File Input and Output

Saurav Samantaray

Department of Mathematics

Indian Institute of Technology Madras

October 28, 2024

Redirecting Output

- Being able to transfer data between applications is an essential requirement of most scientific computing softwares.
- We have already introduced basic C++ commands for writing text and the contents stored by a variable to the console.
- In a Linux based system this output may very easily be redirected to a single file rather than the screen.
- Should the executable file be ”test.exe”, this output may be printed to the file ”test.txt” by executing the following

```
$ ./ test.exe > test.txt
```

- There might be some terminal outputs we still may want to be flashed on the screen.
- For example if the program encounters a division by zero, we would want it to get flashed on the screen right at the moment it occurs instead of it getting stored, and then being accessed.

Redirecting Output

- When the output is supposed to be redirected to file, in the way discussed, we can still get console outputs by the using `std::cerr` instead of `std::cout`

```
int x y;
if (y == 0)
{
   std::cerr << "Error -divison by zero" << std::endl;
}
```

- The syntax for `std::cerr` is identical to `std::cout`.
- When the console output is not redirected to file there is no difference between the effect of these two commands.
- Upon output redirection to files only the `std::cout` are saved in the file and `std::cerr` ouputs are still available at the console

Writing to File

- Storing output in a single file might be sufficient for some applications.
- But, for example if we are writing a finite difference code to calculate the solution of a given differential equation, we may want to store
  - the nodes of the mesh in one file,
  - the solution in another file, etc
- Therefore it is necessary to be able to be write output to more than one file.
- C++ provides an extremely large number of commands for printing to file.
- That being said almost all the 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`.

Writing to File

- See ”out.cpp”
- First we declare an *output stream variable* `write_output`,
- by specifying it of type `ofstream`
- along with this the file name, `output.dat` is also specified.
- Next we check if the file is open via `assert`.
- Writing to file is quite similar to console output,
- we replace `std::cout` with `write_output`, which writes the entries to the file (`output.dat`) associated with the output stream variable.
- Finally, when all required data has been written to file, we ”close the file handle”
- Like console outputs, outputs to files are also buffered i.e. the output may not be immediately written to the file.
- 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.

Writing to File

- It is quite important that a file is closed as soon as all the relevant data to be stored in it are available, as
- 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.
- It is also possible to flush a buffer without closing the file handle.
- This is done in a similar way as it is done for console output, as shown bellow for the output stream variable `write_output`

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

Writing to File

- It was mandated to check if a file is open, before attempting to write data to it.
- Why is this important?
- There may be scenarios when the file cannot be opened, perhaps we did not have permission to write to that file, or a directory we have specified doesn’t exist;
- then writing to the `ofstream` may cause no error even though writing to the file is not possible.
- For example if we renamed the location of the output file to a folder we are restricted from writing to

```
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.

Writing to File

- The executable created from ”out.cpp” 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.
- Whether or not the file ”output.dat” existed before once the code is executed, there will be a file called ”output.dat” that is listed below.

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

- Suppose that, rather than deleting the file if it exists, we want our code to append data to the end of this file.
- this could be achieved by modifying
  `ofstream write_output("output.dat");` to
  `ofstream write_output("Output.dat", ios::app);`

Writing to File

- If the file ”output.dat” did not exist and we were to execute the code, with the modified lines
- it would then create the file ”output.dat” shown as before
- If we were then to execute the code a second time we would then end up with the file output.dat being modified as:

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

Setting the Precision of the Output

- The key formatting command for scientific computing applications is specification of the precision of the output.
- See ”set_pre.cpp”.
- The number in brackets after the `precision` commands specifies the number of significant figures that the output is correct to.
- When the precision is set to 10 significant figures, but only eight significant figures will be printed in ”set_pre.cpp”:
- this is because the variable x is only given to eight significant figures, and so the remaining accuracy requested is redundant.

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.
- 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 (`std::cin`).
- `std::cin` replaced by the input stream variable.
- Suppose we want to input the file ”output.dat” created before
- We know that this file has three rows and two columns, and so we may read this file using the code
- See ”inp.cpp”
- The assertion 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.

Reading from File

- In the previous code, we knew that the file we were reading has three rows and two columns, and so we knew when writing this code that the statements inside the `for` loop had to be executed three 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 end of the file 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. (See ”mod_inp.cpp”)