Blocks, Functions and Reference Variables

5

The code developed in this book up to this point has been restricted to code that may be placed inside curly brackets after the initial line of code "`int main(int argc, char* argv[]);`". Readers with previous programming experience will be aware of the limitations this places when writing code. For example, if we were to apply the same operations in different places in the code we would have to repeat the lines of code that performed these operations everywhere in the code where they were required. It would be much more convenient if we could write a function that we could call whenever we wanted to perform these operations. This chapter introduces the C++ machinery for writing functions.

5.1 Blocks

A block is any piece of code between curly brackets. A variable, when declared inside a block, may be used throughout that block, but only within that block. This is demonstrated in the code below. In line 9, we attempt to use the variable `j` when it is only declared—and therefore available—in the block enclosed within the curly brackets in lines 4 and 8. In the language of programmers, "the scope of `j` is the block between lines 4 and 8". If we attempted to use the code fragment below, the compiler would report this attempted use of `j` as an error: `j` is said to be *out of scope* at line 9.

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_5

```
    {
       int i;
       i = 5;  // OK
       {
          int j;
          i = 10; // OK
          j = 10; // OK
       }
       j = 5;  // incorrect - j not declared here
    }
```

The same variable name may be used for a variable declared both inside a block—termed the *local variable*—and outside the scope of any function (including the `main` function)—termed the *global variable*. Both of these variables may be accessed inside the block as shown in the code below, using the example of both a global and a local variable called `i`. Furthermore, we may define a variable `j` in both the outer block and the inner block: inside the inner block the value of `j` stored by the variable declared in the outer block is not accessible. The multiple declaration of both `i` and `j` in the code below is bad programming practice, as it can clearly lead to confusion. In fact, since the scope of variables is so important, we suggest that variables are declared only within the block where they are needed, close to their first use. This multiple declaration of variables is known as *variable shadowing* and you can avoid it happening by turning on "shadow warnings" in your compiler. With the GNU `g++` compiler this is achieved by adding the `-Wshadow` flag to the compilation command.

```
#include <iostream>
int i = 5; // global i

int main(int argc, char* argv[])
{
  int j = 7;
  std::cout << i << "\n";
  {
    int i = 10, j = 11;
    std::cout << i << "\n"; // local value of i is 10
    std::cout << ::i << "\n"; // global value of i is 5
    std::cout << j << "\n"; // value of j here is 11
    //The other j (value 7) is inaccessible
  }
  std::cout << j << "\n"; // value of j here is 7
  return 0;
}
```

5.2 Functions

Now that we have defined what we mean by a block of code we may demonstrate how to write functions.

5.2.1 Simple Functions

A simple program that writes and uses a function to determine the minimum value of two double precision floating point variables x and y, and stores it in the double precision variable minimum_value is shown below. Note the *function prototype* that is line 3 in the listing below. The function prototype tells the compiler what input variables are required, and what variable, if any, is returned. In the example below, the function prototype explains that later in the code there will be a function called CalculateMinimum that requires two double precision floating point variables as input, and returns one double precision floating point variable. The function prototype can be thought of as being similar to declaring a variable. The variable names a and b in the prototype are ignored by the compiler and don't have to be included, but their inclusion can clarify the program. Note that the function prototype ends with a semi-colon.

Lines 15–29 of the code contain the statements that perform the tasks required by the function. This code begins with a line of code that is identical to the function prototype (including the variable names) without the semi-colon. After this there is a block of code that ends with a return statement that returns the value required to the point in the code where this function was called from. Note that there is no need to declare the variables a and b inside the function—the declaration in line 15 has done this already. Variables such as minimum that are used inside the function but are not part of the function prototype must be declared within the function block. Line 8 demonstrates how to call a function: the variables in brackets (x and y in this case) are sent to the function, and are known as the *arguments* of the function. The variable returned from the function is stored as minimum_value.

```
1  #include <iostream>
2
3  double CalculateMinimum(double a, double b);
4
5  int main(int argc, char* argv[])
6  {
7     double x = 4.0, y = -8.0;
8     double minimum_value = CalculateMinimum(x, y);
9     std::cout << "The minimum of " << x << " and " << y
10              << " is " << minimum_value << "\n";
11
12    return 0;
13 }
14
```

```
double CalculateMinimum(double a, double b)
{
   double minimum;
   if (a < b)
   {
      minimum = a;
   }
   else
   {
      // a >= b
      minimum = b;
   }

   return minimum;
}
```

