Recitation 8: File IO

This recitation is due **Saturday, March 10th, by 6 pm**

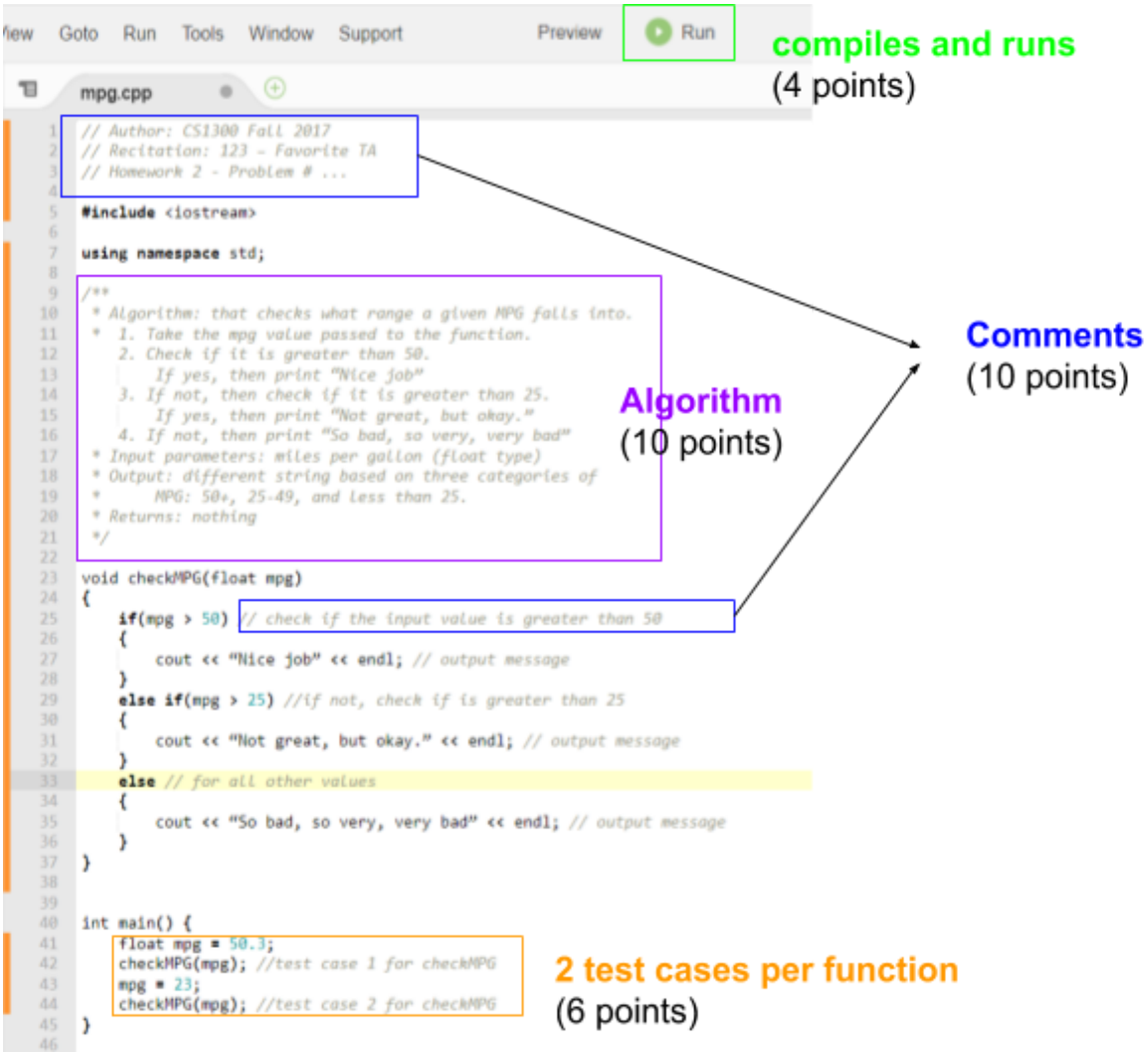
- All components (Cloud9 workspace, moodle quiz attempts, and zip file) must be completed and submitted by **Saturday, March 10th 6:00 pm** for your solution to receive points.
- Recitation attendance is required to receive credit.
- **Develop in Cloud9:** For this recitation assignment, write and test your solution using Cloud9.
- **Submission:** All three steps must be fully completed by the submission deadline for your homework to receive points. Partial submissions will not be graded.
    1. Make sure your Cloud9 workspace is shared with your TA: Your recitation TA will review your code by going to your Cloud9 workspace. TAs will check the last version that was saved before the submission deadline.
        - Create a directory called *Rec8* and place all your file(s) for this assignment in this directory.
    2. **Submit to the Moodle Autograder:** Head over to Moodle to the link *Rec 8*. You will find one programming quiz question for each problem in the assignment. You can modify your code and re-submit (press Check again) as many times as you need to, up until the assignment due date.
    3. **Submit a zip file to Moodle:** After you have completed all the questions and checked them on Moodle, you must submit a zip file with the .cpp file(s) you wrote in Cloud9. Submit this file going to *Rec 8 (File Submission)* on moodle.

