, 2, 3, etc., starting with the variable `Sun`. Note that the default value starts at 0 and not 1. Alternatively, it is possible to initialise one or more of the variables to any integer value, for example:

```
enum WeekDays
{
    Sun = 10,
    Mon,
    Tue,
    Wed = 6,
    Thu,
    Fri,
    Sat
};
```

Variables that are not explicitly initialised are given initial values counting upwards from the preceding initialised variable i.e. `Sun = 10`, `Mon = 11`, `Tue = 6`, `Wed = 7`, `Thu = 8` and so on.

## 1.4 OPERATORS

*Operators* are used to perform a specific operation on a set of operands in an expression. Operators can be of two types:

*Unary* – take only one argument and

*Binary* – take two arguments.

### 1.4.1 The Assignment Operator

The *assignment operator* simply assigns a value to a variable, for example:

```
x = 4;
```

The statement above assigns to the variable `x` the value 4. Note that the assignment operator always reads from *right* -> *left*, and never the other way around. The following statement is valid in C++:

```
x = y = z = 3;
```

In this statement, the value 3 is assigned to all three variables `x`, `y` and `z`. Expressions that are evaluated within the assignment operator, such as:

```
x = x + 1;
```

are read as ‘*add the value 1 to x and assign this value to x*’ i.e. increase the value of x by 1. Indeed, this statement can also be written in a more compact form using *compound assignment operators*, written as:

```
x += 1;
```

In this statement, as before, x is simply *incremented* by 1 and the value assigned to x. Alternatively, if x is *decremented* by 1 we can write:

```
x -= 1;
```

The *increase* (++) and *decrease* (--) operators can also be used to get the same result as above, that is:

```
x++; and x--;
```

An interesting characteristic of the increase and decrease operators is that they can be used either *prefix* or *postfix*. If the increase operator is used as a prefix i.e. ++x; the value of x is increased *before* the expression is evaluated. When used as a postfix i.e. x++; the value of x is increased *after* being evaluated. Source 1.6 shows an example of the prefix and postfix increase operators.

#### SOURCE 1.6: PREFIX AND POSTFIX OPERATORS

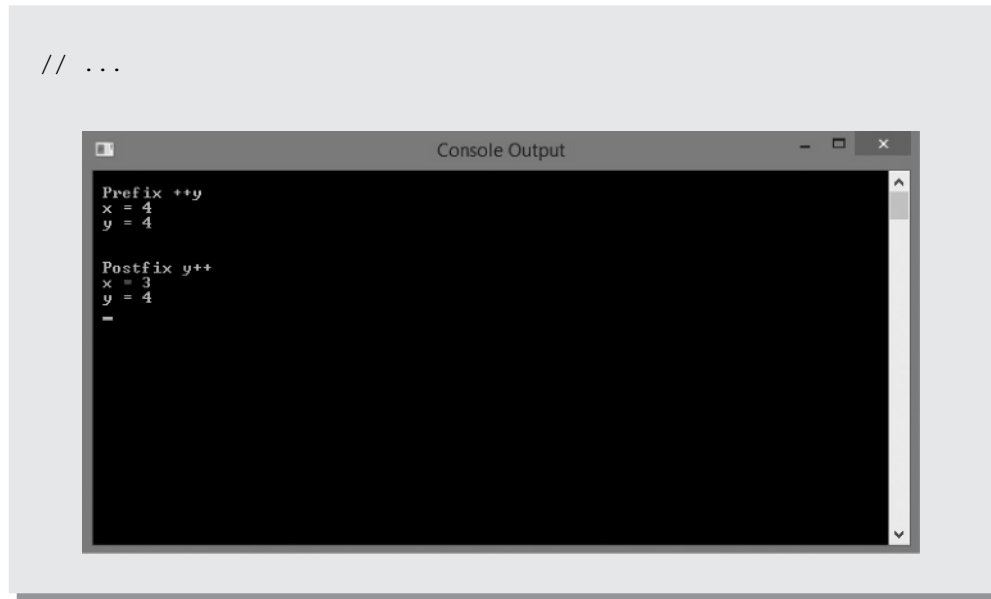
```
// ...

// Declare variable
int y = 3;

// Prefix
cout << "\n " << "Prefix ++y" << "\n ";
cout << "x = " << ++y << "\n ";
cout << "y = " << y << "\n\n";

// Reset y
y = 3;

// Postfix
cout << "\n " << "Postfix y++" << "\n ";
cout << "x = " << y++ << "\n ";
cout << "y = " << y << "\n ";
```



### 1.4.2 Arithmetic Operators

There are five basic C++ *arithmetic operators* as shown in Table 1.4. The only one that may not be familiar is the *modulo operator* (%) used for determining the remainder of *integer* division as shown in Source 1.7.

**TABLE 1.4** Arithmetic operators

Operator	Name
+	Addition
-	Subtraction
*	Multiplication
\	Division
%	Modulo

#### SOURCE 1.7: MODULO OPERATOR

```
// ...

// Declare variables
int x = 11, y = 3;
```

```
cout << "\n " << "Remainder of 11 divided by 3 = " << x%y << "\n ";  
  
// ...
```



### 1.4.3 Relational Operators

Sometimes it is necessary to test the relationship between two expressions so that some action can be performed based on the outcome of the result. *Relational operators* can be used to perform such tasks. The most common C++ relational operators are shown in Table 1.5.

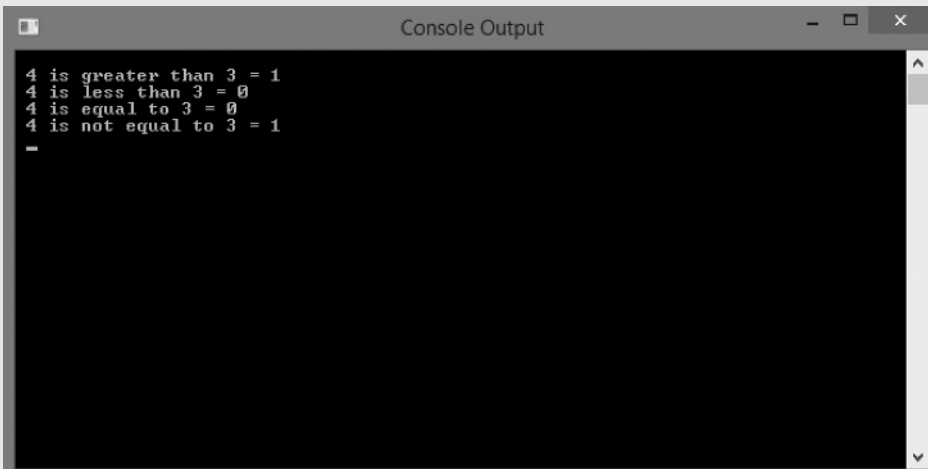
The only result of a relational operator expression when evaluated is either 1 (*true*) or 0 (*false*) i.e. a *Boolean* value. C++ automatically converts a Boolean value to an integer as shown in Source 1.8.

**TABLE 1.5** Common relational operators

Operator	Name
==	Equal to
!=	Not equal to
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to

**SOURCE 1.8: RELATIONAL OPERATORS**

```
// ...  
  
cout << "\n " << "4 is greater than 3 = " << (4 > 3) << "\n ";  
cout << "4 is less than 3 = " << (4 < 3) << "\n ";  
cout << "4 is equal to 3 = " << (4 == 3) << "\n ";  
cout << "4 is not equal to 3 = " << (4 != 3) << "\n ";  
  
// ...
```



In Source 1.8, the relational expressions are enclosed in parentheses so that C++ evaluates the relationship first and then sends the result to `cout`. Note that the relation operator to test for *equality* is `==` and not a single `=` which refers instead to the assignment of a value to a variable.

**1.4.4 Logical Operators**

In order to test for more complex expressions, *logical operators* can be combined with relational operators. Logical operators are normally associated with *Boolean algebra*<sup>4</sup> and as such, produce a Boolean result. The three most common C++ logical operators are shown in Table 1.6.

<sup>4</sup> *Boolean algebra* is a logical calculus of truth values 0 and 1 developed by George Boole in the 1840s.



**TABLE 1.6** Common logical operators

Operator	Boolean Operation
&&	AND
	OR
!	NOT

For example, suppose you wanted to select only those equities that had a price (Price) above P and market capitalisation (MarketCap) above MC. This could be expressed in C++ as follows:

```
(Price > P) && (MarketCap > MC)
```

#### 1.4.5 Conditional Operator

The *conditional operator* (?) can be used in the following shorthand format:

```
(condition) ? result1 : result2
```

Which is read '*if condition is true the expression returns result1, if it is not it will return result2*'. Source 1.9 shows a typical example of using the conditional operator in this format.

#### **SOURCE 1.9: THE CONDITIONAL OPERATOR (?)**

```
// ...

// Declare variables
int a, b, c;
// Assign values to a and b
a = 4;
b = 6;
// Use conditional expression
c = (a > b) ? a : b;

cout << "\n " << "Conditional expression gives " << c << "\n ";

// ...
```



In Source 1.9, `a` and `b` are assigned the values of 4 and 6, respectively, which makes the expression `(a > b)` evaluate to false, thus the first value after the question mark is ignored and the second value (after the colon) accepted; resulting in the value of 6 for the conditional expression.

## 1.5 INPUT AND OUTPUT

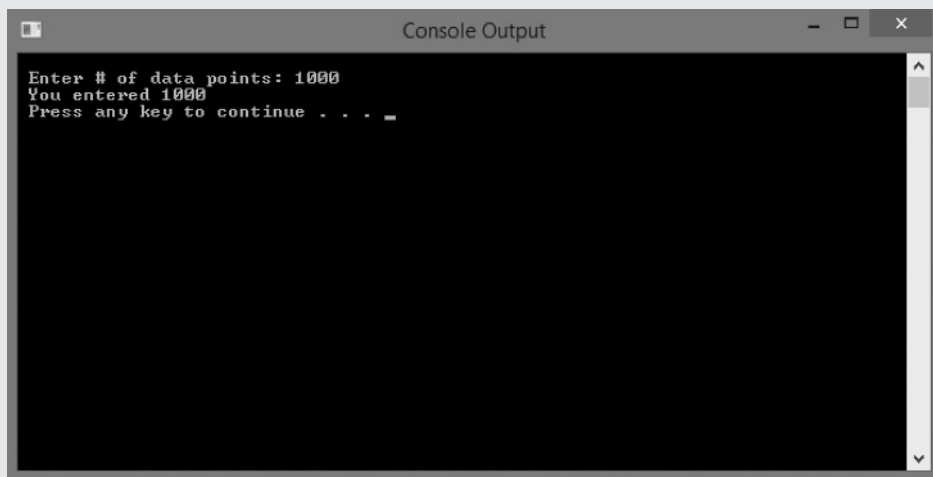
For the majority of cases, programs will require *inputs* from the keyboard and *outputs* to the console window. We have already encountered the `<iostream>` header file from the C++ Standard Library which allows us to handle I/O in our programs. As we have already seen in all of the above programs, output to the console window is handled with `cout` along with the *insertion operator* (`<<`). Inputs from the keyboard are handled by `cin` along with the *extraction operator* (`>>`). Source 1.10 shows an example of using both `cout` and `cin`.