Note that only one variable may be returned from a function. Although sufficient for some purposes, we may sometimes want to return more variables. We will see how this may be done later in this chapter. Of course, there are some circumstances where we do not want a function to return any variable: such functions may be prototyped as a `void` function. The code below contains an example of a function that prints out a message informing a candidate whether or not they have passed an exam. This function requires two integer variables as input: the first of these contains the mark that a candidate has scored; the second contains the pass mark for the exam.

```
#include <iostream>

void PrintPassOrFail(int score, int passMark);

int main(int argc, char* argv[])
{
   int score = 29, pass_mark = 30;
   PrintPassOrFail(score, pass_mark);

   return 0;
}

void PrintPassOrFail(int score, int passMark)
{
   if (score >= passMark)
   {
      std::cout << "Pass - congratulations!\n";
   }
   else
   {
      // score < passMark
      std::cout << "Fail - better luck next time\n";
   }
}
```

A function can only change the value of a variable sent to a function *inside* that function: changes made within the function will have no effect on this variable after the function has been executed and the code continues to execute statements in the block where the function has been called from. This is because a copy is made of any variable that is sent to a function, and it is this copy of the variable, and not the original variable, that is modified inside the function. For example, the following function has no effect on the variable `x` outside the function, even though the value of `x` *is* changed inside the function.

```cpp
#include <iostream>

void HasNoEffect(double x);

int main(int argc, char* argv[])
{
   double x = 2.0;
   HasNoEffect(x);
   std::cout << x << "\n";  // will print out 2.0

   return 0;
}

void HasNoEffect(double x)
{
   // x takes the value 2.0 here
   x += 1.0;
   // x takes the value 3.0 here
}
```

5.2.2 Returning Pointer Variables from a Function

In Sect. 5.2.1, we demonstrated how to write functions that returned either a variable that wasn't a pointer, or had no return type. Functions can be used to return pointer variables as well, as shown in the code below. In this case, we have written a function that allocates memory for a matrix dynamically, and returns the pointer to the memory allocated. The array can then be used as if the memory were allocated in the `main` function, as demonstrated in lines 8 and 9. Because every `new` requires a matching `delete` we have avoided leaking memory by also providing a function called `FreeMatrixMemory` to free up the memory created in `AllocateMatrixMemory`. Both `AllocateMatrixMemory` and its partner function `FreeMatrixMemory` operate in the manner described in Sect. 4.2.2.

```
double** AllocateMatrixMemory(int numRows, int numCols);
void FreeMatrixMemory(int numRows, double** matrix);

int main(int argc, char* argv[])
{
  double** A;
  A = AllocateMatrixMemory(5, 3);
  A[0][1] = 2.0;
  A[4][2] = 4.0;
  FreeMatrixMemory(5, A);
  return 0;
}

// Function to allocate memory for a matrix dynamically
double** AllocateMatrixMemory(int numRows, int numCols)
{
  double** matrix;
  matrix = new double* [numRows];
  for (int i=0; i<numRows; i++)
  {
    matrix[i] = new double [numCols];
  }
  return matrix;
}

// Function to free memory of a matrix
void FreeMatrixMemory(int numRows, double** matrix)
{
  for (int i=0; i<numRows; i++)
  {
    delete[] matrix[i];
  }
  delete[] matrix;
}
```

5.2.3 Use of Pointers as Function Arguments

We concluded Sect. 5.2.1 by explaining that any changes to a variable made inside a function would have no effect outside that function. This has the advantage that if a variable is altered unintentionally then the impact of this is localised to the function where this unintentional alteration was made. However, there are occasions where we *do* wish changes to a variable inside a function to have an effect outside a function. For example, if we are given a complex number in polar form, $z = re^{i\theta}$, we may wish to write a function that returns the real part, denoted by the variable `x`, and imaginary part, denoted by the variable `y`, of this number. We have noted earlier that a function can only return one variable, and so we may not return both the variable x and the variable y. It would therefore be useful to include the variables x and y in the function call. However, this would not work either, as the values assigned to these variables would not have any effect outside the function. Fortunately pointers provide us with one way around this problem. Instead of sending the variables x and y to the function, we send the *addresses* of these variables to the function. When the function is called, copies are made of the addresses of these variables, and it is these copies that are sent to the function. Changes to these addresses will not have any effect outside the function as we are working with a copy of these addresses. However, we can change the contents of the variable without changing the address through de-referencing the pointer, and this will have an effect outside of the function. This is demonstrated in the code below.

