1 Getting Started

In this introductory chapter, you will learn a little bit about the features of C++ in terms of some of the common "buzzwords" you may have heard about the language, and in terms of its strengths and weaknesses. You will also learn how to edit, compile and run your first C++ program. This chapter also includes information on variables and simple ways of getting data into and out of your programs.

The chapter concludes with tips on how you might, as a novice C++ programmer, go about debugging your programs. We have included tips with every chapter in this book. They are presented at an increasing level of sophistication—this should match your gaining knowledge as you read through the book and attempt some of the exercises.

1.1 A Brief Introduction to C++

A very large number of programming languages for writing computer software exist. If one of these programming languages was the most suitable for all purposes, then it would be expected that everyone would use this language, and all other languages would eventually become obsolete. This, however, is certainly not the case. It seems appropriate to begin this book by describing the key features of C++, allowing us to explain why C++ is a suitable programming language for scientific computing applications and why it isn't the only suitable choice of language.

J. Pitt-Francis and J. Whiteley, *Guide to Scientific Computing in C++*, Undergraduate Topics in Computer Science,
https://doi.org/10.1007/978-3-319-73132-2_1

1.1.1 C++ is "Object-Oriented"

You may have heard that C++ is an "object-oriented" language and have wondered what that means. What marks a language which is object-oriented out from one that is not? Fundamentally, it is because the basic unit of the language is an *object* or *class*—an entity which brings together related functionality and data. We will probe the ideas behind objects and classes more deeply in Chap. 6.

Many books on C++ start by defining *object-orientation* more explicitly. If this book were aimed at a computer science or software engineering audience, then we would find it necessary to define some specific concepts related to object-orientation. We would need to convince you of the importance of the following concepts.

- *Modularity*. All the data of a particular object, and the operations that we perform on this object, are held in one or two files, and can be worked on independently.
- *Abstraction*. The essential features and functionality of a class are put in one place and the details of how they work are unimportant to the user of the class. For example, if you are using a linear system library to solve matrix equations you should not need to know the precise details of how matrices are laid out in memory or the exact order that a numerical solver performs its operations. You should only need to know how to use the functionality of the library.
- *Encapsulation*. The implementation of an object is kept hidden from the user of the class. This is not only about clarity (*abstracting* away the detail). It is also about preventing the user from accidentally amending internal workings of, for example, a linear solver, stopping it from working effectively.
- *Extensibility*. Functionality can be reused with selected parts extended. For example, much of the core of a linear solver is in matrix-vector products and scalar products—this type of functionality need only be implemented once, then other parts of the program can build on it.
- *Polymorphism*. The same code can be used for a variety of objects. For example, we would like to use similar looking C++ code to raise a matrix of complex numbers to a given power as we would to raise a real number to a given power—even though the basic arithmetic operations "behind the scenes" are different.
- *Inheritance*. This, perhaps the most important feature of object-orientation, allows for code reuse, extensibility and polymorphism. For example, a new linear solver for singular matrix systems will share many of the features of a basic linear solver. Inheritance allows the new solver to derive functionality from the basic solver, and then build on this functionality.

We are not going to discuss these terms in any more detail at this time. It is not that these things are unimportant. Quite the contrary—all these concepts add up to make C++ a very powerful language. However, we can cover the basics of programming without object-orientation. We will describe classes and objects in Chap. 6 and revisit some of these concepts. Then we can show exactly why *inheritance*, for instance, is so powerful when we come to explain it in Chap. 7.

1.1.2 Why You Should Write Scientific Programs in C++

Since you have selected a book with the words "C++" and "Scientific Computing" in the title, then the chances are that you have decided to start writing your scientific programs in C++. Perhaps not. Perhaps you are considering your options, or perhaps the choice of language has been foisted on you.

It is not our place to fight battles about which language is the very best, especially because the choice of language for a program will often depend on the problem that is being solved. In the field of numerical scientific programming, there are many languages being used, with most scientists opting for MATLAB®,[1] C/C++ or Fortran.

The first and most compelling reason for using C++ (as well as C and Fortran) is because they are *fast*. That is, with careful programming and optimisations, they can be compiled to a machine code program which is able to use the full power of the available hardware. Many scripting languages (such as MATLAB and Python) are *interpreted* languages, meaning that the code which you write is translated to machine code at run time. Other modern languages (such as Java and C#) compile halfway—to a hardware-independent byte-code which is then interpreted at run time. Run time interpretation means that some of the computer's power is spent on the conversion process and also that it is harder to apply optimisations. Nowadays MATLAB, Python and Java implementations use clever tricks such as caching compilation steps and just-in-time compilation to make programs run faster. Nevertheless, these tricks require computational effort and so these languages may not fully utilise the power of all hardware.

A second reason for using C++ is that there is a *wealth of numerical libraries* for scientific computing in C++ and related languages. Lots of numerical algorithms were established in the 1950s and were then incorporated into software libraries (such as EISPACK and LINPACK) in the 1970s.[2] If you write your own code using well-established, well-tested software then you are building on decades of experience and improvement.

A third reason for choosing to write in C++ is that there is a *wide-range of open source and commercial tools* to support you. We used the free GNU compiler tool-set to test the programs in this book and you can use any C++ compiler to compile them for your computer. In contrast, if we were distributing MATLAB programs, you would need to have MATLAB and a licence installed on your computer because it is a proprietary product. There are similar open source products (such as GNU Octave) but there is no guarantee that a MATLAB program will produce the same answer when run in Octave. Because it is closed source, the meaning of a program can change between versions of MATLAB. For example, when just-in-time compilation was introduced in MATLAB 7 the operational semantics of the language subtly changed. This meant that a small minority of MATLAB programs which were known to work well with one

[1] MATLAB is a registered trademark of The MathWorks, Inc.