### SOURCE 1.10: INPUT AND OUTPUT

```
// ...  
  
// Declare variable  
int n;  
// Get value from keyboard  
cout << "\n " << "Enter # of data points: ";
```

```
// Assign value to a
cin >> n;
// Output result
cout << " " << "You entered " << n << "\n ";

system("PAUSE"); // Pause console window
return 0; // Return null integer and exit
}
```



In Source 1.10, an integer `a` holds the value entered by the user when prompted by the `cout` statement. Once the user inputs a value and presses `Enter` the value is stored in `a` and subsequently output to the screen through the second `cout` statement. `cin` can only store the value into `a` once the `Enter` key has been pressed. It is possible to allow the user to input several values when prompted by using the concept of *chaining* as shown in Source 1.11.

### SOURCE 1.11: CHAINING

```
// ...

// Declare variables
int a, b;
// Get values from keyboard
```

```
cout << "\n " << "Enter two values: ";  
// Assign value to a and b using chaining  
cin >> a >> b;  
  
cout << " " << "The two values were " << a << " and " << b << "\n ";  
  
// ...
```



In Source 1.11:

```
cin >> a >> b;
```

is equivalent to the two statements:

```
cin >> a;  
cin >> b;
```

In both cases the user must input two values before pressing Enter for the program to continue. Note that in Source 1.10 and 1.11 I used `system("PAUSE")` to hold the console window instead of `cin.get()` so as not to confuse the compiler when using console input within the main body of the program.

## 1.6 CONTROL STRUCTURES

---

It is possible to *control* the order of execution of statements in a program through two special types of structure, namely *branching* and *looping*.

### 1.6.1 Branching

The most common type of branching statement is the *decision-making* (or *selection*) structure. Decision-making structures control program execution through the dependence on one or more specified conditions being satisfied. The `if` structure is by far the most popular type of decision-making structure and there are two basic forms, namely:

```
if (condition) statement;
and
if (condition)
{
    statement;
}
```

The `if ... else` and `if ... else if` structures are extensions of the `if` structure that allows further conditions to be tested, that is:

```
if (condition)
{
    statement1;
}
else
{
    statement2;
}
and
if (condition)
{
    statement1;
}
else if
{
    statement2;
}
else
{
    statement3;
}
```

By mixing relational and logical operators it is possible to create increasingly complex decision-making structures. The `if` structure can also be *nested* (or *embedded*) within a set of `if` structures. Source 1.12 and 1.13 show examples of using the `if` and `if ... else if` structures.

### SOURCE 1.12: IF ... ELSE STATEMENT

```
// ...

int x;

cout << "\n " << "Enter a number: ";
cin >> x;

//if...else statement
if (x > 100)
{
    cout << "\n " << "Number greater than 100" << "\n ";
}
else
{
    cout << " " << "Number less than 100" << "\n ";
}

// ...
```



**SOURCE 1.13: IF ... ELSE ... IF STATEMENT**

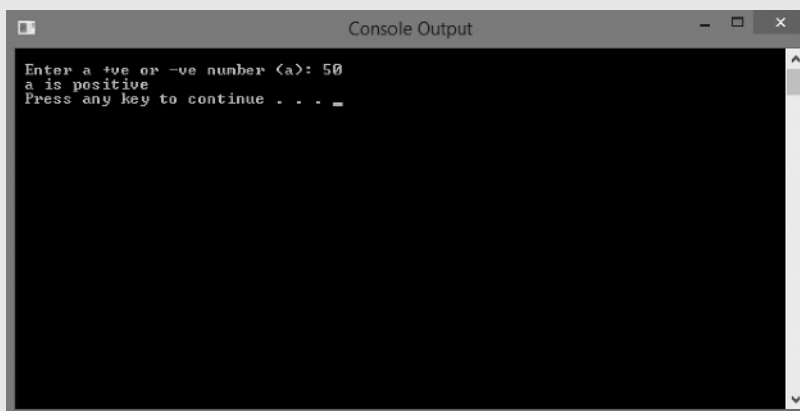
```
// ...

int a;

cout << "\n " << "Enter a +ve or -ve number (a): ";
cin >> a;

// if ... else if statement
if (a > 0)
{
    cout << " " << "a is positive" << "\n ";
}
else if (a < 0){
    cout << " " << "a is negative" << "\n ";
}
else
{
    cout << " " << "a is 0" << "\n ";
}

// ...
```



Another type of decision-making structure in C++ is the `switch` statement that allows the execution of different sets of statements depending on the value of one expression. The syntax for the `switch` structure is as follows:

```
Switch (expression)
{
Case constant1:
statement1;
break;
Case constant2:
statement2;
break;
.
.
.
default:
statement;
}
```

The switch statement works in the following way:

switch evaluates expression and checks to see if it is equivalent to constant1, if it is, execute statement1 until it reaches break. When the break statement is reached the program jumps to the end of the switch structure. If expression is not equal to constant1 it is checked against constant2 and if it is equal to this, the program will execute statement2 until break is reached when it then jumps to the end of the switch structure. If expression does not match any of the constants, the program executes the statement after default, if it exists (optional).

Unlike if structures that can test for a variety of conditions, switch structures can only test for equality. Also, only constants can follow the case statement and not variables or expressions. Source 1.14 shows an example of using the switch structure.

#### **SOURCE 1.14: THE switch STATEMENT**

```
int n;
...
cout << "\n " << "Choose 1, 2, 3 or 4: ";
cin >> n;

// Switch statement
switch (n)
{
    case 1:
        cout << " " << "You chose 1" << "\n ";
        break;
```



```
case 2:
    cout << " " << "You chose 2" << "\n ";
    break;
case 3:
    cout << " " << "You chose 3" << "\n ";
    break;
case 4:
    cout << " " << "You chose 4" << "\n ";
    break;
default:
    cout << " " << "1, 2, 3, or 4 not chosen!" << "\n ";
}

// ...
```



### 1.6.2 Looping

In C++, *looping* involves using *iteration* structures in which a particular statement is repeated a certain number of times, or, while a condition is satisfied.

### 1.6.3 The for Loop

The for loop performs a repetitive task with a counter which is initialised and changes on each iteration. The general syntax of the for loop is as follows:

```
for (initialisation; condition; action) statement;
```

Alternatively, the `for` loop with a statement block is written as follows:

```
for (initialisation; condition; action)
{
    statements;
}
```

In general, the *action* involves *incrementing* or *decrementing* the value of the counter. Source 1.15 shows an example of using the `for` loop.

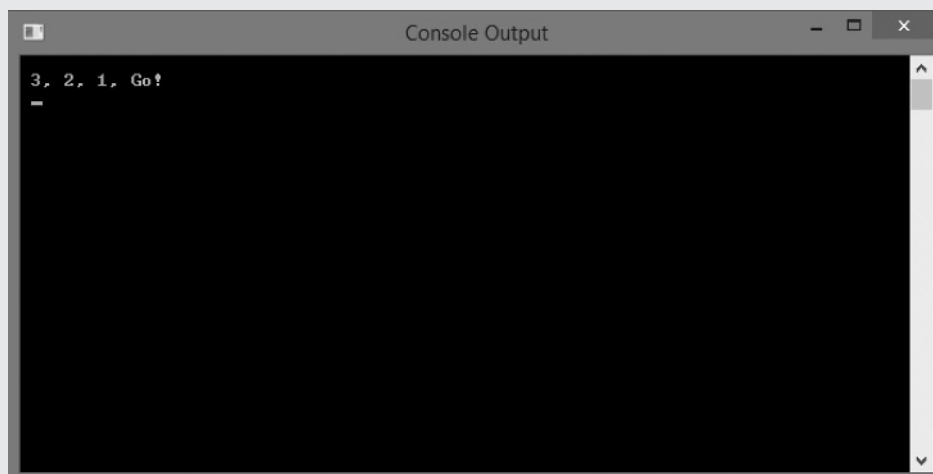
### SOURCE 1.15: THE FOR LOOP

```
// ...

cout << "\n";

// for loop
for (int i = 3; i > 0; i--)
{
    cout << " " << i << ", ";
}
cout << " " << "Go!" << "\n ";

// ...
```



It is possible to specify more than one expression in any of the fields in the parentheses using the *comma operator* (,) which acts as a separator for more than one expression. For example, suppose you wanted to initialise more than one variable in the `for` loop, that is:

```
for (n = 0, m = 10; n! = m; n++, m--)  
{  
    statements;  
}
```

In this case, `n` and `m` are initialised with a value of 0 and 10, respectively. Since `n` is incremented by one and `m` decremented by one after each iteration, `n! = m` (i.e., `n` not equal to `m`) condition will become false after the 10<sup>th</sup> iteration when both `n` and `m` equal 10. Also note that any of the fields inside the parentheses of the `for` loop can be omitted although there must be a semi-colon (;) in their place. For example:

```
for (; i < 100; i++) statement;
```

is perfectly valid if there is no need to initialise the counter `i`.

#### 1.6.4 The while Loop

The syntax for the `while` loop is written as:

```
while (condition) statement;
```

Alternatively, the `while` loop with a statement block is written as:

```
while (condition)  
{  
    statements;  
}
```

The `while` loop repeats the statement while the condition is true, when the condition is false, looping ends. A key difference between the `while` and `for` loops is that the former does not require an initialisation (or action) in the loop structure. Source 1.16 shows an example of using the `while` loop.

#### SOURCE 1.16: THE WHILE LOOP

```
// ...  
  
int n;  
  
cout << "\n " << "Enter countdown number: ";
```

```
cin >> n;
cout << " ";

// while loop
while (n > 0)
{
    cout << n << ", ";
    -n;
}
cout << "Go!" << "\n ";

// ...
```



Source 1.16 can be interpreted in the following steps:

- Step 1: User assigns a value to integer `n`.
- Step 2: The while condition `(n > 0)` is checked:
  - true: go to Step 3.
  - false: go to Step 5.
- Step 3: Execute statement block.
- Step 4: Return to Step 2.
- Step 5: End program.

Note that the `while` loop must end at some point, therefore a provision must be made inside the statement block to force the condition to become `false`. In Source 1.16, `n` is decremented by 1 after each loop until it reaches 0 when the condition `n > 0` is no longer satisfied and the loop is forced to end.