Note that lines 4–6 of the code are really meant to be one long line, giving the function prototype of CalculateRealAndImaginary. Since the line is long, we have split it across several lines and indented the continuation lines for clarity (see Sect. 6.6 for a discussion of stylistic conventions when writing code). The prototype lists the arguments for the function. The first two arguments are double precision floating point variables representing the magnitude (denoted by r) and argument (denoted by theta) of the specified complex number. The third and fourth arguments are pointers to—that is, the addresses of—the real part and imaginary part of the complex number. In line 12, we declare integers x and y that represent the real and imaginary parts of the complex number. To use the function CalculateRealAndImaginary, we send the addresses of these variables to the function. Behind the scenes a copy of these addresses is made, and it is these copies that are used in the function in lines 20–26. However, these copies refer to the same memory as the original variables x and y, and so it is this memory that the results of the calculations in lines 24 and 25 are stored in.

Listing 5.1 Use of pointers with functions

```
1  #include <iostream>
2  #include <cmath>
3
4  void CalculateRealAndImaginary(double r, double theta,
5                                 double* pReal,
6                                 double* pImaginary);
7
8  int main(int argc, char* argv[])
9  {
10    double r = 3.4;
11    double theta = 1.23;
12    double x, y;
13    CalculateRealAndImaginary(r, theta, &x, &y);
14    std::cout << "Real part = " << x << "\n";
15    std::cout << "Imaginary part = " << y << "\n";
16
17    return 0;
18 }
19
```

```
void CalculateRealAndImaginary(double r, double theta,
                               double* pReal,
                               double* pImaginary)
{
   *pReal = r*cos(theta);
   *pImaginary = r*sin(theta);
}
```

5.2.4 Sending Arrays to Functions

When sending arrays to functions—whether or not the memory has been allocated dynamically—it should be noted that it is the address of the first element of the array that is being sent to the function. In common with sending the pointer to a variable to a function, changes to this address will not have an effect in the code from which this function is called: however, the contents of this address—that is, the contents of the array—may be changed. As such, any changes made to an array inside a function *will* have an effect when that variable is used subsequently outside the function.

We begin by showing how to send arrays whose size is known at compile time to a function. This is shown in the listing below. Note that we do not have to specify the size of the first index of an array in the function prototype. This size is computed by the compiler. It may be included if desired, but this will be ignored when the code is compiled.

```
#include <iostream>
#include <cmath>

void DoSomething(double u[], double A[][10],
                 double B[10][10]);

int main(int argc, char* argv[])
{
   double u[5], A[10][10], B[10][10];

   DoSomething(u, A, B);

   // This will print the values allocated in
   // the function DoSomething
   std::cout << u[2] << "\n";
   std::cout << A[2][3] << "\n";
   std::cout << B[3][3] << "\n";

   return 0;
}

void DoSomething(double u[], double A[][10],
                             double B[10][10])
```

```
{
   u[2] = 1.0;
   A[2][3] = 4.0;
   B[3][3] = -90.6;
}
```

Arrays whose size has been dynamically allocated can also be sent to a function. Example code for this is shown below.

```
#include <iostream>
#include <cmath>

void DoSomething(double* u, double** A);

int main(int argc, char* argv[])
{
   double* u = new double [10];
   double** A = new double* [10];
   for (int i=0; i<10; i++)
   {
      A[i] = new double [10];
   }

   DoSomething(u, A);

   // This will print the values allocated in
   // the function DoSomething
   std::cout << u[2] << "\n";
   std::cout << A[2][3] << "\n";

   delete[] u;
   for (int i=0; i<10; i++)
   {
      delete[] A[i];
   }
   delete[] A;

   return 0;
}

void DoSomething(double* u, double** A)
{
   u[2] = 1.0;
   A[2][3] = 4.0;
}
```