[2] The original version of MATLAB was written in Fortran and was intended as a simple interface into parts of the EISPACK and LINPACK Fortran libraries.

version of MATLAB could produce incorrect results, errors or warnings on another version.

A fourth reason for C++ is that it has a *flexible memory management model*. In a Java program, some of the system memory is used in the interpretation and you rely on a garbage collector to tidy up memory which you are no longer using, and so you may not be able to predict how much memory a program is going to need. In C++ you can make this prediction, but this is a double-edged sword because you are also responsible for making sure that memory is managed properly.

A final reason to program in C++ is that it is an *object-oriented language*. We haven't yet told you what this means exactly, but it is widely held that writing in an object-oriented style leads to programs which are easier to understand, to extend, to maintain and to refactor.

1.1.3 Why You Should Not Write Scientific Programs in C++

It is worth stressing that C++ is not the best language for every occasion. Some people say that *other languages may be faster*. Many scientific programmers believe that Fortran will always give the best performance in terms of raw speed and would reject C++ on the basis that features such as pointer chasing and virtual method look-up (don't worry if you haven't heard of these terms, or don't know what they mean—you may never need to!) result in the code being executed at suboptimal speed. This may have some truth, but the fact that object-orientation leads to greater readability (as mentioned above) makes it a reasonable compromise language. It can be a very fast language and it is also a good language for readability.

Sometimes *other languages are better for a specialised task*. Scripting languages such as Perl and Python are ideal for text processing and string manipulation. If you need to sum columns of numbers from files then you could write a C++ program, but a short, disposable script would be far quicker to implement.

Some languages are *better for writing prototype programs or for plotting data*. MATLAB excels in the field of rapid prototyping—short programs to quickly explore some algorithm or phenomenon. To test a particular linear algebra algorithm on a range of matrices with various sizes and structures would take a few lines of MATLAB, but in C++ you might have to write several files and compile against someone else's libraries. MATLAB also has the advantage of a fully-integrated graphical development environment, making many programming tasks easy without having to rely on extra tools. Furthermore, MATLAB has an in-built plotting environment, so if you want to visualise the results of your algorithms quickly MATLAB might be your best choice.

So C++ may not be the best choice of language in *every* situation. However, there are many situations in which C++ has the ideal fit for a particular problem. The discussion above may be enough to convince you that it is worth getting started with C++.

1.1.4 Scope of This Book

Most C++ programs for scientific computing can be written very effectively by using only a fraction of the total capabilities of the language. This book focuses on the aspects of C++ that you are most likely to utilise, or to encounter in other programmer's code, for scientific computing applications. When writing your own programs, you may occasionally need to understand one of the more advanced features of the language. In the Further Reading section at the end of this book, we direct the reader to a collection of resources that provide a more comprehensive description of the whole C++ language [5–8].

1.2 A First C++ Program

It is very common to introduce a programming language by using a program that prints the text "Hello World" to the screen. A simple example of a C++ program that does this is shown below. The code in Listing 1.1 illustrates several basic features of C++ programs. In line 1 of this code, we include the header file `iostream`. The name `iostream` pertains to **i**nput and **o**utput **stream**ing and is required in any C++ program that inputs data from the keyboard or outputs data to the console, that is, the screen. The second feature to note is that there is a section of code that:

- begins with the line of code "`int main(int argc, char* argv[])`" (line 3 of this code);
- is followed by more code enclosed between curly brackets, { and }; and
- the code within the curly brackets ends with the statement "`return 0;`".

The section of the code between curly brackets contains the instructions that we want the computer to execute. The part of line 3 inside brackets allows us to execute the code using user-specified arguments: we will postpone a discussion of this functionality until Chap. 3. Note that comments have been inserted into the code in lines 5, 6, 7 and 9 to aid the reading of the code by humans: anything between the comment opener "`/*`" and the comment closer "`*/`", or any line that starts with "`//`" is a comment, and is ignored when the code is converted into an executable, computer readable file. We have used the extension `.cpp` for the code below to indicate that the file `HelloWorld.cpp` is a C++ program. Choice of this extension is entirely a matter of personal choice: other authors use the extensions `.C`, `.c++`, `.cxx` or `.cc`.

We now focus on the purpose of lines 10 and 12: these lines of code each contain an instruction to the computer, and are known as *statements*. Note that all statements end with a semi-colon. It is sufficient for the time being for the reader to know that line 10 is the line of code that directs the computer to print the contents within the quotation marks to the screen. The "`\n`" denotes a new line, and so the phrase "Hello World", followed by a new line, will be printed to the screen. The word `cout` is a contraction of **c**onsole **out**put, that is, printing to the screen.

Listing 1.1 `HelloWorld.cpp`

```
1  #include <iostream>
2
3  int main(int argc, char* argv[])
4  {
5    /* This is a comment and will be ignored by the compiler
6    Comments are useful to explain in English what
7    the program does */
8
9    // Print "Hello World" to the screen
10   std::cout << "Hello World\n";
11
12   return 0;
13 }
```

The word "`int`" at the start of line 3 indicates that the last line of the code within curly brackets will return an integer value. This is carried out using the statement in line 12 "`return 0;`". Returning the value zero indicates to the computer that the program has reached the end without encountering any problems.