### 1.6.5 The `do ... while` Loop

The `do ... while` loop is exactly the same as the `while` loop, except that the statement block (or single statement) is evaluated at least once even if the condition is not satisfied. The `do ... while` loop is written as:

```
do
{
    statements
}
while(condition)
```

A typical use of the `do ... while` loop is when the condition that determines the end of the loop is determined within the loop as shown in Source 1.17.

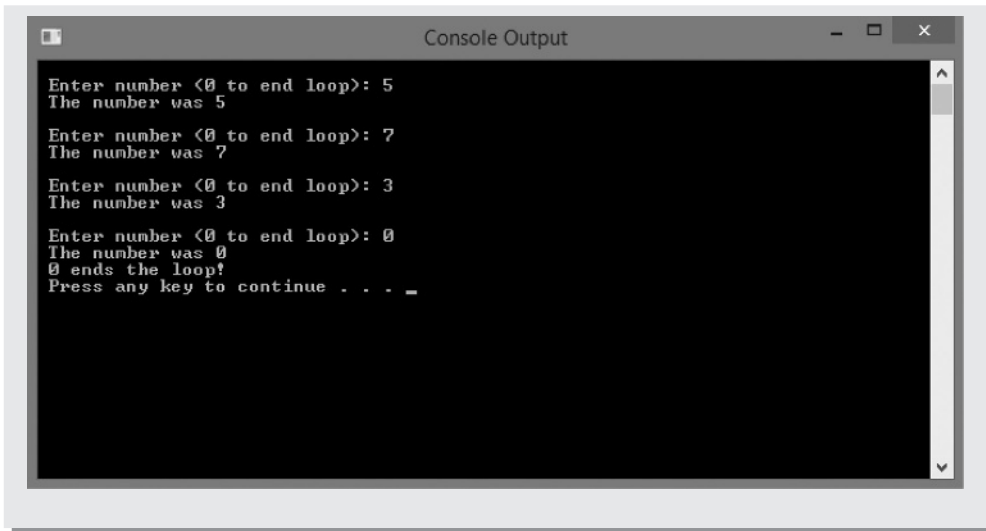
#### SOURCE 1.17: THE `DO ... WHILE` LOOP

```
// ...

int n;

// do...while loop
do
{
    cout << "\n " << "Enter number (0 to end loop): ";
    cin >> n;
    cout << " " << "The number was " << n << "\n ";
}
while(n != 0);
    cout << "0 ends the loop!" << "\n ";

// ...
```



## 1.7 ARRAYS

C++ provides a data structure, the array, which stores a fixed-size sequential collection of elements (data) of the same type. All arrays consist of contiguous memory locations with the lowest address corresponding to the first element and the highest address to the last element. Source 1.18 shows a typical initialisation and implementation of an array structure. Note that in C++ the first element of an array is [0] i.e. indexing starts at 0 not 1.

### SOURCE 1.18: ARRAYS

```
// ...

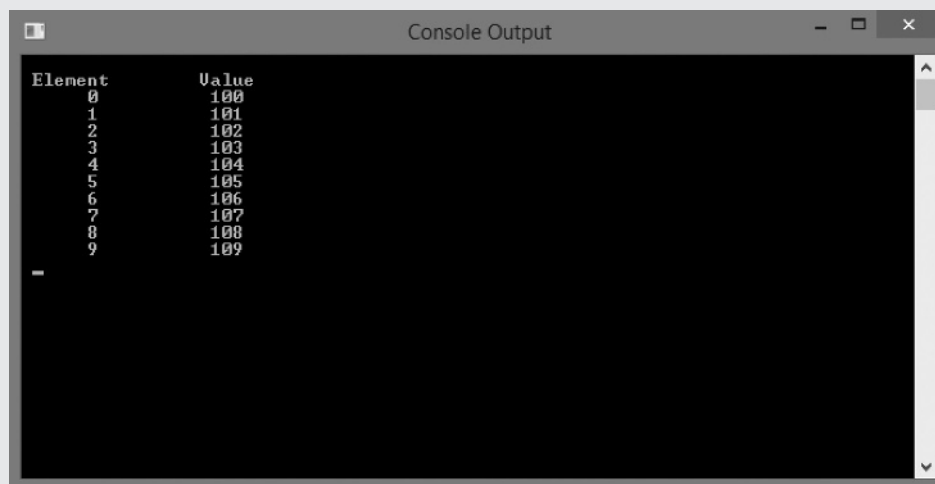
// Declare an array of 10 integers
int n[10];

// Initialise elements of array
for (int i=0; i<10; i++)
{
    n[i] = i + 100; // Set element at location i to i + 100
}

cout << "\n " << "Element" << setw(13) << "Value" << "\n";
```

```
// Output each array element and value
for (int j=0; j<10; j++)
{
    cout << setw(7) << j << setw(13) << n[j] << "\n";
}
cout << " ";

cin.get();
return 0;
}
```



Element	Value
0	100
1	101
2	102
3	103
4	104
5	105
6	106
7	107
8	108
9	109

In Source 1.18, we have made use of setting field width to make the output look more tidy using `setw()` which is declared in the header `<iomanip>`.

## 1.8 VECTORS

Just like arrays, vectors use contiguous storage locations for their elements (data) within so-called sequence containers. However, unlike arrays, their size can change dynamically, with their storage being handled automatically by the container. Internally, vectors use a dynamically allocated array to store their elements. Arrays need to be reallocated in order to grow in size when new elements are inserted, which implies allocating a new array and moving all elements to it. This is a relatively expensive task in terms of processing time, and thus, vectors do not reallocate each time an element is added to the container. Instead, vector containers may allocate some extra

storage to accommodate for possible growth, and thus the container may have an actual capacity greater than the storage strictly needed to contain its elements (i.e. its size). Source 1.19 shows a typical initialisation and implementation of a vector.

### SOURCE 1.19: VECTORS

```
// ...

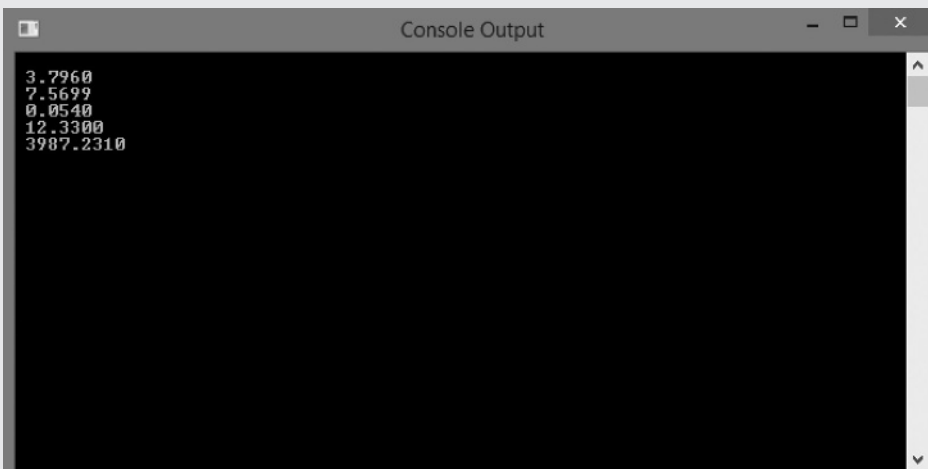
vector<double> v; // Vector of doubles

v.push_back(3.796);
v.push_back(7.56989857);
v.push_back(0.054);
v.push_back(12.33);
v.push_back(3987.231);

cout << "\n ";
cout << fixed << setprecision(4); // Set number precision

for (unsigned int i=0; i<v.size(); i++)
{
    cout << v.at(i) << "\n ";
}

// ...
```



```
3.7960
7.5699
0.0540
12.3300
3987.2310
```



In Source 1.19, we include the header `<vector>` to use the functions to manipulate vectors e.g. `push_back()` and `size()`. Note that when using vectors within `for` loops it is often necessary to explicitly set the increment counter to `unsigned int`. For example, when using the `size()` method of vector the returned value is always going to be positive and so declared `unsigned int` in the official C++ documentation. If we just use an `int` for the increment counter in the `for` loop the two variable types will clash since `unsigned int` is not the same as `int`. The `at()` function returns a reference to the element at position `n` in the vector and automatically checks whether `n` is within the bounds of valid elements in the vector, throwing an `out_of_range` exception if it is not (e.g. if `n` is greater than its size). Note that we have made use of setting number precision using the statement:

```
cout << fixed << setprecision(4)
```

where both `fixed` and `setprecision()` are declared in the header `<iomanip>`.

## 1.9 FUNCTIONS

Often programs can become long and hard to follow making them difficult to understand and debug. In C++ it is possible to break down a program into smaller more manageable units of code known as *modules*, or more formally *functions*. A function is a unit of code that operates on one or more parameters passed into the function which, in general, returns a value based on a set of mathematical operations. The general structure of a function is written as:

```
type function_name(type param1, type param2, ...)
{
    statements;
    return expression;
}
```

The first line of the function is known as the *function* header where the `type`, function name and parameter list are declared. The function header and body together are known as the *implementation* of the function. A function can return only *one* value (e.g. `int`, `double`, etc.) or if there is no value to return the function is declared of type `void`. A function can have as many parameters, each separated by a comma, as necessary or no parameters at all. Indeed, functions can also pass other functions as parameters provided they are declared correctly. Note that the data type returned by a function must *match* that of the header declaration. Source 1.20 below shows a simple program for multiplying two type `double` variables.

**SOURCE 1.20: A SIMPLE FUNCTION**

```
// ...

// Function prototype
double Product(double a, double b);

int main()
{
    SetConsoleTitle(L"Console Output"); // Set title of console window

    // Declare variables
    double x, y;

    cout << "\n " << "Enter x and y: ";
    cin >> x >> y;

    cout << " " << "Product = " << Product(x,y) << "\n ";

    system("PAUSE");
    return 0;
}

// Function definition
double Product(double a, double b)
{
    return(a*b);
}
```



In Source 1.20, the function is first declared using a function *prototype* above the `main()` subroutine. The prototype is a *copy* of the function header terminated with a semi-colon (;). Be aware that when declaring a function (through the *function prototype*) we refer to *parameters*, however, when calling the function, we refer to *arguments*. Arguments are the parameters passed to the function when it is called, and once inside the function, these parameters can be used, and changed, just like any other variable. Within a function prototype, it is possible to omit their names, for example we could have written the prototype for the `Product()` function as:

```
double Product(double, double);
```

In Source 1.20, `x` and `y` are the parameters of the `Product()` function and `a` and `b` are the arguments of the `Product()` function call. When `Product()` is called the values entered for `a` and `b` are *passed* to `x` and `y` and subsequently used in the function body. The function can change the values of `x` and `y` but cannot change the values of `a` and `b`. If it is a requirement that the value of a parameter passed to a function is only allowed to read it and not *change* the value inside the function, the parameter can be set to *constant* in both the prototype and header declaration. For example:

```
double Product(const double x, double y);
```

In this case, the value of `x` cannot be changed in the body of the function; however, `y` still can be changed. When using arguments in a function call it is also possible to *initialise* the values for the parameters in the function prototype. For example:

```
int Init(int x, int y = 10, int = 30);
```

Note that once a parameter has an initial value then all other parameters to the right of it must also be declared with initial values. We have already seen that it is not explicitly necessary to state parameter names in function prototypes, so the third parameter in `Init()` is equally acceptable.

