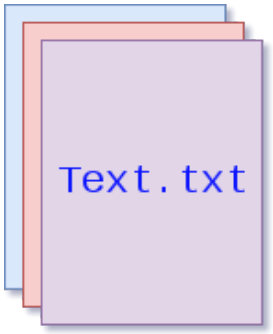
Input Files



Files are able to be accessed via a program. Text and data files are what you will encounter in CSI-CSIII. In order to read from an input file, the file has to previously exist. The preprocessor directive **#include <fstream>** is required.

Opening a file can be done two ways. The type is **ifstream**. **ifstream variable("fileName.extension");** or it can be done on two lines using the member function **open()**:

```
ifstream iFile;
iFile.open("fileName.extension");
```

You are able to put the name of the file into a string variable. You may (before C++11) need to use the member function **c_str()** such as

```
string filename = "fileName.ext";
iFile.open(filename.c_str());
```

It is good practice to immediately check whether or not the file was successfully opened:

```
if(iFile.fail()) {
cout << "File failed to open\n";
exit(1); }
```

From here, your variable can be used in place of cin to read from the file. For example, if the file contains a bunch of integers, you can:

```
int x;
iFile >> x;
```

Variable **x** now contains the next integer from the file. To retrieve more than one piece of data at a time, you can **iFile >> x >> y;**

Other types are able to be read from files, such as **string** and **double**.

Files are read from top to bottom, left to right. You cannot traverse backward in a file, only forward. You can, however, take a peek at the next value in the file before actually reading it into a variable with the member function **peek()** such as **cout << iFile.peek();** This will display in integer format (base 10) and corresponds to the ASCII table. For example, if the next character in the file is a 5, **iFile.peek()** will print out 53. If the next character is a space, this member function will print out 32. If you would like to ignore the next character in the file, you can invoke **iFile.ignore();** This includes ignoring whitespace.

Remember, there is undefined behavior that can come about while reading from a file. If you try to put a character into an integer, this will not work and result in a garbage value in the integer

variable. **iFile.peek()** will not properly work as well if this happens. However, you can put integers into a char as they correspond to the ASCII table. When finished with the file, invoke the **close()** member function such as **iFile.close();** This ensures the file closed properly.

Usually when reading from a file, you read the contents within a loop. The condition of the loop usually depends upon whether or not there are more pieces of data left to read from the file. There are a few ways of doing this, and one of these ways seems logical, but may not always have the intended consequences.

The member function **eof()** returns a **bool** true if the end-of-file marker has been reached by the last input operation and false if there is still more data to be read from the file. However, the end-of-file marker is not visible in the file, but it exists right after the last piece of data. Therefore, this example will produce an incorrect result:

```
fstream iFile("file.dat");
int x;
while(!iFile.eof()) { //Loop while file is not at the end of file
    iFile >> x;
    cout << x << ' ';
}
```

Let's say that **file.dat** contains the integers **1 2 3** all on one line separated by a space. The first iteration of the loop extracts the **1** and outputs it. The second iteration of the loop extracts the **2** and output it. The third iteration of the loop extracts the **3** and outputs it. Now, here is where you may think the **while** loop terminates as there is no physical data left to be read. However, there is that invisible end-of-file marker and the **eof()** function will not return false until after it has read this marker. Therefore the loop iterates again. Interestingly, it does not extract the marker into **x**. The contents of **x** will be the last piece of data read. Therefore, **3** will be output again giving the final output **1 2 3 3**.

Try to avoid this condition. A better alternative is to do as such:

```
while(iFile >> x)
```

This will execute as long as there is data you can physically see left to be read. There will be no double output as the above method.

Another method would be to use the **getline** function (extract whole line) as seen in **InputString.cpp** example program in the **String Class** section. You can use this function with a **char** array as well. Example:

```
char extract[100]; //This will extract up to 100 chars including \0
while(iFile.getline(extract, 100, '\n')) //or when it reaches \n
```