Before moving on to explain how to get your computer to print `Hello World` to your screen we pause to discuss some stylistic issues of which you should be aware. You will see in the listing above that all lines of code within the curly brackets have been indented. This is not compulsory. However, it is standard practice when coding to indent these lines: this will become clearer in later chapters when we embed code within more than one set of curly brackets. The number of spaces indented is entirely for the programmer to decide: all spaces—termed "white space"—are ignored when executing the code above. A final point is that lines in C++ may be as long as the programmer wishes, and may run over the end of the line in the text editor used to write your C++ programs. For clarity, it is generally advisable to split a potentially long line over several lines. We will demonstrate this later when writing more complex statements.

The code in Listing 1.1 is a correct C++ program for printing the text "Hello World" to the screen. However, before this program may be executed it must first be translated into a format that the computer can read: this process is known as *compilation*. We now explain what compilation is, and how to do it.

1.3 Compiling a C++ Program

Many readers will have experience of scientific computing in MATLAB. A key difference between C++ and MATLAB is that a C++ program must be *compiled* before it can be executed. There are many different ways that compilation can be performed which we now discuss.

1.3.1 Integrated Development Environments

As you take your first steps in learning a new programming language, you may not want to invest a lot of time in installing new software and configuring applications to help you develop programs. For this reason, we recommend that you begin writing programs with your favourite text editor and a command line compiler (see the following Sect. 1.3.2). However, as your programs and projects grow in size you will need to manage multiple files each containing various parts of the program. This becomes difficult when the number of files becomes large, and you may spend a lot of time switching between files in order to look up what you called some function or argument. At this point in your code development, we would recommend that you switch to using an Integrated Development Environment (IDE).

Examples of IDEs that are available for C++ programmers at the time of writing include KDevelop for Linux, Microsoft Visual Studio for Windows, XCode for Mac OS X, and the cross-platform IDEs CLion and Eclipse. Eclipse is open source, runs on most operating systems and is well-maintained by a community of developers. Because it was originally built for developing Java programs, it is necessary to install a "C/C++ development tools plug-in" should it be used for developing C++ programs.

The functionality of various IDEs varies according to their level of sophistication, but most present the seasoned programmer with several advantages over an old-school compile at the command line approach. Common features of IDEs are listed below. Don't worry if you do not fully understand all the terms used: these will become clear as you work through this book.

1. A program editor with syntax highlighting such as keyword colouring, automatic code indentation and identification of illegal programming constructs.
2. Context aware editing, so that you immediately know what functionality is present in one of your classes as you type its name.
3. Build automation, where your entire project code is managed so that changes to small parts of a large program only result in small compilation steps. Build automation is traditionally done with a hand-crafted file known as a `Makefile`, which we introduce in Sect. 6.2.4.1. Many IDEs analyse your code for dependencies and then use a `Makefile` behind the scenes.
4. On-the-fly compilation gives the system the ability to constantly save and compile your program as you write it.
5. "Step through" graphical debugging lets you walk through a program as it runs, pause it at critical points, and examine the internal state of its variables. (More information on debuggers is given in Sect. 7.7.)
6. Automatic code generation is particularly useful in IDEs for graphical tool development. When the user selects that they want to include a button on a graphical tool in their program some "boiler plate" code is generated including the functions that are activated when the button is pressed—these are then filled in by the programmer.

1.3.2 Compiling at the Command Line

When using the Linux operating system,[3] C++ codes may be compiled and executed at the command line within a terminal window. Many compilers—both open source and commercially developed—are available. In this book, we assume that the reader has access to the GNU `gcc` compiler. To ensure that this compiler is installed, open a terminal window and type "`which g++`" followed by return. Hopefully the computer will respond by reporting the location of this compiler, for example,

```
$ which g++
/usr/bin/g++
$
```

If the compiler is not installed, it may be downloaded from https://gcc.gnu.org/, where instructions for installation may also be found.

To compile the code given in Listing 1.1, open a terminal window and create a directory where code may be saved. Move into this directory, and save the code as "`HelloWorld.cpp`". In the same directory type

```
g++ -o HelloWorld HelloWorld.cpp
```

In the command above, `g++` tells the computer that we want to use the GNU `gcc` compiler for C++. The section of the command "`-o HelloWorld`" tells the computer that we want to name the executable file "`HelloWorld`". The "`-o`" is known as the *flag* that the computer expects will be followed by the executable name, in this case `HelloWorld`. The command ends by stating the C++ file that we wish to compile. This command produces an executable file called `HelloWorld`. This executable may be run by typing "`./HelloWorld`" inside the terminal. Running this executable will result in the text "Hello World" being printed to the screen inside the terminal.

If we were to compile the code using the command above, but without the flag and the executable name, then an executable file would still be produced. A default name would be allocated to the executable file. For many compilers, this default executable name is `a.out`.

[3] If you are working on a Mac operating system, we recommend that you install the Xcode developer tool-set. This comes complete with a GNU C++ compiler which you can use on the command line or within the developer environment. If you are working on a Windows operating system, we recommend that you install MinGW (a minimal environment for using GNU tools within Windows). Alternatively, you may want something more sophisticated built on MinGW such as Cygwin (a Unix-like environment) or Code::Blocks (an open source windows development environment containing MinGW and the GNU C++ compiler).

1.3.3 Compiler Flags

If we were to attempt to compile a code that was not written using correct C++ syntax, then the compiler would report an error, and would not produce an executable file. As such, the compiler can be thought of as a helpful tool that has the capability to perform some validation of the correctness of the code.