5.2.5 Example: A Function to Calculate the Scalar Product of Two Vectors

Suppose we want to calculate the scalar product of two vectors of double precision floating point numbers of length *n*. Calculating the scalar product could be embedded within a function that inputs the two arrays, and the length *n* of both vectors, and returns a double precision floating point variable that represents the scalar product of the two vectors: see Sect. A.1.2 for a discussion of how to calculate the scalar product of two vectors. We would first need to allocate memory for the two vectors. We could then call the function that calculates the scalar product, before finally deleting the memory allocated to the two vectors. Code for this is shown below.

```cpp
#include <iostream>

double CalculateScalarProduct(int size, double* a,
                              double* b);

int main(int argc, char* argv[])
{
   int n = 3;
   double* x = new double [n];
   double* y = new double [n];
   x[0] = 1.0;  x[1] = 4.0;  x[2] = -7.0;
   y[0] = 4.4;  y[1] = 4.3;  y[2] = 76.7;
   double scalar_product = CalculateScalarProduct(n, x, y);
   std::cout << "Scalar product = "
             << scalar_product << "\n";
   delete[] x;
   delete[] y;

   return 0;
}

double CalculateScalarProduct(int size, double* a,
                              double* b)
{
   double scalar_product = 0.0;
   for (int i=0; i<size; i++)
   {
      scalar_product += a[i]*b[i];
   }
   return scalar_product;
}
```

5.3 Reference Variables

In Sect. 5.2.3, we demonstrated the use of pointers to allow changes made to a variable within a function to have an effect outside the function, and showed how this could be used to allow a function to, in effect, return more than one variable. An alternative to using pointers is to use *reference variables*: these are variables that are used inside a function that are a different name for the same variable as that sent to a function. When using reference variables any changes inside the function will have an effect outside the function. These are much easier to use than pointers: all that has to be done is the inclusion of the symbol & before the variable name in the declaration of the function and the prototype—this indicates that the variable is a reference variable. It is actually the case that references behave like pointers behind the scenes, but without the programmer having to convert to an address with & on the function call (as in Listing 5.1) and without having to de-reference inside the function—they provide a layer of syntactic sugar to ease the programmer's burden. We now modify the example code in Listing 5.1 that wrote a function that calculated the real and imaginary parts of a complex number given in polar form to use references instead of pointers.

```
#include <iostream>
#include <cmath>

void CalculateRealAndImaginary(double r, double theta,
                               double& real,
                               double& imaginary);

int main(int argc, char* argv[])
{
   double r = 3.4;
   double theta = 1.23;
   double x, y;
   CalculateRealAndImaginary(r, theta, x, y);
   std::cout << "Real part = " << x << "\n";
   std::cout << "Imaginary part = " << y << "\n";

   return 0;
}

void CalculateRealAndImaginary(double r, double theta,
                               double& real,
                               double& imaginary)
{
   real = r*cos(theta);
   imaginary = r*sin(theta);
}
```

5.4 Default Values for Function Arguments

If we are writing a function to implement an iterative technique, such as the Newton–Raphson technique for finding a root of a nonlinear equation, we will usually be content if the solution is accurate to within a tolerance of, say, 10^{-6}. Only on very rare occasions would we want to change this tolerance. We might also want to restrict the number of function evaluations: the Newton–Raphson iteration will probably be implemented using a `while` loop, and numerical rounding errors may prevent the error being sufficiently small for the iteration to terminate. Under these conditions, we would never exit the `while` loop, and the program that called this function would never terminate. It would therefore be prudent to write a function for implementing the Newton–Raphson technique that sets a default tolerance for the solution, and a default maximum number of iterations. We would then be able to call this function without specifying these default values. However, if we did want to call this function with different values then we would like to be able to do this. This is easily achieved by setting default values in the function prototype. This is demonstrated below in a program that uses the Newton–Raphson technique for calculating the cube root of a given number K through solving the nonlinear equation $f(x) = x^3 - K = 0$. Using a given initial guess x_0, the Newton–Raphson method results in the iteration