Please follow the same submission guidelines first outlined in Homework 3. Here is a summary:

- 4 points for code compiling and running
- 6 points for two test cases for each function in main
    - Make sure to zip up any test files you create with your .cpp.
- 10 points for an algorithm description for each function
- 10 points for comments
    - required comments at top of file

```
// CS1300 Spring 2018
// Author:
// Recitation: 123 – Favorite TA
// Cloud9 Workspace Editor Link: https://ide.c9.io/…
// Recitation 8 - Problem # ...
```

    - comments for functions / test cases



File IO

During this class so far, we have been using the `iostream` standard library. This library provided us with methods like `cout` and `cin`. `cin` is a method is for *reading from* standard input (i.e. in the terminal via a keyboard) and `cout` is for *writing to* standard output.

In this recitation we will discuss file input and output, which will allow you to *read* and *write* from a file. In order to use these methods, we will need to use another C++ standard library, `fstream` (file stream).

These headers must be included in your C++ file before you are able to process files.

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

Opening a file:

The C++ input/output library is based on the concept of **streams**. An *input stream* is a source of data, and an *output stream* is a destination for data.

Step 1: create an object (a variable) of file stream type. There are three types of stream objects:

1. If you want to open a file for reading only, then the `ifstream` object should be used.

2. If you want to open a file for writing only, then the `ofstream` object should be used.

3. If you want to open a file for reading and writing, then the `fstream` object should be used.

stream	read	write
`ifstream`	Y	N
`ofstream`	N	Y
`fstream`	N	N

For example:

```
ofstream myfile; //create an output file stream for writing to file
```

Step 2: once you have an object (a variable) of file stream type, you need to open the file. You cannot read from or write to a file before opening it. To open the file, you use the method (function) named **open**. For ifstream and ofstream objects, the method takes only one parameter: the file name as a string (surrounded by " " if fiving file name directly).

For example:

```
ofstream myfile; //create an output file stream
myfile.open("file1.txt");  //open the file file1.txt with the file stream
```

The following section is from: http://www.cplusplus.com/doc/tutorial/files/, a tutorial on input/output with files from cplusplus.com. You are encouraged to go read the rest of the document.

In order to open a file with a stream object we use its member function open:

```
open (filename, mode);
```

Where filename is a string representing the name of the file to be opened, and mode is an optional parameter with a combination of the following flags:

ios::in	Open for input operations.
ios::out	Open for output operations.
ios::binary	Open in binary mode.
ios::ate	Set the initial position at the end of the file. If this flag is not set, the initial position is the beginning of the file.
ios::app	All output operations are performed at the end of the file, appending the content to the current content of the file.
ios::trunc	If the file is opened for output operations and it already existed, its previous content is deleted and replaced by the new one.