In C++, function prototypes are usually put in files with extension *.h*, known as *header files*, while function definitions go into files with the extension *.cpp*, known as *source files* (see Classes later). *Preprocessor directives* (#) can be used to avoid multiple includes during compilation, for example:

```
#ifndef HEADERNAME_H
#define HEADERNAME_H
... contents of the header file.
#endif
```

On most modern compilers it is possible to replace the above statements with a single `pragma` statement as follows:

```
#pragma once
```

### 1.9.1 Call-by-Value vs. Call-by-Reference

When function arguments are *call-by-value* it implies that *copies* of any variables within the parameter list are made as they are called. So, in Source 1.20, copies of *a* and *b* in the body of `Product()` are made during execution and not *a* and *b* themselves. This works fine but can be *inefficient* when dealing with large numbers of parameters and function calls in a program. In order to get around this problem, we can use *call-by-reference* which passes the *memory address* of the argument variables to the function rather than the variables themselves. The function body can then act directly on the variables and dynamically change them in memory as required. Changing the value of the variable stored in the allocated memory address is the same as changing the value of the variable itself. If we are passing an argument *a* by reference to a parameter variable *x*, then *x* is said to be an *alias* for *a* and both share the same address in memory. That is, when the function acts on *x* in its body, it essentially also acts on *a*, so when it changes *x*, it changes *a* as well. In order to pass an argument by reference, the *reference operator* (`&`) is placed in front of the parameter name in both the function prototype and definition.

A major disadvantage of using references in argument lists is that the function is able to modify the input arguments. What we would ideally need is the address of a variable while at the same time not being able to modify it in any way. In this case, we can define the variable to be a *constant reference*. Source 1.21 shows how call-by-value, call-by-reference and constant reference are used to control access to arguments in a function.

#### SOURCE 1.21: CALL-BY-VALUE, CALL-BY-REFERENCE AND CONSTANT REFERENCE

```
// ...
// Function prototype
void swap(int i, int j);

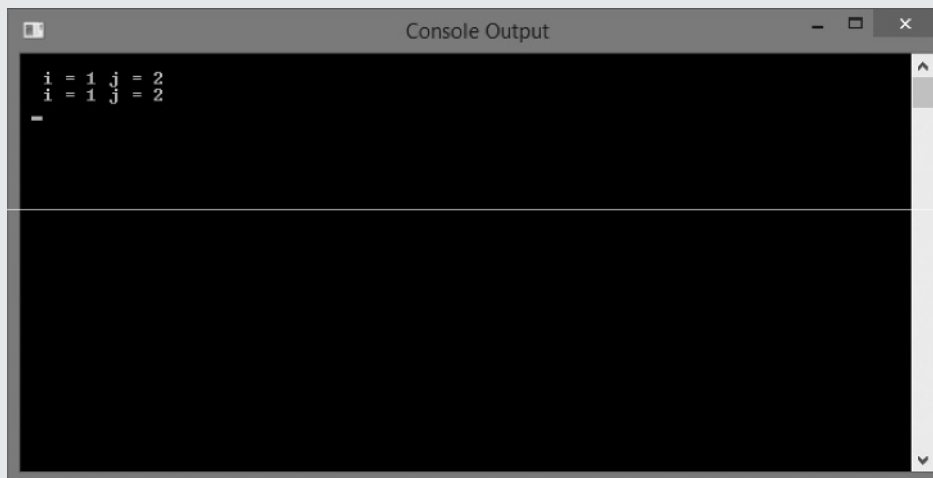
int main()
{
    SetConsoleTitle(L"Console Output");

    //declare variables
    int i = 1, j = 2;

    cout << "\n " << " i = " << i << " j = " << j << "\n ";
    swap(i,j);
    cout << " i = " << i << " j = " << j << "\n ";
```

```
        cin.get();
        return 0;
    }

    // Function definition
    void swap(int i, int j)
    {
        int t;
        t=i;
        i=j;
        j=t;
    }
}
```



```
Console Output
i = 1 j = 2
i = 1 j = 2
```

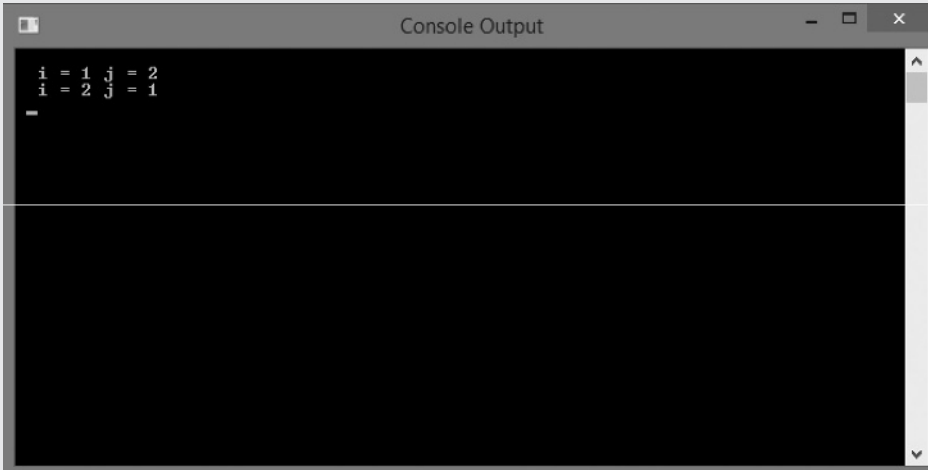
```
// ...
// Function prototype
void swap(int &i, int &j);

int main()
{
    SetConsoleTitle(L"Console Output");

    //declare variables
    int i = 1, j = 2;

    cout << "\n " << " i = " << i << " j = " << j << "\n ";
    swap(i,j);
    cout << " i = " << i << " j = " << j << "\n ";
}
```

```
    cin.get();  
    return 0;  
}  
  
// Function definition  
void swap(int &i, int &j)  
{  
    int t;  
    t=i;  
    i=j;  
    j=t;  
}
```



```
i = 1 j = 2  
i = 2 j = 1
```

```
// ...  
// Function prototype  
void swap(const int &i, const int &j);  
  
int main()  
{  
    SetConsoleTitle(L"Console Output");  
  
    //declare variables  
    int i = 1, j = 2;  
  
    cout << "\n " << " i = " << i << " j = " << j << "\n ";  
    swap(i,j);  
    cout << " i = " << i << " j = " << j << "\n ";
```

```
    cin.get();  
    return 0;  
}  
  
// Function definition  
void swap(const int &i, const int &j)  
{  
    int t;  
    t=i;  
    i=j;  
    j=t;  
}
```

```
error C3892: 'i' : you cannot assign to a variable that is const  
error C3892: 'j' : you cannot assign to a variable that is const
```

In Source 1.21, when we make `i` and `j` `const` and pass-by-reference (`&`), both `i` and `j` are shown as errors in the `swap()` function since we cannot change a variable that is declared `const`.

Also, note that vectors are very large on memory and as such should *always* be passed into functions by *reference*, for example:

```
void fillVector(vector<int>&);
```

If you do not plan to make any changes to the vector it is still necessary to pass it in by reference but in this case add a `const` modifier, for example:

```
void fillVector(const vector<int>&);
```

### 1.9.2 Overloading Functions

In C++, it is possible to create functions that have the same type and name but different number of parameters or data types. This is known as *overloading a function* (or *polymorphism*). The main reason for overloading functions would be where two or more functions perform similar tasks but have different parameters. For example, we could use operator overloading to declare two `swap()` functions; one that accepts integer values and another that accepts doubles as shown in Source 1.22.

**SOURCE 1.22: OVERLOADING THE FUNCTION swap()**

```
// ...
// Function prototype
void swap(int &i, int &j);
void swap(double &k, double &m);

int main()
{
    SetConsoleTitle(L"Console Output");

    //declare variables
    int i = 1, j = 2;
    double k = 2.86, m = 3.84;

    cout << "\n " << " i = " << i << " j = " << j << "\n ";
    swap(i,j);
    cout << " i = " << i << " j = " << j << "\n ";

    cout << "\n " << " k = " << k << " m = " << m << "\n ";
    swap(k,m);
    cout << " k = " << k << " m = " << m << "\n ";

    cin.get();
    return 0;
}

// Function definition (integers)
void swap(int &i, int &j)
{
    int t;
    t=i;
    i=j;
    j=t;
}

// Function definition (doubles)
void swap(double &k, double &m)
{
    double t;
    t=k;
    k=m;
    m=t;
}
```



A screenshot of a 'Console Output' window with a black background and white text. The text shows the output of a C++ program. It displays two lines of integer assignments: 'i = 1 j = 2' followed by 'i = 2 j = 1'. Below these, it shows two lines of floating-point assignments: 'k = 2.86 n = 3.84' followed by 'k = 3.84 n = 2.86'. The window has a standard title bar with a close button, a maximize button, and a scroll bar on the right side.

```
i = 1 j = 2
i = 2 j = 1

k = 2.86 n = 3.84
k = 3.84 n = 2.86
```

In Source 1.22, the compiler decides the definition to use based on the arguments provided when the function is called. To choose the definition to use, the compiler first searches for a definition with parameters that match an invocation exactly. If an exact match is not found, the compiler tries to match by converting types where possible. If a suitable definition cannot be found, a compilation error occurs.

## 1.10 OBJECT ORIENTED PROGRAMMING

As already mentioned, only those parts of the C++ language that we need to successfully implement the tools and techniques in the rest of the book have been covered. However, one very useful development in C++, and something we will make great use of in this book, is the concept of *Object-Oriented Programming* (OOP). OOP allows the building of large and complex programs that can be broken down into smaller self-contained reusable code units known as *classes*. In OOP, data and its manipulation are brought together into a single entity called an *object*. Programs then consist of one or more objects interacting with each other to solve a particular problem. The object is responsible for its data and its data can only be manipulated by a predefined list of acceptable operations. In essence, OOP aims to emulate the way humans interact with the world around them. In this way, pretty much anything can be modelled as an object. Consider a typical day, get out of bed, have a cup of coffee, catch a bus to work, go to a restaurant for lunch, go to your home, eat your dinner with a knife and fork, watch television etc. It is possible to look on life as a series of interactions with things. These things we call objects. We can view all of these objects as consisting of data (*properties*) and operations that can be performed on them (*methods*).