Suppose we have written code where a calculation was stored as a variable, but this variable is never subsequently used. Although this may be written with correct C++ syntax it is likely that this is an error—we would expect that the result of every calculation will subsequently be used somewhere in the code, or there would be no point in performing this calculation. Compilers have the capacity to warn us of unexpected occurrences such as this by the use of *compiler flags*. The compilation command below will warn us of instances such as these.

```
g++ -Wall -o HelloWorld HelloWorld.cpp
```

The compiler flag `-Wall` above is a contraction of **w**arning **all**. The compilation command above will warn us of anything unexpected that is not actually an error, but will still create an executable file. We give an example instance of a situation in which the compiler will warn of a probable programming error as one of our programming tips in Sect. 2.6.3. Suppose we want to be stricter than this, and want the compiler to treat anything unexpected as an error and, therefore, not to create an executable file when this occurs. This may be achieved using the compilation command below.

```
g++ -Wall -Werror -o HelloWorld HelloWorld.cpp
```

There are a large number of compiler flags available for most compilers. At this stage, there is no need to know about any more than the basic flags. We have shown how to use compiler flags to perform some validation of the code written. We will now discuss three more flags that are particularly valuable when writing scientific computing applications. The first flag we discuss may be used to optimise the performance of the executable file. The default is no optimisation. By using the "`-O`" (upper case `o`) flag as shown below, the executable file should execute more quickly although compilation may take longer.

```
g++ -O -o HelloWorld HelloWorld.cpp
```

If we are debugging a program, it is important that the executable and the debugger have information about which line in the source code produced specific machine instructions. Normally this information is not retained after compilation. In order to produce a non-optimised version of the code with debugging information preserved, we use the "`-g`" flag.

```
g++ -g -o HelloWorld HelloWorld.cpp
```

The last flag that we introduce here is one that allows us to link to a library of mathematical routines. We instruct the compiler to link to this library using the command below.

```
g++ -lm -o HelloWorld HelloWorld.cpp
```

We may use as many flags as we wish when compiling—simply list them one after the other when compiling the code.

1.4 Variables

In the example code in Listing 1.1 we simply printed some text to the screen. In most programs, especially scientific computing applications, we wish to store entities and perform operations on them. These entities are known as *variables*. In C++ programs, in common with most compiled languages, the variables must be declared to be an appropriate type before they are used.

1.4.1 Basic Numerical Variables

The two most common types of variable that are used in scientific computing applications are *integers* and *double precision* floating point variables. Loosely speaking, if a numerical variable does not—and never will—require a decimal point it may be stored as an integer variable: if not it should be stored as a floating point variable. If a code uses two integers denoted by `row` and `column`, and one double precision floating point variable denoted by `temperature`, we may declare these before they are used, and set their values, using the following code fragment.

Listing 1.2 Declaring variables

```
    int row, column;
    double temperature;
    row = 1;
    column = 2;
    temperature = 3.0;
```

The statements in lines 1 and 2 of the code above allocate memory for two integer variables `row` and `column`, and one double precision floating point variable `temperature`. It is important to understand that, whilst memory is allocated for these variables, we do not know until we assign values to these variables in lines 3–5 what values are stored by these variables. A common mistake is to assume that these variables are initialised to zero when the memory is allocated: this is true some of the time, but you should not rely on this.

Note the use of the decimal point for the double precision floating point variable `temperature` in line 5 of the listing above. This is not strictly necessary, but emphasises that this variable is a floating point variable. Use of this decimal point has the advantage that, provided we compile the code with suitable flags, compilation will trigger a warning if we had mistakenly declared this variable to be an integer.

We strongly encourage the use of variable names that have some relation to the variable that they represent, for example `row` as a variable that contains the index to the row of a matrix (see Sect. 6.6 for a longer discussion of naming conventions for variables). There are certain rules that variable names in C++ must adhere to, but these rules are not particularly restrictive. The first rule is that all variables in C++ programs should begin with a letter. All other characters in variable names must be letters, numbers or underscores. Variable names are case–sensitive, and so "`ROW`" is a different variable to "`row`". We would not, however, recommend writing a program with one variable called "`ROW`" and another variable called "`row`" as the potential for confusing these variables is obvious. One final restriction is that some names, such as `int, for, return` may not be used as variable names because they are used by the language. These words are known as *reserved words* or *keywords*.

A variable may be *initialised* when defining the variable type. For example, the code fragment in Listing 1.2 may be written as the following code fragment.

```cpp
    int row = 1, column = 2;
    double temperature = 3.0;
```

The value of more than one variable may be assigned in each statement, as shown below.

```cpp
1  int row = 1, column = 2;
2  row = column = 3;
```

However, line 2 in the code fragment above may cause confusion—it actually means

```cpp
    int row = 1, column = 2;
    row = ( column = 3 );
```

and so both `row` and `column` take the value 3 after this fragment of code has been executed. However, it may be mistakenly read to be

```
int row = 1, column = 2;
( row = column ) = 3;
```

in which `row` would first take the value 2 (which was the initial value of column), and then `row`, because it is the *result* of the assignment `row = column`, would take the value 3. The value of column is unaffected. There is clearly potential for introducing errors when assigning more than one value in each statement, and so we do not recommend this approach.

