3 File Input and Output

Being able to transfer data between applications is an essential requirement of most scientific computing software. For example, data defining the boundary of an object may be generated from an image processing application. This data may subsequently be used by many applications written by a variety of users. To allow exchange of data between applications in this manner requires us to store data in a clearly specified format. Reading and writing files to a given specification therefore plays a key role in scientific computing applications, and is the subject of this chapter.

3.1 Redirecting Console Output to File

We introduced basic C++ commands for writing text and the contents stored by a variable to the console in Sect. 1.5. On a Linux system this output may very easily be redirected to a single file rather than the screen. Should the executable file be called `SampleCode`, this output may be printed to the file `SampleOutput.txt` by executing at the command line, as described in Sect. 1.3.2, with the executable name being followed by a specification of the file to be written to, as shown below:

```
$./SampleCode > SampleOutput.txt
$
```

When output has been redirected to file in this way, you may prefer to print to screen errors encountered by the program. This can be done using `std::cerr` as shown below. The word `cerr` is a contraction of **c**onsole **err**or.

```
int x, y;
if (y == 0)
{
  std::cerr << "Error - division by zero\n";
}
else
{
  // y not zero
  std::cout << x/y << "\n";
}
```

The syntax for `std::cerr` is identical to that for `std::cout`. When the console output is not redirected to file there is no difference between the effect of these two commands. However, when output is redirected to a specified file, only the `std::cout` statements are redirected: the output from a `std::cerr` statement will still be printed to the screen. Should output from the code above be redirected to file, then the value given by dividing `x` by `y` will be written to the specified file unless the variable `y` takes the value zero. Under these circumstances, the message "`Error - division by zero`" will be printed to the screen instead.

3.2 Writing to File

In the previous section, we explained how all the output of an application may be printed to a single file. This may be adequate for some applications, but is definitely not adequate for all applications. For example, were we to write a code to calculate the finite element solution of a given differential equation we may want to store the nodes of the mesh in one file, the connectivity array defining the elements in another file, the finite element solution in another file, and—perhaps—the nodes comprising the individual faces of the elements in another file. We therefore need to be able to write output to more than one file. Although C++ offers an extremely large number of commands for printing to file, almost all file formats can be achieved by using a very small subset of these commands.

Writing to, or reading from, file requires the additional header file `fstream`. In the code below, we show how to write to file. We first declare an *output stream variable* `write_output` by specifying it as being of type `std::ofstream`, and also specify the filename "`Output.dat`" as shown in line 9. Line 10 then checks that the file has been successfully opened: we return to this point below. Writing to file is similar to console output, but replacing `std::cout` with `write_output` in line 13: this writes the entries of the arrays `x` and `y` to the file associated with the output stream variable, in this case `Output.dat`. Finally, in line 15, when all required data has been written to file, we "close the file handle". In Sect. 1.5.1, we explained that console output is buffered, and so the output may not immediately be written to the console. Output to file is also buffered: closing the file handle *flushes* the buffer: that is, all data that has been buffered is written to file before the computer executes any further statements. It is important that this is done: if another part of the program reads a file which is still being written to, then we cannot be certain what data—if any—has yet been written to disk. Closing the file handle has the further effect that no more data can be written to this file: this prevents the file being corrupted by mistakenly attempting to write further data. We note at this point that explicitly closing the file handle on line 15, and in many of our later examples, is actually redundant for the simple reason that the call to `close()` will be run automatically as the file handle is tidied when the `main` function finishes. However it is good practice for the novice programmer to make this call explicitly and thereby to know when to expect output from their program to be written to file.

Listing 3.1 Basic writing to file

```
 1 #include <cassert>
 2 #include <iostream>
 3 #include <fstream>
 4
 5 int main(int argc, char* argv[])
 6 {
 7   double x[3] = {0.0, 1.0, 0.0};
 8   double y[3] = {0.0, 0.0, 1.0};
 9   std::ofstream write_output("Output.dat");
10   assert(write_output.is_open());
11   for (int i=0; i<3; i++)
12   {
13     write_output << x[i] << " " << y[i] << "\n";
14   }
15   write_output.close();
16   return 0;
17 }
```

It is also possible to flush a buffer without closing the file handle. This is done in a similar way as for console output in Sect. 1.5.1, and is demonstrated below for the output stream variable `write_output`.

```
  write_output.flush();
```

We explained above that it is important to check that a file has been opened (line 10 of the Listing 3.1) before attempting to write any data to it. If the file cannot be opened—perhaps we did not have permission to write to that file, or a directory we have specified does not exist—then writing to the `ofstream` may cause no error even though writing to the file is not possible. For example, if in line 9 we renamed the location of the output file to a folder we are restricted from writing to as follows:

```
9   std::ofstream write_output("/etc/Output.dat");
```

then we might expect the program to fail as we are unlikely to have permission to write to the folder `/etc/`. However, without the test for the file being open the code will exit normally, producing no output file. This would clearly be very frustrating for the user of the code.