Consider the simple example of a motor car that has hundreds of properties e.g. colour, model type, year, engine capacity, leather interior etc. Let us concentrate on a single property: velocity. Every car has the property of velocity. If it is parked the velocity is zero, if it is on the road it is moving at a certain speed in a particular direction. What are the methods that can modify the car's velocity? We can press the accelerator to increase the speed, we can press the brake to reduce the speed and finally we can turn the steering wheel to alter the direction of the motion. The speedometer is a method we can consult at any time to access the value of the speed component of velocity. However, it is clear that without using any of the other methods we cannot directly manipulate the velocity (unless we have a crash in which case the velocity goes to zero very rapidly).

### 1.10.1 Classes and Abstract Data Types

The basic building block of OOP is the *class*. A class defines the available characteristics and behaviour of a set of similar objects. A class is an *abstract* definition that is made concrete at run-time when objects based upon the class are *instantiated* and take on the behaviour of the class. Data abstraction is the process of creating an object whose implementation details are *hidden* (or *encapsulated*) and the object is used through a well-defined *interface*. Data abstraction leads to an *abstract data type* (ADT), e.g. when you use a floating point number in a program you are not really bothered about exactly how it is represented inside the computer, provided it behaves in the manner you expect. ADTs should be used independent of their implementation meaning that even if the implementation changes the ADT can be used without modification. Most people would be unhappy if they took their car to the garage for a service and afterwards the mechanic said '*She's running lovely now, but you'll have to use the pedals in reverse from now on*'. If this were the case, we could not think of the car as an ADT. However, the reality is that we can take a car to a garage for a major overhaul (which represents a change of implementation) and still drive it in exactly the same way afterwards.

Consider the concept of a 'Vehicle' class. The class would include *methods* such as `Steer()`, `Accelerate()` and `Brake()`. The class would also include *properties* such as `Colour`, `NumberOfDoors`, `TopSpeed`. The class is an abstract design that becomes real when objects such as `Car`, `RacingCar` and `Tank` are created, each with its own version of the class's methods and properties. Furthermore, *message passing* (or *interfacing*) describes the communication between objects using their *public interfaces*. There are three main ways to pass messages: *properties*, *methods* and *events*:

- *Properties* can be defined in a class to allow objects of that type to advertise and allow changing of state information, such as the 'TopSpeed' property.

- *Methods* can be provided so that other objects can request a process to be undertaken by an object, such as the `Steer()` method.
- *Events* can be defined that an object can raise in response to an internal action.

Other objects can subscribe to these so that they can react to an event occurring. An example for vehicles could be an 'ImpactDetected' event subscribed to by one or more 'AirBag' objects.

### 1.10.2 Encapsulation and Interfaces

*Encapsulation* refers to the process of *hiding* the implementation details of an object. A washing machine is a good example of an encapsulated object. We know that inside a washing machine are a whole series of complex electronics, however, we do not need to understand them to wash our clothes. In terms of our concept of an object, encapsulation hides the properties, some methods, and all method implementation details of an object from the outside. For example, the velocity of a car cannot be magically changed, we have to press the accelerator or brake – methods that we do not need to know the details of. In this respect the velocity of the car is hidden from outside interference, but can be changed by visible methods.

An *interface* is a simple *control panel* that enables us to use an object. The great benefit of the interface is that we only need to understand the simple interface to use the washing machine or drive the car. This is much easier than understanding the internal implementation details.

Another benefit is that the implementation details can be changed or fixed and we can still use the washing machine or car. For example, suppose your car breaks down and you take it to the garage and they replace the engine with a bigger and better one. The car operates in exactly the same way since the interface has remained unchanged. So, it is extremely important to design a good interface that will not change. The inner workings can be modified and cause no external operational effect. That is, once the user understands the interface, the implementation details can be modified, fixed or enhanced, and once the interface is unchanged the user can seamlessly continue to use the object.

The interface establishes *what* requests you can make for a particular object. However, there must be code somewhere to satisfy that request. This, along with the *hidden data*, comprises the *implementation*. The goal of the *class creator* is to build a class that exposes only what's necessary to the user and keeps everything else hidden. Why? Because if it's hidden, the programmer cannot use it, which means that the class creator can change the hidden portion at will without worrying about the impact on anyone else. The hidden portion usually represents the tender insides of an object that could easily be corrupted by a careless or uninformed programmer, so hiding the implementation reduces program bugs. Once a class has been created and tested, it should (ideally) represent a useful *unit of code*. Code reuse is one of the

greatest advantages that OOP languages provide. The simplest way to reuse a class is to just use an object of that class directly.

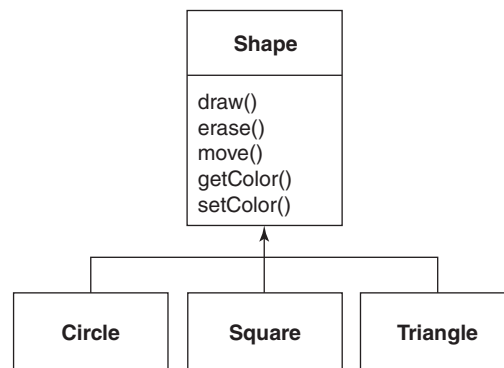
### 1.10.3 Inheritance and Overriding Functions

It seems a pity, however, to go to all the trouble of creating a class and then be forced to create a brand new one that might have similar functionality. It would be nicer if we could take the existing class, *clone it* and then make additions and modifications to the clone. This is effectively what you get with *inheritance*, with the exception that if the original class (the *base class*) is changed, the modified ‘clone’ (the *derived class*) also reflects those changes. For example, consider the base type Shape in which each shape has a size, a colour, a position and so on. Each shape can be drawn, erased, moved, coloured, etc. From this, specific types of shapes are derived (*inherited*) e.g. circle, square, triangle and so on, each of which may have additional characteristics and behaviours. Certain shapes can be flipped, for example. Some behaviours may be different, such as when you want to calculate the area of a shape. The type *hierarchy* embodies both the similarities and differences between the shapes as shown in Figure 1.1.

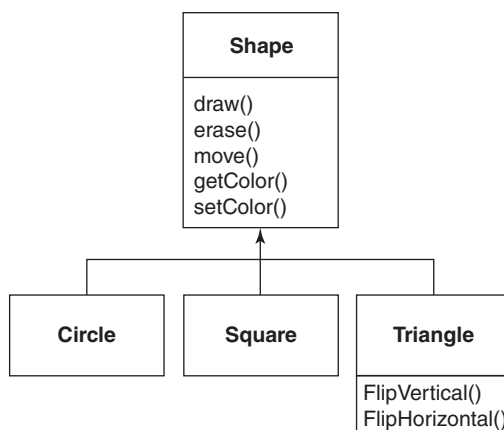
When you inherit from an existing type, you create a *new type*. This new type contains not only all the members of the existing type but more importantly it *duplicates the interface* of the base class.

So, all the messages you can send to objects of the base class you can also send to objects of the derived class. Since we know the type of a class by the messages we can send to it, this means that the derived class *is the same type as the base class* e.g. a circle is a shape. This type equivalence through inheritance is one of the fundamental gateways to understanding the meaning of OOP.

In reality, you have two ways to differentiate your new derived class from the original base class. You can simply *add brand new functions* to the derived class. These new functions are not part of the base class interface. This means that the base class simply does not do as much as you wanted it to, so you added more functions.



**FIGURE 1.1** Schematic of a typical class hierarchy



**FIGURE 1.2** Adding new functions to the derived class

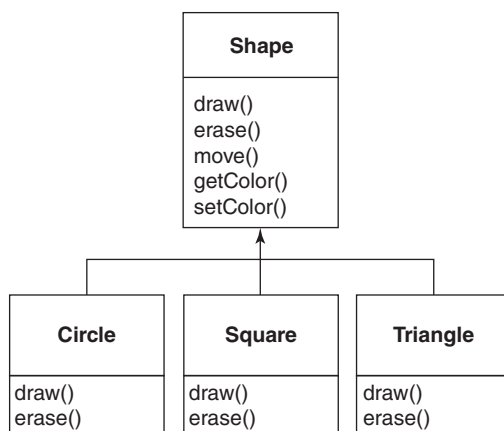
This simple and primitive use for inheritance is, at times, the perfect solution to your problem (see Figure 1.2).

The second and more important way to differentiate your new class is to *change* the behaviour of an existing base-class function. This is referred to as *overriding* that function.

To override a function, you simply create a new definition for the function in the derived class. You're saying, '*I'm using the same interface function here, but I want it to do something different for my new type*', as shown in Figure 1.3.

#### 1.10.4 Polymorphism

We briefly touched upon *polymorphism* when discussing functions above. It is generally perceived as the most feared part of object orientation. Polymorphism, which

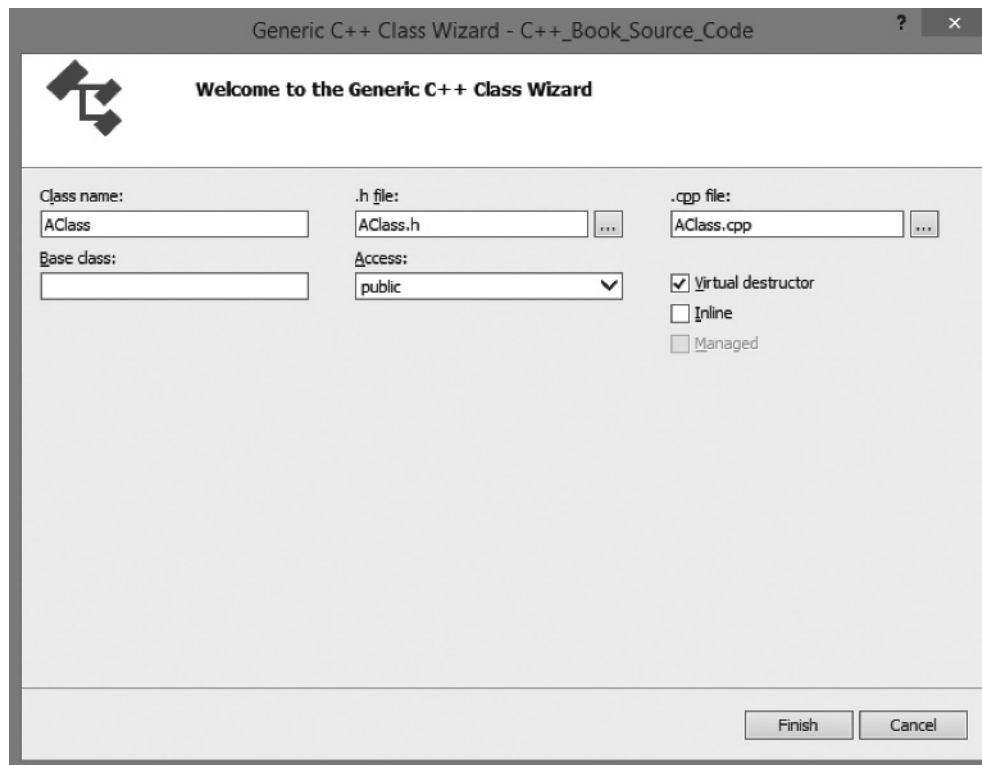


**FIGURE 1.3** Overriding base-class functions

literally means *poly* (*many*) and *morph* (*forms*), will again only be mentioned here for completeness. Objects interact by calling each other's methods. How does some object *A* know the supported methods of another object *B*, so that it can call a valid method of *B*? In general, there is no magic way for *A* to determine the supported methods of *B*. *A* must know in advance the supported methods of *B*, which means *A* must know the class of *B*. However, polymorphism means that *A* does not need to know the class of *B* in order to call a method of *B*. In other words, an instance of a class can call a method of another instance, without knowing its exact class. The calling instance need only know that the other instance supports a method, it does not need to know the exact class of the other instance. It is the instance receiving the method call that determines what happens, not the calling instance.