It is often the case that a programmer intends a variable to be constant throughout the code, for example the numerical value used for the density of a fluid. The programmer can ensure that a variable is guaranteed to be unchanged throughout the code by assigning a value to the variable when it is declared, together with use of the keyword `const` as shown in the fragment of code below.

```
const double density = 45.621;
```

We may want to set the tolerance of some iterative solver to a very small number, for example 10^{-12}. Clearly, we may set this tolerance using the code fragment below.

```
double tolerance = 0.000000000001;
```

The listing above is clearly not ideal—a casual glance at the code does not allow us to distinguish easily between, say, 10^{-10} and 10^{-12}. It would be much clearer if we could write the numerical value in *scientific notation*. This is demonstrated in the code below.

```
double tolerance = 1.0e-12;
```

The letter "e" in the line of code above may be read as "times 10 to the power of": that is, 589.63 may be written `5.8963e2` as $589.63 = 5.8963 \times 10^2$.

1.4.2 Other Numerical Variables

In the previous section, we restricted ourselves to declaring all integer variables using the keyword `int` and all floating point variables using the keyword `double`. There are—however—variants on these variable types which we now discuss.

Integers can be declared as *integers*, *short integers* or *long integers* as shown below.

```
int integer1;
short int integer2;
long int integer3;
```

The actual range of integers that may be stored by each of these variables depends on the system that you are using. For example, on an obsolete 32-bit operating system the `long int` is completely synonymous with the `int` data type—but on modern 64-bit architectures the `long int` is assigned twice as much space as the `int` (so it can store numbers in the range $\pm 9 \times 10^{18}$ as opposed to $\pm 2 \times 10^{9}$).

Variables of type `short int` require the allocation of less memory, with a corresponding reduction in the range of values that may be stored in this memory. It may be tempting to try to use short integers where possible to free up as much memory as possible. We do not recommend this: in software written for scientific computing applications the bulk of memory allocated is usually used to store floating point variables. Reducing the memory allocated to integer variables is unlikely to free a significant volume of memory.

A further classification of each of the integer types is as *signed* or *unsigned* integers. Signed integers may be used to store both positive and negative integers, whilst unsigned integers may be used to store only nonnegative integers. These variables may be used as shown below.

```
signed long int integer4; // signed is unnecessary
unsigned int integer5;
```

The default for any integer is a signed integer, hence there is no purpose in explicitly declaring an integer as a signed integer. A variable of type `unsigned int` is allocated an identically sized memory location as a variable of type `int`. As would be expected, a variable of type `unsigned int` can then store a range of nonnegative integers roughly twice as big as a variable of type `int`. A programmer is, however, unlikely to notice the difference between these two variable types on modern systems.

Floating point variables may be declared using the keywords `float`, `double` or `long double` as shown below.

```
float floating_point_number1;
double floating_point_number2;
long double floating_point_number3;
```

As with integers, the range of numbers that may be stored using each of these variable types depends on the system used. On modern systems it is very rare that the range of numbers that may be stored by a variable of type `double` differs from the range that may be stored by a variable of type `long double`. In the remainder of this book, we do not distinguish between these data types. Variables of type `float` typically store a smaller range of numbers than those of type `double`. Although variables of type `double` require more memory we strongly urge writers of scientific computing applications to use double precision floating point variables: this will minimise the effect of rounding errors, thus removing one potential source of error from any program written.

1.4.3 Mathematical Operations on Numerical Variables

Sample C++ code for performing a variety of mathematical operations on variables is given below. Note the inclusion of the header file `cmath`. This file is needed for some mathematical operations and also includes values of some useful constants, such as `M_PI`, that contains the value of π correct to about 20 decimal places.

```
#include <cmath>

int main(int argc, char* argv[])
{
   double x = 1.0, y = 2.0, z;
   z = x/y;         // division
   z = x*y;         // multiplication
   z = sqrt(x);     // square root
   z = exp(y);      // exponential function
   z = pow(x, y);   // x to the power of y
   z = M_PI;        // z stores the value of pi

   return 0;
}
```

Many other mathematical functions are available. The functions `cos`, `sin`, `tan`, `acos`, `asin`, `atan`, `cosh`, `sinh`, `tanh`, `log`, `log10`, `ceil`, `floor` can be used in exactly the same way as `sqrt` and `exp` in the code above: that is, they accept one argument, and return one value.

Some mathematical functions deserve more explanation. This is done through their implementation in code below.

```
#include <cmath>

int main(int argc, char* argv[])
{
  double x = 7.8, y =  1.65, u = -3.4, z;
  z = fmod(x, y);    // remainder when x is divided by y
                     // z is 1.2 since 7.8 = 4*1.65 + 1.2
  z = atan2(y, x);   // inverse tangent (in radians) of
                     // angle between the vector
                     // (x, y) and the positive x-axis
                     // note the ordering of y and x in
                     // calling the function atan2
                     // z is 0.208465
  z = fabs(u);       // Absolute value of u
                     // z is 3.4
                     // note fabs should not be confused
                     // with abs (the integer equivalent)

  return 0;
}
```

There are many instances in scientific computing code where we wish to increment a variable a by the value b, that is, we want to replace the value that the variable a stores by the value a+b. There are shorthand operations for this and other similar operations in C++, shown in Table 1.1.[4] Note that the a%b operation, pronounced "a mod b", is a modulus operation and may be thought of as the remainder after dividing a by b using integer division as described in Sect. 1.4.4.

Table 1.1 Shorthand for some mathematical operations

Longhand	Shorthand
`a = a + b;`	`a += b;`
`a = a - b;`	`a -= b;`
`a = a * b;`	`a *= b;`
`a = a / b;`	`a /= b;`
`a = a % b;`	`a %= b;` if a and b are integers ($a \bmod b$)
`a = a + 1;`	`a++;` if a is an integer
`a = a - 1;`	`a--;` if a is an integer

[4]The "++" shorthand programming construct, which is also available in the C language, explains the original naming of the language "C++". It is a pun which means "like C but one better".