All these flags can be combined using the bitwise operator OR (|). For example, if we want to open the file example.binin binary mode to add data we could do it by the following call to member function open:

```
ofstream myfile;
myfile.open ("example.bin", ios::out | ios::app | ios::binary);
```

Each of the open member functions of classes ofstream, ifstream and fstream has a default mode that is used if the file is opened without a second argument:

class	**default mode parameter**
ofstream	ios::out
ifstream	ios::in
fstream	ios::in | ios::out

Checking for open file:

It is always good practice to check if the file has been opened properly and send a message if it did not open properly. To check if a file stream successfully opened the file, you can use `fileStreamObject.is_open()`. This method will return a boolean value `true` if the file has successfully opened and `false` otherwise.

```
ifstream myfilestream;
myfilestream.open("myfile.txt");

if (myfilestream.is_open())
{
    // do things with the file
}
else
{
    cout << "file open failed" << endl;
}
```

You can also use the `fileObject.fail()` function to check if the file open was successful or not. This will return `true` if the file open has failed and `false` otherwise.

```
ifstream myfilestream;
myfilestream.open("myfile.txt");

if (myfilestream.fail())
{
    cout << "file open failed" << endl;
}
else
{
    //do things with the file
}
```

Reading from a file using stream insertion (>>)

When you read from a file into your C++ program, you will use the stream insertion syntax you have seen with cin, which inputs information from the keyboard or user, `>>`. The difference is that you will be using the `ifstream` (input file stream) or `fstream` (file stream) object instead of the `cin` object, allowing for the program to read *input from the file* instead of input from the terminal.

You can also use the `>>` operator. `filestream >> line` means to take characters from the filestream up until the next delimiter (a space character or a newline character) and place them in the `line` string variable.

readfile2.cpp

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

using namespace std;

int main()
{
    ifstream myfilestream;
    myfilestream.open("myfile.txt");

    if (myfilestream.is_open())
    {
         string line = "";
         myfilestream >> line;       //Reads from file into variable line
         cout << line << endl;
    }
    else
    {
        cout << "file open failed" << endl;
    }
}
```

output

```
Hello
```

Reading from a file using getline:

If you want to read an *entire line* up to the newline character \n, you should use getline instead of the stream insertion syntax. Say we have a variable `line` of type string. Then, `myfilestream >> line` will read in from `myfilestream` up to the next space character (basically, the next word in the file) and set `line` equal to its contents.

`getline(myfilestream, line)` will read in an ***entire line*** from `myfilestream` and set `line` equal to its contents.

In the following example, you would be given a file with information to read. Then once the print statement is called, you will see a single line of the file printed to the terminal.

For example:

myfile.txt

```
Hello world! This is the first line.
This is the second line.
```

readfile.cpp

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

using namespace std;

int main()
{
    ifstream myfilestream;
    myfilestream.open("myfile.txt");

    if (myfilestream.is_open())
    {
        string line = "";
        getline(myfilestream, line)  //Reads the entire line into variable line
        cout << line << endl;
    }
    else
    {
```

```
        cout << "file open failed" << endl;
    }
}
```

output

```
 Hello world! This is the first line.
```

Writing to a file using stream insertion (<<):

When you write to a file from your C++ program, you will use the stream insertion syntax you have seen with `cout`, `<<`. The difference is that you will be using the `ofstream` or `fstream` object instead of the `cout` object, allowing for the program to direct the output correctly to a file, instead of to the terminal screen (as for `cout`).

For example:

readfile3.cpp

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

using namespace std;

int main()
{
    ofstream myfilestream;
    myfilestream.open("myfile.txt");

    if (myfilestream.is_open())
    {
        myfile << “Writing this line to a file. \\n”;
        myfile << “Writing this second line to a file. \\n”;
    }
    else
    {
        cout << "file open failed" << endl;
    }
}
```

If `myfile.txt` does not yet exist, it will be created by default. Note that if it *does* exist,

the existing contents are overwritten. To append (add on to the end of a file) instead of overwriting, you can do `myfilestream.open("myfile.txt", ios::app)`.

output:

```
Writing this line to a file.
Writing this second line to a file.
```

Reading in all the lines

Many times you would like to read in all the lines from a file. There are many ways to do this, but one way to do so is shown below.

It takes advantage of the fact that the function `getline(filestream, mystr)` returns `true` as long as an additional line has been successfully assigned to the variable `mystr`. Once no more lines can be read in, `getline` returns false.

So we can set up a while loop where the condition is the call to `getline`.

myfile.txt