### 1.10.5 An Example of a Class

Source 1.23 shows an example of a simple class named `AClass` created using the Generic C++ Class Wizard in Visual Studio Express 2012 Windows Desktop as shown in Figure 1.4. Source 1.24 shows an implementation of the simple `AClass`



**FIGURE 1.4** The Generic C++ Class Wizard dialogue box

class. Note that for all classes, the ending brace (}) is followed by a semi-colon (;) to indicate it is a class.

**SOURCE 1.23: THE .h AND .cpp FILES AUTOMATICALLY GENERATED BY THE VISUAL STUDIO 2013 CLASS WIZARD**

```
// AClass.h
#pragma once

class AClass
{
public:
    AClass();
    virtual ~AClass();
private:
};

// AClass.cpp
#include "AClass.h"

AClass::AClass()
{
}

AClass::~~AClass()
{
}
```

**SOURCE 1.24: A SIMPLE IMPLEMENTATION OF THE AClass CLASS**

```
// AClass.h
#pragma once

class AClass
{
public:
    AClass();
    virtual ~AClass();
```

```
// Function declaration
void sayHello();
private:
};

// AClass.cpp
#include <iostream>
using std::cout;

#include "AClass.h"

// Default constructor
AClass::AClass()
{
}

// Default destructor
AClass::~~AClass()
{
}

// Function definition
AClass::sayHello()
{
    cout << " Hello\n";
}

// main.cpp
#include "AClass.h"

#include <windows.h>
#include <iostream>
using std::cout;
using std::cin;

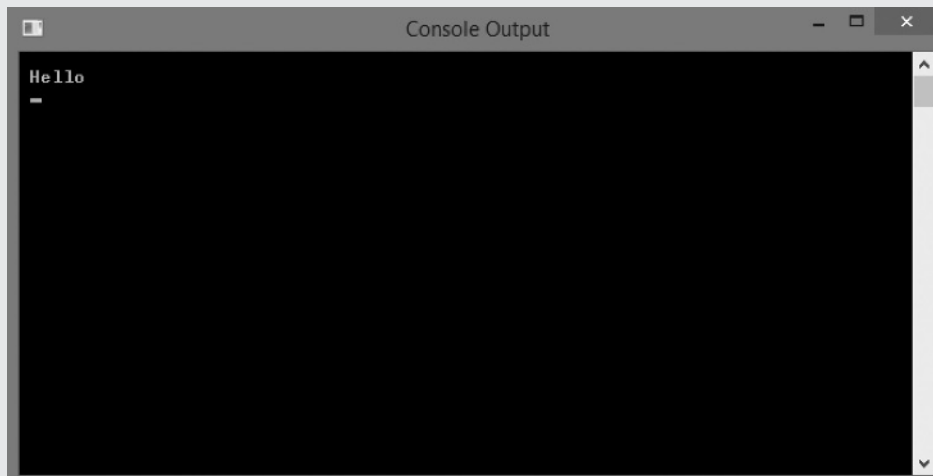
int main()
{
    SetConsoleTitle(L"Console Output");

    // Create an instance of the AClass class (instantiation)
    AClass obj;
    // Call the sayHello() function using the dot operator
    obj.sayHello();

    cin.get();
}
```



```
    return 0;
}
```



### 1.10.6 Getter and Setter Methods

Both member data and functions can be declared either `private` or `public` (protected will not be discussed in this book). Those declared `public` can be accessed from any part of a program that includes the class with use of the *dot notation* (`.`). Those declared as `private` can only be accessed by member functions of the class. The default declaration for member data and functions is `private` and to access these `private` variables from outside of the class we generally use `public setter` (or *mutator*) and `getter` (or *accessor*) methods as shown in Source 1.25. In general, getters do not modify any of the member variables in the class and so it is good practice to make them constant by adding the `const` modifier.

#### SOURCE 1.25: USING SETTER AND GETTER METHODS

```
// AClass.h
#pragma once

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

```
class AClass
{
public:
    AClass();
    void setName(string x); // Setter prototype
    string getName() const; // Getter prototype with const
    modifier

    virtual ~AClass();
private:
    string m_name; // Member variable
};

// AClass.cpp
#include <iostream>
using std::cout;

#include "AClass.h"

AClass::AClass()
{
}

AClass::~~AClass()
{
}

// Setter function
void AClass::setName(string name)
{
    m_name = name;
}

// Getter function with const modifier
string AClass::getName() const
{
    return m_name;
}

// main.cpp
#include "AClass.h"

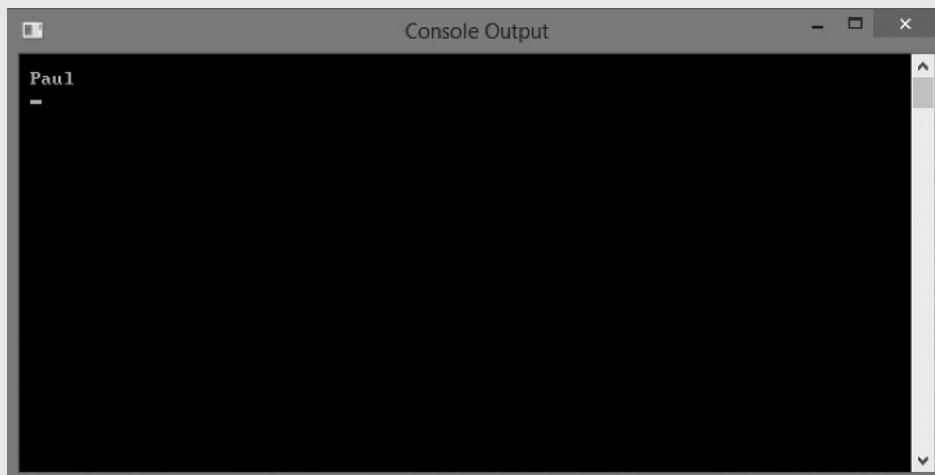
#include <windows.h>
#include <iostream>
```

```
using std::cout;
using std::cin;

int main()
{
    SetConsoleTitle(L"Console Output");

    // Create an instance of the AClass class (instantiation)
    AClass obj;
    // Call the sayHello() function using the dot operator
    obj.setName("Paul");
    cout << '\n' << obj.getName() << '\n'; // Call getter
    using dot operator

    cin.get();
    return 0;
}
```



In Source 1.25, each function prototype in the *.h* file is attached to the relevant class definition in the *.cpp* file using the *scope resolution operator* (`::`). Subsequently, each member function of the class is accessed in the main body using the *dot operator* notation. When we instantiate an object (`obj`) of the class in the main program, the compiler creates a *copy* of all member data and functions of the class for that particular object. If more objects of the same class are declared additional copies of the member data and functions are created for each object. The compiler actually handles all of the same data and functions for each of the objects so that nothing gets mixed

up. Although the code will work correctly it is not necessarily efficient. A more practical solution is to use the address of the object and the *call-by-reference* technique as described above for functions.

### 1.10.7 Constructors and Destructors

The *default constructor* is a member function that has the same name as the class and invoked *automatically* once an instance of an object is *created*. Similarly, the *default destructor* has the same name as the class prefixed with a tilde (~) and invoked *automatically* once an object is *destroyed*. In practice, there is only ever *one* default destructor although the constructor can have several *overloaded* methods. It is generally considered good practice to include the keyword `virtual` before the destructor so as to alleviate any potential *memory leaks*. An example of the default constructor and destructor is shown in Source 1.26.

#### SOURCE 1.26: THE DEFAULT CONSTRUCTOR AND DESTRUCTOR

```
// AClass.h
#pragma once

#include <string>
using std::string;

class AClass
{
public:
    AClass(); // Default constructor
    void setName(string x); // Setter prototype
    string getName() const; // Getter prototype with const modifier

    virtual ~AClass(); // Default destructor
private:
    string m_name; // Member variable
};

// AClass.cpp
#include <iostream>
using std::cout;

#include "AClass.h"
```

```
// Default constructor
AClass::AClass()
{
    cout << " Default constructor called ...\n";
}

// Default destructor
AClass::~~AClass()
{
    cout << " Default destructor called ...\n";
}