1.4.4 Division of Integers

One common error frequently made by inexperienced C++ programmers is in dividing an integer by another integer. Consider the fragment of code below.

```cpp
1    int i = 5, j = 2, k;
2    k = i / j;
3    std::cout << k << "\n";
```

This code fragment will output the value 2, when the value of dividing 5 by 2—that is, 2.5—was actually intended. There are two potential problems with the code fragment as it is written above. The first operation that will be performed when executing line 2 of the listing above is to divide the integer `i` by the integer `j`. The value resulting from this operation will then be stored in the memory allocated to `k`. In C++, division of an integer by another integer will return *only the integer part of this division*: hence dividing `i` by `j` will store the integer part of 2.5, which is 2 (as everything after the decimal point will be ignored). The second part of this statement—the assignment operator—will then assign the value 2 to the integer variable `k`.

It may be thought that modifying the code fragment above so that `k` is defined to be a double precision floating point variable may solve the problem, as shown in the code fragment below.

```cpp
1    int i = 5, j = 2;
2    double k;
3    k = i / j;
4    std::cout << k << "\n";
```

This still does not give the correct value of 2.5. This is because the division is performed in line 3 before the result is stored as the double precision floating point variable `k`. As division of an integer by another integer in C++ returns the integer part of the division, the division of `i` by `j` returns the value 2 as explained above. This value is then stored as the double precision floating point number 2.0 in the memory allocated to `k`.

To divide two integers as if they were floating point variables, we may convert the integers to double precision floating point variables as shown in the code fragment below.

```cpp
1    int i = 5, j = 2;
2    double k;
3    k = ((double)(i)) / ((double)(j));
4    std::cout << k << "\n";
```

The code `((double)(i))` is known as "*explicit type conversion*" and allows us to treat the integer variable `i` as a double precision floating point variable, and so this code fragment does output the correct value of 2.5.

1.4.5 Arrays

Many scientific computing applications are underpinned by algorithms that are based on vectors and matrices. These may be stored in C++ as an entity known as an *array*. If the size of the array is known in advance then it can be declared as follows.

```cpp
  int array1[2];
  double array2[2][3];
```

In the code fragment above, `array1` represents a vector of integers of length 2, whilst `array2` represents a matrix of double precision floating point variables of size 2×3.

In contrast to MATLAB and Fortran, in C++ the indices of an array of length `n` start with entry `0` and end with entry `n-1`. This is known as "*zero-based indexing*". Elements of an array are accessed by placing the indices in separate square brackets, and so we may completely populate the arrays `array1` and `array2` declared above using the following code.

```cpp
 array1[0] = 1; // Note that indexing begins from 0
 array1[1] = 10;
 array2[0][0] = 6.4;
 array2[0][1] = -3.1;
 array2[0][2] = 55.0;
 array2[1][0] = 63.0;
 array2[1][1] = -100.9;
 array2[1][2] = 50.8;
```

We may also perform operations on entries of the array as shown below.

```cpp
 array1[0]++; // increments the value of this entry by 1
 array2[1][2] = array2[0][1] + array2[1][0];
```

Arrays can be initialised when they are declared, for example,

```cpp
  double array3[3] = {5.0, 1.0, 2.0};
  int array4[2][3] = { {1, 6, -4}, {2, 2, 2} };
```

where the array `array3` represents the vector

$$\begin{pmatrix} 5 \\ 1 \\ 2 \end{pmatrix},$$

and `array4` represents the matrix

$$\begin{pmatrix} 1 & 6 & -4 \\ 2 & 2 & 2 \end{pmatrix}.$$

Note that the curly bracket notation may only be used to populate arrays at the same time as when they are declared—for example the code

```
int array5[3] = {0, 1, 2};
```

is acceptable, but the code

```
int array6[3];
array6 = {0, 1, 2};
```

will not be accepted by the compiler.

1.4.6 ASCII Characters

ASCII characters are numbers, uppercase letters, lowercase letters and some other commonly used symbols: most of the characters on your keyboard are ASCII characters. Variables that are ASCII characters are declared using the keyword `char`. Example code using an ASCII character is shown below.

```
#include <iostream>

int main(int argc, char* argv[])
{
  char letter;
  letter = 'a'; // note the single quotation marks

  std::cout << "The character is " << letter << "\n";

  return 0;
}
```

1.4.7 Boolean Variables

Boolean variables take either the value `true` or the value `false`. These variables are commonly used when specifying whether a portion of code should be executed in conjunction with `if` and `while` statements (which will be introduced in Chap. 2). Examples of Boolean variables are given below.

```
  bool flag1, flag2;
  flag1 = true;
  flag2 = false;
```

1.4.8 Strings

The data type `char` represents one ASCII character. A string may be thought of as an ordered collection of characters. For example, "C++" is a string consisting of the ordered list of characters "C", "+", and "+".

To use strings in C++ requires the header file `string`. The library which may be accessed using this header file contains significant functionality for the use and manipulation of strings. The bulk of coding for scientific computing applications requires operations on numerical variables, and so we do not discuss this data type in much detail. In the example code below, we demonstrate how to declare a string, how to determine the length of a string, how to access individual characters of the string, and how to print a string to the console.

