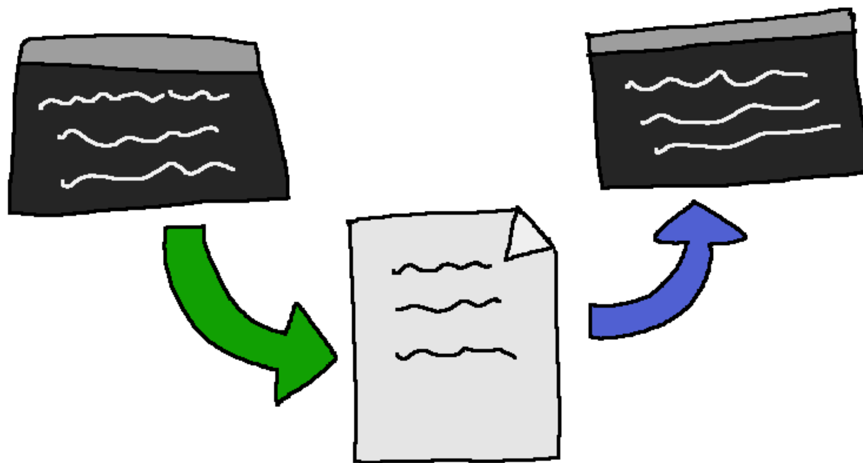
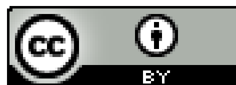
Core Computer Science Notes: File I/O



An overview compiled by Rachel Singh

This work is licensed under a
Creative Commons Attribution 4.0 International License.



Last updated June 16, 2020

Contents

1 Output streams **2**

2 Input streams **4**

2.1 Reading an entire file 5

3 Parsing files **7**

Topic 1

Output streams

In C++ we've been using `cout` to stream information to the console window in our programs. Using `cout` requires including the `iostream` library.

```
1 cout << "Hello, " << location << "!" << endl;
```

Writing out to a text file works in a very similar way. We will need to include the `fstream` library in order to get access to the `ofstream` (output-file-stream) object. Streaming out to a file works in the same way as with `cout`, except that we need to declare a `ofstream` variable and use it to open a text file.

```
1  #include <iostream>      // Console streams
2  #include <fstream>       // File streams
3  using namespace std;
4
5  int main()
6  {
7      // Console output
8      cout << "Hello, world!" << endl;
9
10     // File output
11     ofstream outputFile( "file.txt" );
12     outputFile << "Hello, world!" << endl;
13     outputFile.close();
14 }
```

You can use the output stream operator `<<` to continue chaining together different items - `endl`s, string literals in double quotes, variable values, etc. just like with your `cout` statements.

Once the file is closed, you will see the file on your computer, usually the same directory as your **.cpp** files.

Different file types...

Any file type that is a plaintext file can be built as well - .html files, .csv files, heck, even .cpp files. However, *generally* if you wanted to write a program to output a different file type, you'd use a library to properly convert the data.

Outputting html data:

```
1  #include <fstream>
2  using namespace std;
3
4  int main()
5  {
6      ofstream outputFile( "page.html" );
7      outputFile << "<body>" << endl;
8      outputFile << "<h1>This is a webpage</h1>" << endl;
9      outputFile << "<p>Hello, world!</p>" << endl;
10     outputFile << "</body>" << endl;
11     outputFile.close();
12 }
```

Outputting csv data:

```
1  #include <fstream>
2  using namespace std;
3
4  int main()
5  {
6      ofstream outputFile( "spreadsheet.csv" );
7      outputFile << "COLUMN1,COLUMN2,COLUMN3" << endl;
8      outputFile << "cell1,cell2,cell3" << endl; // row 1
9      outputFile << "cell1,cell2,cell3" << endl; // row 2
10     outputFile << "cell1,cell2,cell3" << endl; // row 3
11     outputFile.close();
12 }
```

Topic 2

Input streams

File input streams work just line console input streams. You will need to create a `ifstream` (input-file-stream) object and open a file, and then you can read in the contents of that file. Files to be read should generally be placed in the same path as your .cpp file, though the working directory on your system may vary.

```
#include <fstream>      // File streams
#include <string>       // Strings
using namespace std;

int main()
{
    string data1, data2;

    // File input
    ifstream inputFile( "file.txt" );
    inputFile >> data1;                 // Read one word
    inputFile.ignore();                 // Clear buffer
    getline( inputFile, data2 );        // Read one line
    inputFile.close();
}
```

Just like with using `cin`, you can use the input stream operator (`>>`) and the `getline()` function with the file streams to get text. You will also need one or more variable to store the text read in.