// Setter function
void AClass::setName(string name)
{
    m_name = name;
}

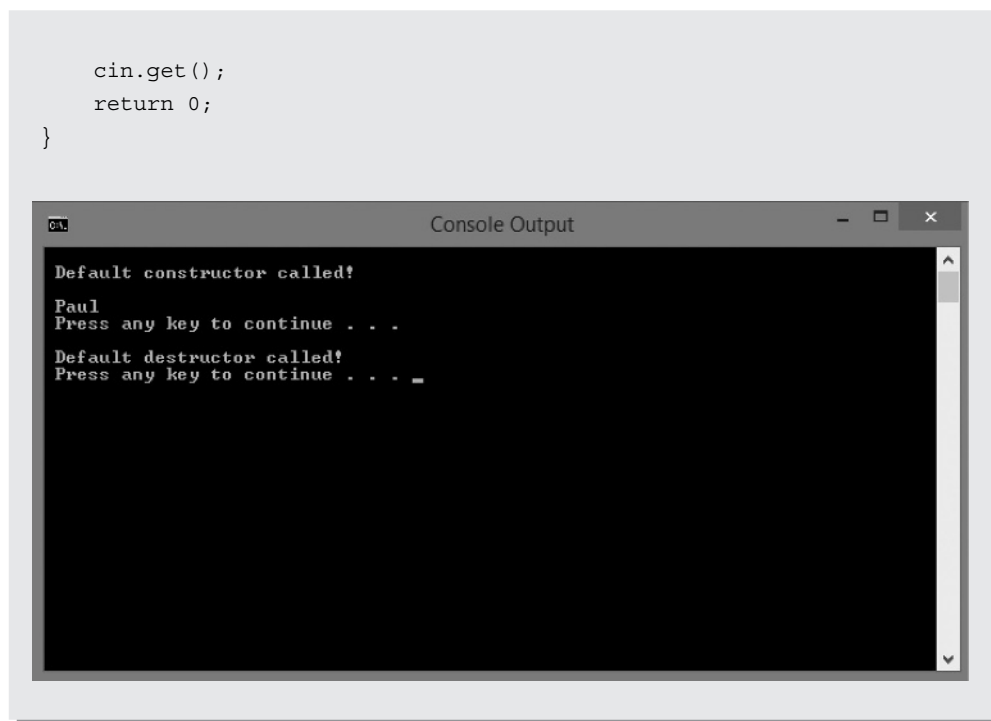
// Getter function with const modifier
string AClass::getName() const
{
    return m_name;
}

// main.cpp
#include "AClass.h"

#include <windows.h>
#include <iostream>
using std::cout;
using std::cin;

int main()
{
    SetConsoleTitle(L"Console Output");

    // Create an instance of the AClass class (instantiation)
    AClass obj;
    // Call the sayHello() function using the dot operator
    obj.setName("Paul");
    cout << '\n' << obj.getName() << '\n'; // Call getter using
    dot operator
```



Both default constructors and destructors are provided automatically in C++ so do not have to be explicitly implemented in the class. However, experienced programmers usually include them for completeness and especially if any member variables need to be *initialised* with default values. Constructors are generally used to initialise data whereas destructors are used to tidy up any outstanding code issues once an object has been destroyed. There are several methods of initialising data with constructors, for example initialising the data inside the constructor:

```
AClass::AClass()  
{  
    m_x = 5;  
}
```

It is also possible to write *parameterised* constructors, for example:

```
AClass::AClass(int x, int y, int z)  
{  
    m_x = x;  
    m_y = y;  
    m_z = z;  
}
```

Obviously in all cases the prototype function for the constructor must match the function definition. The above constructor can also be written using the following notation:

```
AClass::AClass(int x, int y, int z)
{
    m_x(x);
    m_y(y);
    m_z(z);
}
```

It is also possible to initialise member variables using the *colon operator* (:), that is:

```
AClass::AClass : x(5)
{
}
```

Any additional member variables are initialised by separating with commas, for example:

```
AClass::AClass : x(5), y(4), z(3)
{
}
```

### 1.10.8 A More Detailed Class Example

Source 1.27 shows a more detailed example of a class dealing with bank accounts and balances. Many of the issues already discussed are covered in the example as well as some new concepts.

#### **SOURCE 1.27: A DETAILED IMPLEMENTATION OF A BANK ACCOUNT AND BALANCES CLASS**

```
// Bank.h
#pragma once

#include <string>
using std::string;

class Bank
```

```
{
public:
    // Default constructor
    Bank();

    // Overloaded constructor
    Bank(string, int, double);

    // Default destructor
    virtual ~Bank();

    // Accessor functions
    string getName() const;
    int getAccNum() const;
    double getBalance() const;

    // Mutator functions
    void setName(string);
    void setAccNum(int);
    void setBalance(double);

    // Member functions
    void withdraw(double);
    void deposit(double);

    // We are printing static variables so must prefix the
    function with the keyword static.
    // NOTE: only add static in the declaration (.h) NOT
    definition (.cpp)
    static void printBankInfo();

private:
    // Member variables
    string m_name;
    int m_accNum;
    double m_balance;

    // Static member variables
    static int m_totalAccounts;
    static double m_bankBalance;
};

// Bank.cpp
```



```
#include "Bank.h"

#include <iostream>
using std::cout;

// Initialise static member variables
int Bank::m_totalAccounts = 0;
double Bank::m_bankBalance = 10000;

// Every time we instantiate a new user in the bank either thorough the default
// or overloaded constructor we must increment the number of accounts.
Bank::Bank()
{
    m_accNum = 0;
    m_balance = 0.0;
    m_totalAccounts++;
}

// We do not need to modify the default constructor wrt the bank
// balance because it is simply
// initialised to zero and there are no parameters.
// However, for the overloaded constructor there
// is a newBalance parameter for user deposits. So every time a
// user deposits some money the
// new balance must be added to m_bankBalance.
Bank::Bank(string name, int accNum, double balance)
{
    m_name = name;
    m_accNum = accNum;
    m_balance = balance;
    m_totalAccounts++;
    m_bankBalance += balance;
}

// Once an object is destroyed, m_totalAccounts and m_bankBalance
// change i.e, we must decrement
// totalAccounts and bankBalance of the user being destroyed.
Bank::~Bank()
{
    m_totalAccounts--;
    m_bankBalance -= m_balance;
}
```

```
// Accessor functions
string Bank::getName() const
{
    return m_name;
}

int Bank::getAccNum() const
{
    return m_accNum;
}

double Bank::getBalance() const
{
    return m_balance;
}

// Mutator functions
void Bank::setName(string name)
{
    m_name = name;
}

void Bank::setAccNum(int accNum)
{
    m_accNum = accNum;
}

// The setBalance mutator must also be updated. So before doing
// anything we
// must take m_bankBalance and subtract the balance they may
// already have
// (it could be zero but we don not know). Then add on the new
// balance to the
// m_bankBalance.
void Bank::setBalance(double balance)
{
    m_bankBalance -= balance;
    m_balance = balance;
    m_bankBalance += balance;
}

// Member functions
void Bank::withdraw(double withdraw)
```

```
{
    m_balance -= withdraw;
    m_bankBalance -= withdraw;
}

void Bank::deposit(double deposit)
{
    m_balance += deposit;
    m_bankBalance += deposit;
}

void Bank::printBankInfo()
{
    cout << "\n " << "Number of Accounts: " << m_totalAccounts << "\n ";
    cout << "Total Balance: " << m_bankBalance << "\n ";
}

// main.cpp
#include <windows.h>
#include <iostream>
using std::cout;
using std::cin;

#include "Bank.h"

int main()
{
    SetConsoleTitle(L"Console Output");

    cout << '\n' << "Adam created an account and deposited 500";
    Bank Adam("Adam", 0001, 500); // Calling overloaded constructor

    Bank::printBankInfo(); // Calling printBankInfo() inside a class
                             without instantiating an object

    Bank Sarah; // Calling default constructor

    cout << '\n' << "Sarah created an account and deposited 1000";

    Sarah.setName("Sarah"); // Calling mutator function setName()
    Sarah.setAccNum(0002);   // Calling mutator function setAccNum()
    Sarah.setBalance(1000);  // Calling mutator function setBalance()

    Bank::printBankInfo();
```

```
cout << '\n' << "Eric created an account and deposited 1500";
Bank Eric("Eric", 0003, 1500); // Calling overloaded constructor

Bank::printBankInfo(); // Calling printBankInfo() inside a class
without instantiating an object

cout << "\n" << "\n" << "Eric set his balance to 1200";
Eric.setBalance(1200);

Bank::printBankInfo(); // Calling printBankInfo() inside a class
without instantiating an object

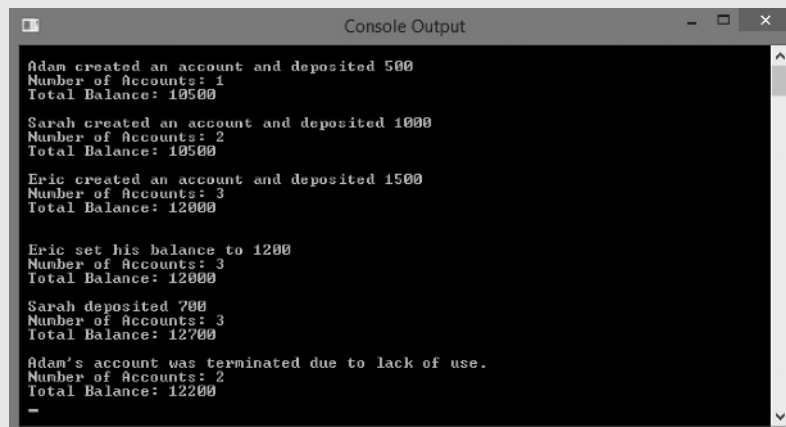
cout << "\n" << "\n" << "Sarah deposited 700";
Sarah.deposit(700);

Bank::printBankInfo(); // Calling printBankInfo() inside a class
without instantiating an object

cout << "\n" << "Adam's account was terminated due to lack of
use";
Adam.~Bank(); // Calling destructor

Bank::printBankInfo(); // Calling printBankInfo() inside a class
without instantiating an object

cin.get();
return 0;
}
```



```
Adam created an account and deposited 500
Number of Accounts: 1
Total Balance: 10500

Sarah created an account and deposited 1000
Number of Accounts: 2
Total Balance: 10500

Eric created an account and deposited 1500
Number of Accounts: 3
Total Balance: 12000

Eric set his balance to 1200
Number of Accounts: 3
Total Balance: 12000

Sarah deposited 700
Number of Accounts: 3
Total Balance: 12700

Adam's account was terminated due to lack of use.
Number of Accounts: 2
Total Balance: 12200
```

In Source 1.27, we have *overloaded* the `Bank()` constructor, that is:

```
Bank();  
Bank(string, int, double);
```

This gives us several methods of instantiating an object of the class within the main body of the program e.g. both Adam and Sarah objects are created using different constructors. We have also added some tidying up code to the default destructor for dealing with closing or terminating accounts.