The executable created from Listing 3.1 will create a new file, `Output.dat`, if this file does not already exist. If this file does exist, the executable generated from the listing above will delete the original file and write a new file with the same name: the original contents of the file will be lost.[1] Whether or not the file `Output.dat` existed before the code above was executed, after execution there will be a file called `Output.dat` that is listed below.

Listing 3.2 The file `Output.dat`

```
0   0
1   0
0   1
```

The code in Listing 3.1 may do what was required, but it may not. Suppose that, rather than deleting the file if it exists, we want our code to append data to the end of this file. This would be achieved by modifying line 9 of Listing 3.1 to

```
9   std::ofstream write_output("Output.dat", std::ios::app);
```

If the file `Output.dat` did not exist and we were to execute the code in Listing 3.1, with line 9 modified as shown above, we would then create the file `Output.dat` shown in Listing 3.2. If we were then to execute the code a second time we would then end up with the file `Output.dat` being modified as shown in Listing 3.3 below.

Listing 3.3 Modified file `Output.dat`

```
0   0
1   0
0   1
0   0
1   0
0   1
```

[1] If you want to check for the existence of a file before opening an output stream to it then a simple thing to do is to first attempt to read from it. See Exercise 3.1

3.2.1 Setting the Precision of the Output

The key formatting command for scientific computing applications is specification of the precision of the output. This is demonstrated in the listing below. The number in brackets after the `precision` commands specifies the number of significant figures that the output is correct to. Note that when the precision is set to 10 significant figures in line 15 of the listing below only eight significant figures will be printed: this is because the variable `x` is only given to eight significant figures, and so the remaining accuracy requested is redundant.

```
 1 #include <iostream>
 2 #include <fstream>
 3
 4 int main(int argc, char* argv[])
 5 {
 6   double x = 1.8364238;
 7   std::ofstream write_output("Output.dat");
 8
 9   write_output.precision(3); // 3 sig figs
10   write_output << x << "\n";
11
12   write_output.precision(5); // 5 sig figs
13   write_output << x << "\n";
14
15   write_output.precision(10); // 10 sig figs
16   write_output << x << "\n";
17   write_output.close();
18
19   return 0;
20 }
```

3.3 Reading from File

When reading from file we first need to declare an *input stream variable* in a similar way to the output stream variable described in Sect. 3.2, and then specify the file that we wish to read. As with output to file, the header file `fstream` should be included. Reading the file is then performed in a similar way to that described for keyboard input in Sect. 1.5.2, with `std::cin` replaced by the input stream variable. Suppose we want to input the file `Output.dat` shown in Listing 3.3. We know that this file has six rows and two columns, and so we may read this file using the code shown in Listing 3.4. The assertion in line 9 ensures that `Output.dat` is on disk in the correct location and with the correct access privileges: if not, the assertion is tripped and the code is terminated.

Listing 3.4 Reading column formatted data

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

int main(int argc, char* argv[])
{
  double x[6], y[6];
  std::ifstream read_file("Output.dat");
  assert(read_file.is_open());
  for (int i=0; i<6; i++)
  {
    read_file >> x[i] >> y[i];
  }
  read_file.close();
  return 0;
}
```

In the code above, we knew that the file we were reading had six rows and two columns, and so we knew when writing this code that the statements inside the `for` loop had to be executed six times. In many scientific computing applications we will want to read a file, but do not know the length of the file in advance. For example, we may know that a file contains a list of the coordinates of an unknown number of points in two dimensions: the file therefore has two columns, but an unknown number of rows. We cannot use a `for` loop as we do not know how many times the statements in this loop need to be executed. Instead, we use the Boolean variable associated with the input stream variable `read_file.eof()`. This variable takes the value `true` when the **e**nd **o**f the **f**ile is reached, and allows us—through the use of a `while` statement—to carry on reading the file while this variable takes the value `false`. Assuming that we know that the number of points is fewer than 100, we may achieve this using the following code. Note that a potential problem with this code as given will be addressed in Exercise 3.2.

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

int main(int argc, char *argv[])
{
  double x[100], y[100];
  std::ifstream read_file("Output.dat");
  assert(read_file.is_open());

  int i = 0;
  while (!read_file.eof())
  {
    read_file >> x[i] >> y[i];
    i++;
  }
  read_file.close();
  return 0;
}
```

One additional feature of reading from file that is of use when writing scientific computing applications is the ability to *rewind* a file so that we can read a file starting from the beginning again. This may be achieved by inserting the statements below into the code at the point where the file should be rewound.

```
  read_file.clear();
  read_file.seekg(std::ios::beg);
```

3.4 Checking Input and Output are Successful

In Sect. 3.2 we advised C++ programmers to confirm that a file has been opened before writing any data to that file. We justified this using the example of attempting to open a file in a directory that doesn't exist. Under these circumstances the intended data would not be written to file, but the code would proceed without informing us of this.