2.1 Reading an entire file

Reading chunks of data:
Let's say you have a file full of data to read in. For this example, the file will be a list of numbers that we want to add together. The file might look something like....

Data.txt:

```
9 15 16 0 10 13 5 16 1 9 2 17 3 3 8
```

In our program, we want to read all the numbers, but to do this, we need to use a **loop** and read in **one number at a time**. We can keep a running total variable to keep adding data to the sum as we go. We can use a loop to continue reading while it is successful, using this as the condition: `input >> readNumber`. This will stop the loop once there's nothing else to read, and it updates the `readNumber` variable with the input each cycle.

```
1  ifstream input( "data.txt" );
2  
3  int sum = 0;     // Sum variable
4  int readNumber; // Buffer to store what we read in
5  
6  // Keep reading while it's possible
7  while ( input >> readNumber )
8  {
9      sum += readNumber;       // Add on to the sum
10 
11     // Output what we did
12     cout << "Read number " << readNumber
13         << ",\t sum is now " << sum << endl;
14 }
15 
16 cout << "FINAL SUM: " << sum << endl;
```

The output of this program would look like this:

```
Read number 9,   sum is now 9
Read number 15,  sum is now 24
(... etc ...)
Read number 3,   sum is now 119
Read number 8,   sum is now 127
FINAL SUM: 127
```

Reading lines of data:
In other cases, maybe you're reading in text data from a file and want to read in a full line at a time. We can use a while loop with the `getline()` function as well to make sure we read each line of a text file:

```
ifstream input( "story.txt" );

string line;

while ( getline( input, line ) )
{
    cout << line << endl;
}
```

The output of this program would look like this:

```
CHAPTER I.
Down the Rabbit-Hole


Alice was beginning to get very tired of sitting by her
sister on the bank, and of having nothing to do: once or
twice she had peeped into the book her sister was
reading, but it had no pictures or conversations in it,
"and what is the use of a book," thought Alice "without
pictures or conversations?"
```

Topic 3

Parsing files

Reading in data from a file is one thing, but making sense of what was read in is another. Are you storing saved data from the last session of the program? Are you trying to parse data to crunch? How do you read that data in logically?

First, if your program is going to be *saving output* that will need to be read in and made sense of later on, how do you organize the data that's output? It's up to you to make sure to structure the output save data in a consistent and readable way...

```
SAVEGAME    RachelsGame
LEVEL       5
GOLD        1000
LOCATION    RegnierCenter
```

If every save file from the program is formatted in the same way, then when reading the file we can make assumptions...

- First word: "SAVEGAME" - not important (human readable)
- Second word: Save game name - store in `gameFileName`
- Third word: "LEVEL" - not important (human readable)
- Fourth word: Player's level - store in `level`

...And so on. This could work, but we could also take advantage of those human-readable labels that were added into the file.

- Read `firstWord`.
- If `firstWord` is "SAVEGAME", then read the next word into `gameFileName`.
- Else if `firstWord` is "LEVEL", then read the next word into `level`.

So, let's say we have our four variables for a save game:

Name	Data type
gameFileName	string
level	int
gold	int
location	string

When we go to save our game file, it would be a simple output like this:

```
// Save the game
ofstream output( "save.txt" );
output << "SAVEGAME " << gameFileName << endl;
output << "LEVEL    " << level << endl;
output << "GOLD     " << gold << endl;
output << "LOCATION " << location << endl;
```

Giving us a save file like:

```
SAVEGAME MyGame
LEVEL    1
GOLD     10
LOCATION OCB
```

Then to read it, we could approach it in a couple of different ways. If we assume that the file will always have exactly four lines of data saved and will always be in the same order, we could read it like:

```
// Load the game
string buffer;
ifstream input( "save.txt" );
input >> buffer; // ignore SAVEGAME
input >> gameFileName;

input >> buffer; // ignore LEVEL
input >> level;

input >> buffer; // ignore GOLD
input >> gold;

input >> buffer; // ignore LOCATION
input >> location;
```

Or, if we weren't sure that order these would show up in, we could store the first item read into `buffer` each time, and based on what that label was ("SAVEGAME", "LEVEL", etc.) we would know what to read *next*...

```
string buffer;
ifstream input( "save.txt" );

while ( input >> buffer )
{
    if ( buffer == "SAVEGAME" )
        input >> gameFileName;

    else if ( buffer == "LEVEL" )
        input >> level;

    else if ( buffer == "GOLD" )
        input >> gold;

    else if ( buffer == "LOCATION" )
        input >> location;
}
```

If we stepped through this, `buffer` is always going to store one of the data labels, because after buffer is read, we immediately read the second item in the line of text. Once the second item is read, the next thing to be read will be the next label.