$$x_n = x_{n-1} - \frac{x_{n-1}^3 - K}{3x_{n-1}^2}, \quad n = 1, 2, 3, \ldots.$$

By setting default values for the tolerance and maximum number of function iterations we may call the function using one of: (i) the default values of these parameters; (ii) specifying the tolerance (the first optional parameter in the function prototype) and using the default maximum number of function iterations; and (iii) specifying both of these parameters. All three of these cases are shown below.

```
#include <cmath>
#include <iostream>

void CalculateCubeRoot(double& x, double K,
                       double tolerance = 1.0e-6,
                       int maxIterations = 100);

int main(int argc, char* argv[])
{
   double x = 1.0;
   double K = 12.0;

   // Calculate cube root using default values
   CalculateCubeRoot(x, K);

   // Calculate cube root using a tolerance of 0.001 and the
   // default maximum number of iterations
   double tolerance = 0.001;
   x = 1.0; // Restart guess
   CalculateCubeRoot(x, K, tolerance);

   // Calculate cube root using a tolerance of 0.001 and a
   // maximum number of iterations of 50
   int maxIterations = 50;
   x = 1.0; // Restart guess
   CalculateCubeRoot(x, K, tolerance, maxIterations);

   return 0;
}

void CalculateCubeRoot(double& x, double K,
                       double tolerance, int maxIterations)
{
   int iterations = 0;
   double residual = x*x*x-K;
   while ((fabs(residual) > tolerance) &&
          (iterations < maxIterations))
   {
      x = x-(x*x*x-K)/(3.0*x*x);
      residual = x*x*x-K;
      iterations++;
   }
}
```

5.5 Function Overloading

Suppose we want to write one function to multiply a vector by a scalar, and another function to multiply a matrix by a scalar. It would seem natural to call both these functions `Multiply`. This is allowed in C++: we write different function prototypes and functions for both of these operations: the compiler then chooses the correct function based on the input arguments. This is demonstrated in the code below, and is known as *function overloading*.

```
#include <iostream>

void Multiply(double scalar, double* u, double* v, int n);

void Multiply(double scalar, double** A, double** B, int n);

int main(int argc, char* argv[])
{
   int n = 2;
   double* u = new double [n];
   double* v = new double [n];
   double** A = new double* [n];
   double** B = new double* [n];
   for (int i=0; i<n; i++)
   {
      A[i] = new double [n];
      B[i] = new double [n];
   }

   u[0] = -8.7;  u[1] = 3.2;
   A[0][0] = 2.3;  A[0][1] = -7.6;
   A[1][0] = 1.3;  A[1][1] = 45.3;
   double s = 2.3, t = 4.8;

   // vector multiplication
   Multiply(s, u, v, n);

   // matrix multiplication
   Multiply(t, A, B, n);

   delete[] u;
   delete[] v;
   for (int i=0; i<n; i++)
   {
      delete[] A[i];
      delete[] B[i];
   }
   delete[] A;
   delete[] B;

   return 0;
}

void Multiply(double scalar, double* u, double* v, int n)
{
   // v = scalar*u (scalar by vector)
   for (int i=0; i<n; i++)
   {
      v[i] = scalar*u[i];
   }
}

void Multiply(double scalar, double** A, double** B, int n)
{
   // B = scalar*A (scalar by matrix)
   for (int i=0; i<n; i++)
   {
      for (int j=0; j<n; j++)
      {
         B[i][j] = scalar*A[i][j];
      }
   }
}
```

Note that we can overload functions based only on the number and type of the *arguments* and not on the return type. This means that we could not have vector multiply function `bool Multiply(double scalar, double* u, double* v, int n)` alongside the version which has a `void` return type. This is because the compiler can infer the correct version of an overloaded function from the types of its arguments from the context in which it is used. This is not the case with the return type, where you may want to call a function which returns something, but then to cast its output to another return type, or ignore its output completely.

5.6 Declaring Functions Without Prototypes

It is good practice to give the function signature prototypes before you write the implementation. This is so that the function `main`, or any other function will recognise the name and argument types of the new function. However, it is possible to skip the writing of the function prototype by writing the function implementation before its first use, as is shown in the code below.

```
#include <iostream>

double Square(double x)
{
   return x*x;
}

int main(int argc, char* argv[])
{
   std::cout << "Square of 2 = " << Square(2) << "\n";
   return 0;
}
```

If prototypes are not given, then the function implementations must be ordered in such a way that each implementation is seen by the compiler before its first use. Note that if two functions are mutually recursive, that is, both functions call the other function, then it will not be possible to order the functions in this way—and so prototypes must be declared in this case.