Even if we do confirm that a file we are intending to write to is open there are other problems that may occur. We may successfully write some data to file, and then reach our disk quota set by the system administrator. Subsequent attempts to write to file would then fail, although the code would continue to execute. Alternatively we may be expecting to read 50 double precision numbers from a file that only contains 40 such numbers. After successfully reading 40 numbers we would like to be informed that we had reached the end of the file, and no more numbers were available to read in. We can check that reading from or writing to file has taken place as expected using the C++ function `ios::good`. Use of this function is illustrated below for the case of writing to file; its use when reading from file follows a similar pattern.

```
  std::ofstream write_output("OutputVerified.dat");
  assert(write_output.is_open());
  for (int i=0; i<100; i++)
  {
    write_output << i << "\n";
    assert(write_output.good());
  }
```

3.5 Reading from the Command Line

In scientific computing applications, it is common for a user to want to set some of the parameters used themselves when executing the code. For example, if code has been written to calculate the temperature distribution in a bar using the finite difference method the user may wish to set the thermal conductivity of the bar, or the number of nodes used in the finite difference grid, at the same time that the code is executed. Fortunately, C++ allows the user to do this when running from the command line.

In Sect. 1.2 we promised to explain the third line of the C++ program given in Listing 1.1, namely the line of code shown below.

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

Although we are not quite ready to explain the *whole* meaning of this line until we have introduced pointers in Chap. 4, we may explain how this line allows us to specify input arguments to a program from the command line. Suppose—as described above—we want to write code that allows us to specify an integer number of nodes, `number_of_nodes`, to be used in a finite difference grid, and a double precision floating point variable, `conductivity`, that represents the thermal conductivity of a bar. This is demonstrated by the following code. We will explain the additional header file `cstdlib` used in line 2, and the functions `atoi` and `atof` used in lines 15 and 16 at the end of this section: for the time being we will focus on how to input data from the command line.

```
1  #include <iostream>
2  #include <cstdlib>
3
4  int main(int argc, char *argv[])
5  {
6    std::cout << "Number of command line arguments = "
7              << argc << "\n";
8    for (int i=0; i<argc; i++)
9    {
10     std::cout << "Argument " << i << " = " << argv[i];
11     std::cout << "\n";
12   }
13
14   std::string program_name = argv[0];
15   int number_of_nodes = atoi(argv[1]);
16   double conductivity = atof(argv[2]);
17   std::cout << "Program name = " << program_name << "\n";
18   std::cout << "Number of nodes = " << number_of_nodes;
19   std::cout << "\n";
20   std::cout << "Conductivity = " << conductivity << "\n";
21
22   return 0;
23 }
```

We would instruct the user to specify these parameters by typing the executable name, followed by the number of nodes to be used in the finite difference grid, followed by the value for the conductivity: that is, if we want to use 100 nodes and a conductivity of 5.0 we would compile the code above to produce the executable `CommandLineCode` and then enter the following at the command line:

```
./CommandLineCode 100 5.0
```

This would produce output

```
$./CommandLineCode 100 5.0
Number of command line arguments = 3
Argument 0 = ./CommandLineCode
Argument 1 = 100
Argument 2 = 5.0
Program name = ./CommandLineCode
Number of nodes = 100
Conductivity = 5
$
```

We see from the code and output above that the integer variable `argc` contains the number of arguments specified at the command line. In this case this is three: these are the executable name `./CommandLineCode`, the integer `100`, and the floating point number `5.0`. These are stored as the ordered list `argv[0],argv[1]`, `argv[2]`, as is demonstrated when we use the `for` loop to print these out. Each of these are stored as arrays of characters, and so we must first convert these arrays of characters to the appropriate variable types. This is performed by lines 14, 15 and 16 of the code listed. In line 15, we use the function `atoi(argv[1])` to convert the array of characters stored by `argv[1]` to an integer. Similarly, `atof(argv[2])` converts `argv[2]` to a floating point variable. The functions `atoi` and `atof` require the header file `cstdlib` which has been included in line 2.

3.6 Tips: Controlling Output Format

If the files that are written are to be read only by a computer, then it does not really matter whether these look attractive or not. For example, if a data file is only to be used for importing into a visualisation package then it does not matter if the format of this file is opaque to humans provided the visualisation package can read the file accurately. If, however, humans may want to look at these files then formatting commands, such as controlling the width of each column may be desirable.