```
Bank::~Bank()  
{  
    m_totalAccounts--;  
    m_bankBalance -= m_balance;  
}
```

Another important concept we have included is that of *static member variables* and *functions*. Static member variables e.g. `m_totalAccounts` and `m_bankBalance` have the same value in any instance of a class and do not even require an instance of the class to exist. However, we must initialise static member variables using a specific scope operator (`::`) syntax, that is:

```
int Bank::m_totalAccounts = 0;  
double Bank::m_bankBalance = 10000;
```

It is also possible to have *static member functions* of a class, for example:

```
static void printBankInfo();
```

Static member functions are functions that do not require an instance of the class, and are called the same way as you access static member variables i.e. with the class name rather than a variable name, for example:

```
Bank::printBankInfo();
```

Be clear that static member functions can only operate on static members, as they do not belong to specific instances of a class.

### 1.10.9 Implementing Inheritance

Recall our initial discussion of inheritance above. Suppose you have a Class A that has several functions and variables you want to access in another Class B. In this case,

it is possible to *inherit* the functions and variables from Class A and use them as you would in Class B. Whenever you inherit from a class, the main class you inherit from is called the *base* class and the class that does the inheriting is called the *derived* class. Source 1.28 shows an example of implementing inheritance.

### SOURCE 1.28: INHERITANCE

```
// Mother.h
#pragma once

class Mother
{
public:
    Mother();
    void sayName();

    virtual ~Mother();
};

// Mother.cpp
#include "Mother.h"
#include <iostream>
using std::cout;

Mother::Mother()
{
}

void Mother::sayName()
{
    cout << "I am part of the Partridge family" << "\n ";
}

Mother::~~Mother()
{
}

// Daughter.h
#pragma once

class Daughter: public Mother
{
public:
```

```
    Daughter();

    virtual ~Daughter();
};

// Daughter.cpp
#include "Mother.h"
#include "Daughter.h"

Daughter::Daughter()
{
}

Daughter::~~Daughter()
{
}

// main.cpp
#include <windows.h>
#include <iostream>
using std::cout;
using std::cin;

#include "Mother.h"
#include "Daughter.h"

int main()
{
    SetConsoleTitle(L"Console Output");

    Mother mum;
    cout << "\n " << "Mother" << "\n ";
    mum.sayName();

    // Daughter object inherits sayName() from the Mother class
    Daughter helen;
    cout << "\n " << "Daughter" << "\n ";
    helen.sayName();

    cin.get();
    return 0;
}
```



#### 1.10.10 Operator Overloading

*Operator overloading* allows the manipulation of standard C++ operators to change the way they are implemented. For example, suppose you had two class objects that you wanted to add together, operator overloading allows you to treat the operator + differently for these objects. Table 1.7 shows a list of operators that can be overloaded in C++.

**TABLE 1.7** Operators that can be overloaded in C++

+	-	*	/	=	<	>	+=	-=	*=	/=	<<	>>
<<=	>>=	==	!=	<=	>=							
++	--	%	&	^	!		~	&=	^=	=	&&	
%=	[]	()	,	->*	->							
new	delete	new[]	delete[]									

Source 1.29 shows an example of overloading the (+) operator.

#### SOURCE 1.29: OVERLOADING THE (+) OPERATOR

```
// Vec2D.h
#pragma once

#include <iostream>
using std::ostream;
```



```
class Vec2D
{
public:
    int m_x, m_y;

    Vec2D();
    Vec2D(int, int);

    // Overloaded (+) operator declared constant and by reference
    Vec2D operator+(const Vec2D&) const;

    virtual ~Vec2D();
};

// Vec2D.cpp
#include "Vec2D.h"

#include <iostream>
using std::cout;

Vec2D::Vec2D() : m_x(0), m_y(0)
{
}

Vec2D::Vec2D(int x, int y) : m_x(x), m_y(y)
{
}

Vec2D Vec2D::operator+(const Vec2D& v) const
{
    // Uses an anonymous Vec2D object
    return Vec2D(m_x + v.m_x, m_y + v.m_y);
}

Vec2D::~~Vec2D()
{
}

// main.cpp
#include <windows.h>
#include <iostream>
using std::cout;
using std::cin;
```

```
#include "Vec2D.h"

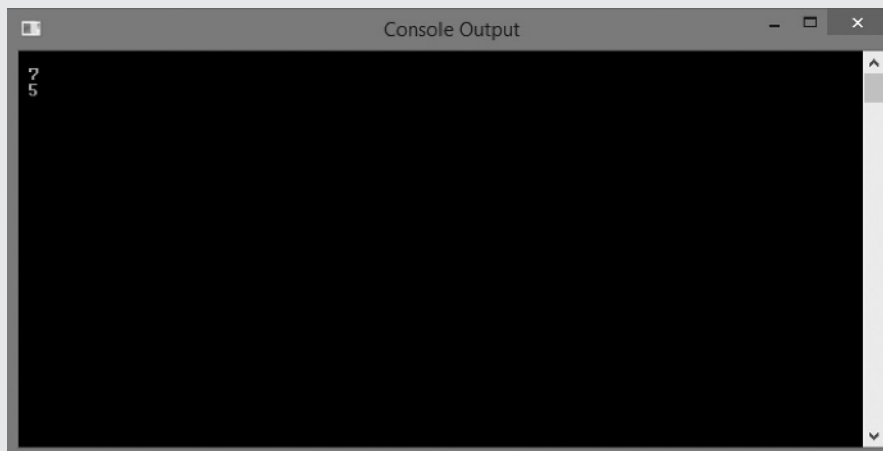
int main()
{
    SetConsoleTitle(L"Console Output");

    Vec2D a(3,3); // Create Vec2D object a
    Vec2D b(4,2); // Create Vec2D object b

    // Use (+) overloaded operator
    Vec2D c = a + b;

    cout << '\n' << c.m_x;
        cout << '\n' << c.m_y; << '\n';

    cin.get();
    return 0;
}
```



Note that there are *three* objects involved in the `operator+` member function: the *current* object that plays the role of the *left operand*, the argument `v` that plays the role of the *right operand* and a *new* object with a return type the same as that of the member function. Furthermore, neither the current object nor the `v` object are modified by the operation so we can add the `const` keyword. What if we wanted to print out `c` directly? For example:

```
cout << "\n " << c << "\n ";
```

This would lead to an error of the type ‘no operator “<<” matches these operands’. Source 1.30 shows how we can overload the *insertion* (<<) operator in Source 1.29 so that we can overcome this error and print out object `c` directly without needing to call its individual member variables i.e. `c.m_x` and `c.m_y`.

### SOURCE 1.30: OVERLOADING THE (<<) OPERATOR

```
// Vec2D.h
#pragma once

#include <iostream>
using std::ostream;

class Vec2D
{
public:
    int m_x, m_y;

    Vec2D();
    Vec2D(int, int);

    // Overloaded (+) operator declared constant and by reference
    Vec2D operator+(const Vec2D&) const;

    // Overload (<<) operator
    // ostream is not a member of the Vec2D class but wants to
    // consume its
    // member variables so we make it a friend of the Vec2D class.
    friend ostream& operator<<(ostream& stream, const Vec2D&);

    virtual ~Vec2D();
};

// Vec2D.cpp
#include "Vec2D.h"

#include <iostream>
using std::cout;

Vec2D::Vec2D() : m_x(0), m_y(0)
{
}

Vec2D::Vec2D(int x, int y) : m_x(x), m_y(y)
```

```
{
}

Vec2D Vec2D::operator+(const Vec2D& v) const
{
    // Uses an anonymous Vec2D object
    return Vec2D(m_x + v.m_x, m_y + v.m_y);
}

ostream& operator<<(ostream& stream, const Vec2D& v)
{
    cout << "(" << v.m_x << ", " << v.m_y << ")";
    return stream;
}

Vec2D::~Vec2D()
{
}

// main.cpp
#include <windows.h>
#include <iostream>
using std::cout;
using std::cin;

#include "Vec2D.h"

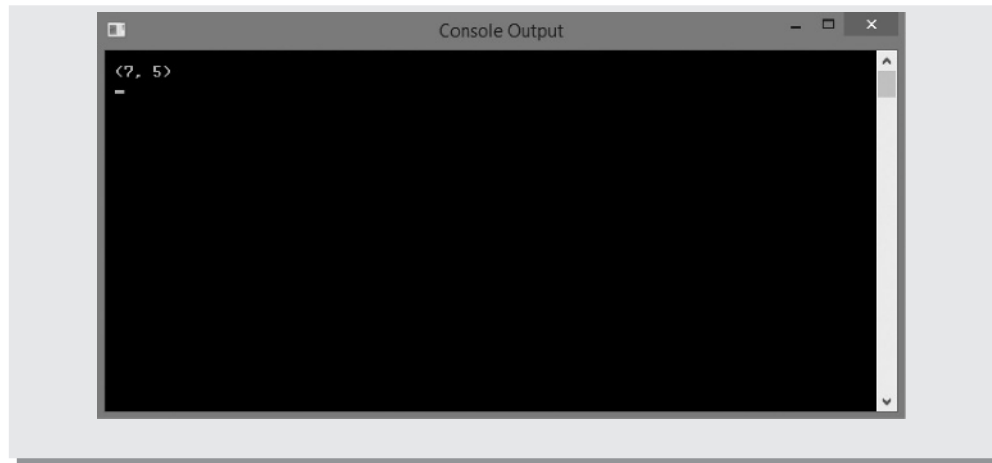
int main()
{
    SetConsoleTitle(L"Console Output");

    Vec2D a(3,3); // Create Vec2D object a
    Vec2D b(4,2); // Create Vec2D object b

    // Use (+) overloaded operator
    Vec2D c = a + b;

    cout << "\n " << c;
    cout << "\n " << c << "\n ";

    cin.get();
    return 0;
}
```



With the statement:

```
cout << "\n " << c << "\n ";
```

the first `<<` invokes the *left shift* operator and shifts `c` into `cout`, then it reads the next `<<` and invokes the left shift operator again and shifts `"\n "` into `cout` i.e. it reads *left -> right*. In Source 1.30, since we are constantly modifying the `ostream` object (i.e. using multiple insertion operations) we must pass it by *reference* so that it takes into account this *operator chaining* process. Note also that `ostream` is not a member (i.e. it is a *non-member*) of the `Vec2D` class but wants to be able to access the *private* members of the class. In this case, we make `ostream` a *friend* of the `Vec2D` class. By default, a non-member function cannot access the private member data of a class unless the class *explicitly* states that that function is allowed to do so, i.e. a class must *give away* its friendship, a friend cannot be simply *assumed* by a non-member function.

Chapter 1 has provided an overview of the key C++ programming ideas and concepts required to develop and build fairly robust quantitative analysis models. All of the programs introduced throughout the book will rely on methods and principles discussed in this chapter. By no means is this a complete resource for developing commercial level software systems but does give an adequate level of experience to begin prototyping such systems. We have also introduced the very powerful concept of OOP hopefully without burdening the reader with too much information and syntax. The idea, as with the other content, is to cover the necessary requirements needed to understand, develop and implement object-oriented programs in order to efficiently solve many of the problems encountered throughout the book. However, there may be several instances when we will be introducing some new C++ not covered explicitly here to solve a particular challenging problem. These will likely rely on more advanced methods in C++ and it makes more sense to discuss these implementations on an individual basis as they are encountered.