5.7 Function Pointers

Suppose we want to write a function to implement the solution of the nonlinear equation $f(x) = 0$ using the Newton–Raphson technique, where f is a user-specified function. We may want to call this function for solving nonlinear equations more than once during the execution of a given program, and for different user-specified nonlinear functions. To achieve this, we need to specify the appropriate nonlinear function each time the function is called. This may be done, as demonstrated in the code below, using *function pointers*.

In the code below, we specify two functions `myFunction` and `myOther-Function`. In line 8, we declare a *function pointer* `*p_function`. This declaration specifies that the function that this pointer refers to must: (i) accept one (and only one) input argument which is a double precision floating point variable; and (ii) return one double precision floating point variable. In line 10, we specify that `p_function` points at the function `myFunction`: calling the function `p_function` in line 11 then has an identical effect to calling `myFunction`. In lines 13 and 14, we demonstrate how to use `p_function` to subsequently call the function `myOtherFunction`.

```
 1  #include <iostream>
 2
 3  double myFunction(double x);
 4  double myOtherFunction(double x);
 5
 6  int main(int argc, char* argv[])
 7  {
 8     double (*p_function)(double x);
 9
10     p_function = &myFunction;
11     std::cout << p_function(2.0) << "\n";
12
13     p_function = &myOtherFunction;
14     std::cout << p_function(2.0) << "\n";
15
16     return 0;
17  }
18
19  double myFunction(double x)
20  {
21     return x*x;
22  }
23
24  double myOtherFunction(double x)
25  {
26     return x*x*x;
27  }
```

The Newton–Raphson method for solving nonlinear equations is defined in Exercise 2.6 in the Exercises at the end of Chap. 2. This is implemented below for two different user-specified functions through the use of function pointers. In lines 5–16, we write a function to implement this algorithm. This function requires specification of: (i) a function pointer to the nonlinear function; (ii) a function pointer to the derivative of the nonlinear function; and (iii) an initial guess to the solution. Note that the function as it stands does not check for divergence, so is unsafe to use in some cases.

In lines 46 and 47, we call the Newton–Raphson solver to solve the equation $\sqrt{x} - 10 = 0$ with initial guess $x = 1$: the nonlinear function `Sqrt10`, and the derivative of the nonlinear function `Sqrt10Prime` are given in lines 19–22 and 26–29 of the code. Similarly, in lines 48 and 49 we call the Newton–Raphson solver to solve the equation $x^3 - 10 = 0$ with initial guess $x = 1$: the nonlinear function `Cube10`, and the derivative of the nonlinear function `Cube10Prime` are given in lines 32–35 and 39–42 of the code.

```
1  #include <cmath>
2  #include <iostream>
3
4  // Implementation of Newton-Raphson iteration
5  double SolveNewton(double (*pFunc)(double),
6                     double (*pFuncPrime)(double),
7                     double x)
8  {
9     double step;
10    do
11    {
12       step = (*pFunc)(x)/(*pFuncPrime)(x);
13       x -= step;
14    } while (fabs(step) > 1.0e-5);
15    return x;
16 }
17
18 // Function to calculate x that satisfies sqrt(x)=10
19 double Sqrt10(double x)
20 {
21    return sqrt(x) - 10.0;
22 }
23
24 // Derivative of function to calculate x that satisfies
25 // sqrt(x)=10
26 double Sqrt10Prime(double x)
27 {
28    return 1.0/(2.0*sqrt(x));
29 }
30
31 // Function to calculate x that satisfies x*x*x=10
32 double Cube10(double x)
33 {
34    return x*x*x - 10.0;
35 }
36
37 // Derivative of function to calculate x that satisfies
38 // x*x*x=10
39 double Cube10Prime(double x)
```

```
{
   return 3.0*x*x;
}

int main(int argc, char* argv[])
{
   std::cout << "Root sqrt(x)=10, with guess 1.0 is "
             << SolveNewton(Sqrt10,Sqrt10Prime,1.0) << "\n";
   std::cout << "Root x**3=10, with guess 1.0 is "
             << SolveNewton(Cube10,Cube10Prime,1.0) << "\n";
   return 0;
}
```