A string in C++ is a little like an array of characters together with a layer of extra functionality. There is no need to understand *why* the length and elements of the string may be accessed in this way: an understanding of *how* is sufficient.

```
1  #include <iostream>
2  #include <string>
3
4  int main(int argc, char* argv[])
5  {
6    std::string city; // note the std::
7    city = "Oxford"; // note the double quotation marks
8    std::cout << "String length = " << city.length() << "\n";
9    std::cout << "Third character = " << city.at(2) << "\n";
10   std::cout << "Third character = " << city[2] << "\n";
11   std::cout << city << "\n"; // Prints the string in city
12   std::cout << city.c_str() << "\n"; // Also prints city
13 }
```

In line 9 and line 10 of the code recall that arrays in C++ have indices that begin from zero: `city.at(2)` and `city[2]` both refer to the entry of the array of characters with index 2, that is, “f”, the third letter of the string “Oxford”. Lines 11 and 12 both have the effect of printing the contents of `city` (“Oxford”) to the screen. Line 12 prints the contents of `city` to the screen, but does so by first converting from a C++ string to a C string, which is an array of type `char`. The string utility function `c_str` is not needed here, but is useful in cases where we need to pass a C++ string to a function which expects an array of type `char`.

1.5 Simple Input and Output

It would be pointless to write a code without having the means to communicate the output of the code to the user, or to some other application. As such, *output* is a programming technique that must be mastered by all programmers. Similarly, the user of software would expect to be provided with the ability to specify data that the software would use to generate output: *input* is therefore just as important a programming skill. We now describe basic C++ commands to allow output to the screen and input from the keyboard. In Chap. 3, we provide a fuller explanation, describing input from, and output to, a file, and a more flexible specification of the format of this output.

1.5.1 Basic Console Output

We have already briefly discussed console—or screen—output in Sect. 1.2, and have seen that the statement

```
std::cout << "Hello World\n";
```

prints the text “Hello World” to the screen, followed by a new line.

We may use `std::cout` to write more than one entity to the console at a time. This is best explained by example: consider the statements below.

```
int x = 1, y = 2;
std::cout << "x = " << x << " and y = " << y << "\n";
```

The second statement above tells the computer to first print the string “`x =` ”, followed by the value assigned to the variable `x`, then the string “ `and y =` ”, then the value assigned to the variable `y`, and finally to finish with a new line. The output is therefore

```
x = 1 and y = 2
```

Note that any spaces required in the output must be included within quotation marks in the statement that begins `std::cout`.

We have already seen one formatting command for output in C++: the new line formatting command `\n`. Some other useful formatting commands are shown in Table 1.2.

Table 1.2 Some formatting commands for console output

Command	Symbol
new line	`\n`
tab	`\t`
’	`\'`
”	`\"`
?	`\?`
bell sound	`\a`

Output from C++ is *buffered*. Sometimes, for example, if the computer is busy doing a large volume of computation, the program may not print the output to the screen immediately. If immediate output is desirable then use the statement “`std::cout.flush();`” after any `std::cout` command to ensure the output is printed before any other statements are executed, as shown in the listing below. As with certain aspects of string manipulation discussed in Sect. 1.4.8, at this stage it is sufficient to understand how to send output to the console immediately without worrying why it is done in this way.

```
   std::cout << "Hello World\n";
   std::cout.flush();
```

1.5.2 Keyboard Input

Keyboard input for numerical variables and characters is achieved using the input stream `std::cin`, where `cin` is a contraction of **c**onsole **in**. As with console output, the `iostream` header file must be included. The following code prompts someone to enter their Personal Identification Number—commonly known as their PIN—and then assigns the number entered to the integer variable `pin`.

```
  int pin;
  std::cout << "Enter your PIN, then hit RETURN\n";
  std::cin >> pin;
```

`std::cin` may be used to ask for more than one input at a time, as shown below.

```
  int account_number, pin;
  std::cout << "Enter your account number\n";
  std::cout << "and then your PIN followed by RETURN\n";
  std::cin >> account_number >> pin;
```

Keyboard input for variables of type `string` is slightly different. An example of how to input a string is given below. As with the commands for basic manipulation of strings given in Sect. 1.4.8, we do not attempt to explain why strings are input in this way: this will become clear when more advanced features of C++ are explained later in this book.

```
#include <iostream>
#include <string>

int main(int argc, char* argv[])
{
  std::string name;
  std::cout << "Enter your name and then hit RETURN\n";
  std::getline(std::cin, name);
  std::cout << "Your name is " << name << "\n";

  return 0;
}
```

1.6 The `assert` Statement

Scientific computing applications usually require a massive number of complicated mathematical computations. If any one of these computations is incorrect, then the final results of the computation will usually be incorrect. Finding the source of the error is an excruciatingly tedious process, and so we strongly recommend the use of the features of the C++ language that allow identification of unexpected occurrences such as an attempt to compute the square root of a negative number.

In Chap. 9 we point to the notion that there are various levels or degrees of error. In particular, we introduce *exceptions*, which are a feature of the C++ language that allow very effective handling of an unexpected occurrence when a code is being run. A less sophisticated approach is to use `assert` statements, as demonstrated in the code below. Note the inclusion of the extra header file `cassert` that is required to use `assert` statements.

```
1  #include <iostream>
2  #include <cassert>
3  #include <cmath>
4
5  int main(int argc, char* argv[])
6  {
7     double a;
8     std::cout << "Enter a non-negative number\n";
9     std::cin >> a;
10    assert(a >= 0.0);
11    std::cout << "The square root of "<< a;
12    std::cout << " is " << sqrt(a) << "\n";
13    return 0;
14 }
```