Below we show how to implement three commonly desired formatting techniques which we now list before demonstrating.

1. *Output in scientific format.* Scientific format is where a number is written as a product of one number with only one significant figure to the left of the decimal point and an integer power of 10, that is, 465.78 in scientific format is 4.6578×10^2, which may be written in C++ notation as `4.6578e2`. This is achieved by the use of the flag `std::ios::scientific` which requires the header file `fstream`.
2. *Always showing a $+$ or $-$ sign.* The default setting for an output stream is not to print a plus sign before a positive number. To line up numbers in neat columns, we may wish to always precede a number with a plus or minus sign: this is achieved by the use of the flag `std::ios::showpos` which requires the header file `fstream`.
3. *Precision of scientific output.* When scientific format is used the `precision` statement works slightly differently to that described in Sect. 3.2.1: in this case the precision specified is the number of digits *after* the decimal point, and so the number of significant figures is one greater than this number (as there is another significant figure before the decimal point). Furthermore, in contrast to the precision set in Sect. 3.2.1, when scientific format is used zeros are added after the decimal point to ensure that all output is of exactly the same width.

These formatting techniques are demonstrated in the code below.

```cpp
#include <iostream>
#include <fstream>

int main(int argc, char* argv[])
{
  std::ofstream write_file("OutputFormatted.dat");
  // Write numbers as +x.<13digits>e+00 (width 20)
  write_file.setf(std::ios::scientific);
  write_file.setf(std::ios::showpos);
  write_file.precision(13);

  double x = 3.4, y = 0.0000855, z = 984.424;
  write_file << x << " " << y << " " << z << "\n";

  write_file.close();
  return 0;
}
```

3.7 Exercises

3.1 This question assumes that you are starting from the code in the listing below.

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

int main(int argc, char* argv[])
{
  double x[4] = {0.0, 1.0, 1.0, 0.0};
  double y[4] = {0.0, 0.0, 1.0, 1.0};

  return 0;
}
```

1. Extend the code above to print the arrays `x` and `y` to a file called `x_and_y.dat` so that the data file has the four elements of `x` on the top line, and the four elements of `y` on the next line.
2. Extend the code so that the output stream is flushed immediately after each line of the file is written.
3. Extend the code so that the precision is set to 10 significant figures, the output is in scientific notation, and plus signs are shown for positive numbers.
4. Amend the program so that it does not automatically create a fresh file `x_and_y.dat` every time it is run. Have the program first attempt to open the file `x_and_y.dat` as an `ifstream` for reading. If the file can be successfully opened then, after closing the `ifstream`, warn the user. Have the program prompt the user as to whether it should erase the existing file or append to the existing file.

3.2 This question uses the data file `x_and_y.dat` that was written in the previous exercise. The code below assumes that we know that the data file has 4 columns and that we want to count the number of rows.

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

int main(int argc, char* argv[])
{
  std::ifstream read_file("x_and_y.dat");
  if (!read_file.is_open())
  {
    return 1;
  }
  int number_of_rows = 0;
  while(!read_file.eof())
  {
    double dummy1, dummy2, dummy3, dummy4;
    read_file >> dummy1 >> dummy2;
    read_file >> dummy3 >> dummy4;
    number_of_rows++;
  }
  std::cout << "Number of rows = "
            << number_of_rows << "\n";
  read_file.close();
  return 0;
}
```

Run the code above. This code does not give the correct answer. Why is this? Does the code give the correct answer if the final newline character is removed from the file `x_and_y.dat`? Modify the code so that it gives the correct answer.
[*Hint: You might investigate the use of* `read_file.fail()` *which may be used to probe whether the last read on the file stream was unsuccessful.*]

3.3 Write code to implement the implicit (or backward) Euler method to solve the initial value ordinary differential equation

$$\frac{\mathrm{d}y}{\mathrm{d}x} = -y, \qquad y(0) = 1,$$

on the interval $0 \leq x \leq 1$ using a constant step size h. Allow the user to specify the number of grid points, N they want to use at the command line, and use an `assert` statement to ensure that the number of grid points is greater than 1. Use the number of grid points to calculate the step size h. Your code should print a file called `xy.dat` that has two columns: the calculated values of x; and the calculated values of y. Plot the data from the file `xy.dat` and hence compare it with the true solution $y = e^{-x}$.
[*The implicit Euler method* (*see, for example, Süli and Mayers* [3]) *for this problem results in the difference relation*

$$y_0 = 1, \qquad \frac{y_n - y_{n-1}}{h} = -y_n, \qquad n = 1, 2, \ldots, N-1,$$

where h is step size and y_n is the solution at $x_n = nh$, $n = 0, 1, 2, \ldots, N-1$, where N is the number of grid points, and we have used zero-based indexing for the vectors `x` and `y`.]