5.8 Recursive Functions

In some applications, we may wish to call a function from within the same function: this is known as *recursion*, and is possible in C++. A simple application of this is the calculation of the factorial of a positive integer n, denoted by fact(n), and written mathematically as $n!$, which is defined by

```
fact(n) = n × fact(n-1),    n > 1,
fact(n) = 1,                n = 1.
```

Code to implement this recursive definition of the factorial function is given below: we simply call the function CalculateFactorial from within the same function as many times as required.

```
#include <iostream>
#include <cassert>

int CalculateFactorial(int n);

int main(int argc, char* argv[])
{
   int n = 7;
   std::cout << "The factorial of " << n
             << " is " << CalculateFactorial(n) << "\n";

   return 0;
}

```

```
int CalculateFactorial(int n)
{
   assert (n > 0);
   if (n == 1)
   {
      return 1;
   }
   else
   {
      // n>1
      return n*CalculateFactorial(n-1);
   }
}
```

5.9 Modules

Suppose we want to write a code to allow us to solve linear systems of the form **Ax** = **b**, where **A** is a square, invertible matrix of size n, **b** is a specified vector of size n, and **x** is a vector to be calculated of size n. It would be useful if we could write all the functions required to solve this linear system and then allow these functions to be called through an appropriate function—that is, we want to write a function called `SolveLinearSys` with the prototype shown below.

```
void SolveLinearSys(double** A, double* x, double* b, int n);
```

The function `SolveLinearSys` has all the information required to solve the linear system, and any functions required can be called from within this function. This then allows us to solve any suitably defined linear system using just the single line of code shown below.

```
   SolveLinearSys(A, x, b, n);
```

The function `SolveLinearSys`, and all other functions associated with this linear solver, are known as a *module*. In more concrete terms, a module is a collection of functions that performs a given task. Every module has an *interface*. In the example above, this was defined by the prototype of the function `SolveLinearSys`, and may be thought of as a list of variables that contains: (i) those that must be input to the module; and (ii) those that are output by the module.

Modules are very useful when sharing code. For example, if a colleague has written code for solving linear systems as described above then it would be a very simple task for another colleague to utilise this code. All that is required is an understanding of the interface and what the purpose of the code is: there is no need to understand the mathematical algorithm that determines *how* the linear system has been solved, and the module may be thought of as a “black box”.

5.10 Tips: Code Documentation

As you begin to write more programs, there is often a temptation to “just get on with the coding” without paying specific attention to quality. After all “you generally know where you are going and understand the program which you are writing”. It is important to bear in mind, though, that your code will not always be as well understood as it is now. You might come back to a given file in three years’ time, because you need to correct it or to add some new functionality to it. Alternatively, you may at some stage hand your programs over to someone else who has the job of working out what you were doing.

Our tip in this chapter is that computer programs should be human-readable, as well as machine-readable. Even the smallest portion of code may prove to be opaque unless we include enough commentary to aid the human reader. Take for example the function given below, which calculates the p-norm of a vector. Without comments in the code, it would not be obvious what was happening, even though there are only a few lines of code. A hint is given in the name of the function, `CalculateNorm`, but what is it meant to do? What is the significance of the arguments `s` and `p`?

```cpp
#include <cmath>
double CalculateNorm(double* x, int s, int p)
{
   double a = 0.0;
   for (int i=0; i<s; i++)
   {
      double temp = fabs(x[i]);
      a += pow(temp, p);
   }
   return pow(a, 1.0/p);
}
```

In the code segment below, we give a description of the function immediately before its definition. This description gives, in line 3, a means of mapping the mathematics of the function to its implementation. The rest of the description gives an alternate place to find more information about the p-norm (lines 4–6) and an explanation of some of the arguments as necessary. In the body of the function, the loop has been commented to describe what its *functional purpose* is: it is about computing a sum over the elements of the vector. Finally, the return value is commented with a few words of explanation.

```
1  #include <cmath>
2  //  Function to calculate the p-norm of a vector:
3  //      =  [ Sum_i  ( |x_i|**p )  ]  **(1/p)
4  //  See "An Introduction to Numerical Analysis" by
5  //  Endre Suli and David Mayers, page 60, for definition
6  //  of the p-norm of a vector
7  //  x is a pointer to the vector which is of size vecSize
8
9  double CalculateNorm(double* x, int vecSize, int p)
10 {
11    double sum = 0.0;
12    //Loop over elems x_i of x, incrementing sum by |x_i|**p
13    for (int i=0; i<vecSize; i++)
14    {
15       double temp = fabs(x[i]);
16       sum += pow(temp, p);
17    }
18    //Return p-th root of sum
19    return pow(sum, 1.0/p);
20 }
```