The code above invites the user to enter a nonnegative number, and returns the square root of this number. Before the square root is calculated, we check that the number really is nonnegative through the assert statement. We will see in Chap. 2 that the ">=" that appears in line 10 of the code is the "greater than or equal to" operator: this line of code therefore checks that the variable a is nonnegative. To see the effect of the assert statement, we first save the code as program.cpp and then compile the code without any optimisation flags to produce executable a.out. If, when this executable is run, the number -5 is entered, the code terminates at the assert statement with the following error message.

```
a.out:: program.cpp:10: int main(int, char**): Assertion 'a >= 0.0' failed
```

A further C++ function that is useful in conjunction with assertions is the function std::isfinite. This allows confirmation that a variable x contains a finite value, and not an infinite value (obtained, for example, by dividing a non-zero number by zero) or some other value that is not defined as a number (such as the square-root or logarithm of a negative number).[5] The use of this function along with an assert statement is illustrated in the code fragment below.

```
   double x;
   assert(std::isfinite(x));
```

Although we emphasise that this is a very rudimentary technique for identifying errors, and that we will introduce more sophisticated techniques later, assert statements can provide significant information: in the error message above we see that the exact line of code where the problem occurred has been identified. Another

[5] For those values which fail the std::isfinite test it is possible to differentiate between infinite numbers (using std::isinf) and those which are "not a number" (using std::isnan).

advantage of `assert` statements is that they can be automatically removed when the code is compiled with the "`-DNDEBUG`" flag. This allows you to test code with the assertions activated but to distribute a faster program that has the assertions deactivated by compiling using the command

```
g++ -DNDEBUG program.cpp
```

1.7 Tips: Debugging Code

There are many tools designed to aid with the debugging of code. The most basic of these is the compiler, and the flags associated with the compiler, as described in Sects. 1.3.2 and 1.3.3. More sophisticated tools exist, but they are aimed at larger scale projects, such as those that we will develop in later chapters of this book.

Rather than learning to use a sophisticated debugging tool whilst in the early stages of learning C++, we suggest below some simpler techniques for debugging the code that you will be writing when tackling the exercises in the early chapters of this book.

Compile your code frequently. Saving your code and compiling it using the warning compiler flag described in Sect. 1.3.3 every time a few statements are added is a useful diagnostic to see if any potential problems are being introduced. If there are any problems, comment out the new statements and recompile. Then add the statements in one at a time until the problem line is identified. When you first write code in C++ you may be amazed how often you forget the basic syntax such as a semi-colon at the end of a statement.

Save your project frequently. If you have code that works and you need to add new functionality, then do not throw away the old version. If things go wrong then you will be able to see exactly what you changed and if all else fails you will have a working version to roll back to. If it is critical that you are able to roll back to a working version of the code, or if you are in a collaborative project, we recommend that you use a version control system.[6]

Always test the code with a simple example. For example, if you are writing code to add the elements of two arrays verify the output by comparison with a calculation that you have carried out yourself.

Understand errors that arise when executing the code. If your program complains of a "segmentation error" when executing, it is likely that you have attempted to access a member of an array that is out-of-range: that is, you may have attempted to access the 6th entry of an array that was only declared to have 4 elements.

[6]There are many open source version control systems such as CVS, Subversion, Mercurial or Git to help you with this. There are also organisations who will host your code repository for you.

Use output. If you need to know where your program is crashing, and why, then print out some values of variables at key points in the execution. Do not forget to `flush` the output so that it appears before the program crashes!

Use assertions. If you expect a certain property at the start of a section of code, for example, that the scale factor is nonzero or that the argument of a square-root is nonnegative, you can check for it using assertions (introduced in Sect. 1.6).

C++ arrays are indexed beginning from zero. If the array `temperature` is declared as having 4 elements, the statement "`temperature[4] += 1.0;`" will cause problems.

Use a debugger. If all else fails then debug your program using a debugger. Tips on using a debugger are to be found in Sect. 7.7.

1.8 Exercises

1.1 To ensure that your compiler is correctly set up, copy and save the file `HelloWorld.cpp` displayed in Listing 1.1, compile it, and execute it.

1.2 Write code that asks a user to enter two integers from the keyboard and then writes the product of these integers to the screen.

1.3 Write code that declares two vectors as arrays of double precision floating point numbers of length 3 and assigns values to each of the entries. Extend this code so that it calculates the scalar (dot) product of these vectors and prints it to screen. Finally, extend the code so that it prints the Euclidean norm of both vectors to screen.
[*See Sect.* A.1.2 *for a definition of the scalar product, and Sect.* A.1.5 *for a definition of the Euclidean norm of a vector.*]

1.4 Write code that declares four 2×2 matrices of double precision floating point numbers, `A`, `B`, `C`, `D`, and assigns values to the entries of `A` and `B`. Let `C` = `A` + `B`, and `D=A*B`. Extend your code so that it calculates the entries of `C` and `D`, and then prints the entries of these matrices to screen.

1.5 Write code that invites the user to input separately strings that store their given name and their family name. Print the user's full name to screen.

1.6 I want to record the number of cars that drive past my house each day for five consecutive days, and calculate the average of these numbers. Create an integer array to store these five numbers, and then write code to calculate the average of these numbers. Execute your code using the sample data 34, 58, 57, 32, 43. Verify that you get the correct answer of 44.8.

[*Hint: read the material in Sect.* 1.4.4 *on converting integers to double precision floating point numbers*.]

1.7 Investigate the use of the compiler error warning flags discussed in Sect. 1.3.3. For example: (i) declare an integer as a constant variable and then attempt to change this value later in the code; and (ii) attempt to set an integer variable to the value 3.2.