Note that documenting code is sometimes more of an art than a science. There is a balance to be struck concerning the right level of documentation. Too many comments can make the program less readable rather than more readable. Our tip here is that you should describe what part of the problem the code is solving and, perhaps, *how* it is solving that problem. Do not be tempted to describe the code in overmuch detail. For example, the comment on the loop in line 12 of code above could have read

```
12    //  Loop over values of i going from 0 to vecSize-1
```

While this comment is accurate (describing the range of the loop variable `vecSize`), it does nothing to aid a programmer in their understanding of the code.

The formatting of the code documentation can also help readability. A simple tip is that using empty lines to break code and comments into sections can make the code look more readable. If you want to emphasise something you can simulate underlining with hyphens or underscores, for example,

```
4     //  Very important comment
5     //  ----------------------
```

Alternatively, you can emphasise something by putting it in a box:

```
  /**********************************************
   **********************************************
   **           CalculateNorm(...)            **
   **                                          **
   **   Function to calculate p-norm of vector  **
   **********************************************
   **********************************************/
```

5.11 Exercises

In all exercises, we suggest that you use dynamic allocation of memory for vectors and matrices as described in Sect. 4.2. Be sure that you are correctly de-allocating memory when using dynamic allocation of memory, as explained in the exercises at the end of Chap. 4.

5.1 Write code that sends the *address* of an integer to a function that prints out the *value* of the integer.

5.2 Write code that sends the address of an integer to a function that changes the value of the integer.

5.3 Write a function that swaps the values of two double precision floating point numbers, so that these changes are visible in the code that has called this function.

1. Write this function using pointers.
2. Write this function using references.

5.4 Write a function that can be used to calculate the mean and standard deviation of an array of double precision floating point numbers. Note that the standard deviation σ of a collection of numbers x_j, $j = 1, 2, \ldots, N$ is given by

$$\sigma = \sqrt{\frac{\sum_{j=1}^{N} (x_j - \bar{x})^2}{N - 1}}$$

where $\bar{x}$ is the mean of the numbers.

5.5 Write a function `Multiply` that may be used to multiply two matrices given the matrices and the size of both matrices. Use assertions to verify that the matrices are of suitable sizes to be multiplied.

5.6 Overload the function `Multiply` written in the previous exercise so that it may be used to multiply:

1. a vector and a matrix of given sizes;
2. a matrix and a vector of given sizes;
3. a scalar and a matrix of a given size; and
4. a matrix of a given size and a scalar.

5.7 The p-norm of a vector $\mathbf{v}$ of length n is given by

$$\|\mathbf{v}\|_p = \left(\sum_{i=1}^{n} |v_i|^p\right)^{1/p}$$

where p is a positive integer. Extend the code in Sect. 5.10 to calculate the p-norm of a given vector, where p takes the default value 2.

5.8 The determinant of a square matrix may be defined recursively: see Sect. A.1.3. Write a recursive function that may be used to calculate the determinant of a square matrix of a given size. Check the accuracy of your code by comparison with the known formulae for square matrices of size 2 and 3:

$$\begin{aligned}
\det\begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix} &= A_{00}A_{11} - A_{01}A_{10}, \\
\det\begin{pmatrix} A_{00} & A_{01} & A_{02} \\ A_{10} & A_{11} & A_{12} \\ A_{20} & A_{21} & A_{22} \end{pmatrix} &= A_{00}(A_{11}A_{22} - A_{12}A_{21}) - A_{01}(A_{10}A_{22} - A_{12}A_{20}) \\
&\quad + A_{02}(A_{10}A_{21} - A_{11}A_{20}).
\end{aligned}$$

5.9 Write a module for solving the 3×3 linear system $\mathbf{Au} = \mathbf{b}$ where $\mathbf{A}$ is nonsingular.

5.10 Write a module for solving the $n \times n$ linear system $\mathbf{Au} = \mathbf{b}$ using Gaussian elimination with pivoting, where $\mathbf{A}$ is nonsingular. See Sect. A.2.1.3 for details of this algorithm.