```
Hello world! This is the first line.
This is the second line.
Here's a third line.
And a fourth.
```

readfile4.cpp

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

using namespace std;

int main()
{
    ifstream myfilestream;
    myfilestream.open("myfile.txt");

    if (myfilestream.is_open())
    {
        string line = "";
        int lineidx = 0;
        while (getline(myfilestream, line))
```

```
        {
            cout << lineidx << ": " << line << endl;
            lineidx++;
        }
    }
    else
    {
        cout << "file open failed" << endl;
    }

}
```

output

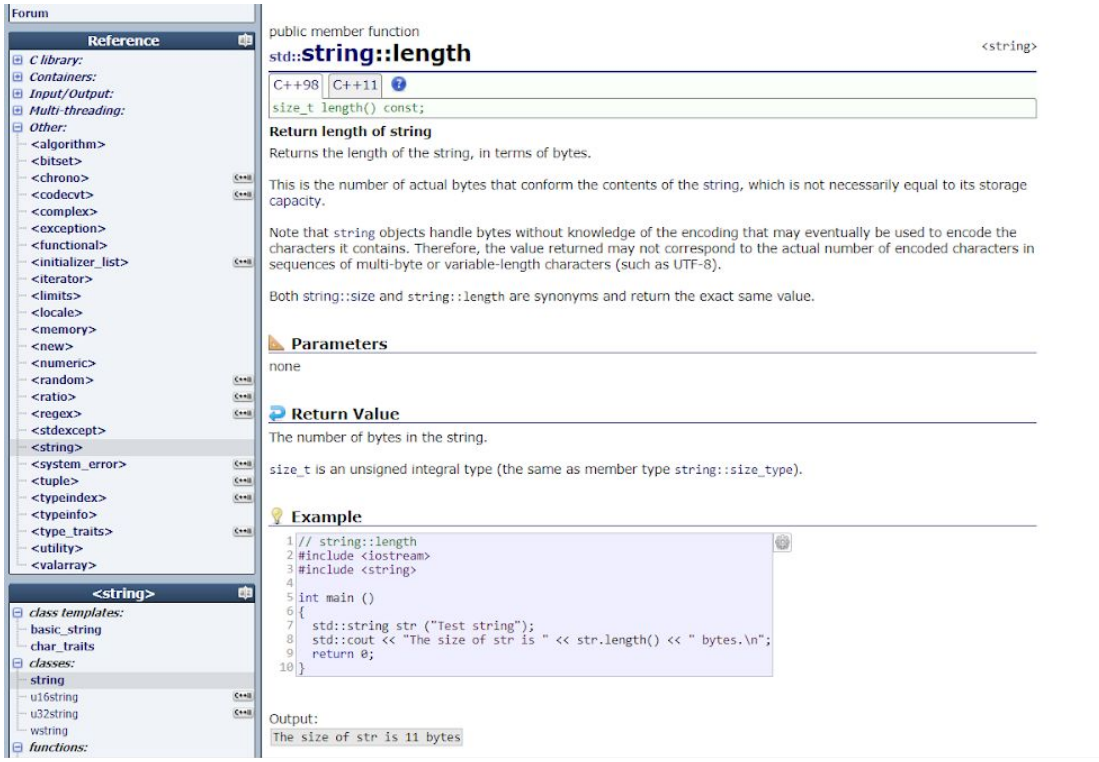
```
0: Hello world! This is the first line.
1: This is the second line.
2: Here's a third line.
3: And a fourth.
```

Closing a file:

When you are finished processing your files, it is recommended to close all the opened files before the program is terminated. The standard syntax for closing your file is `myfilestream.close();`

Other functions

Here are some useful functions for this weeks assignment and recitation. We've provided links to documentation on how to use them. These pages provide explanation on the function as well as examples on how to use them:



- string.length() (find the length of a string)
- string.empty() (determine if a string is empty or not)
- getline (extract characters from a stream and place into a string)
- fstream::open (open a file)
- fstream::is_open (determine if a file successfully opened)
- ios::fail (determine if a file successfully opened)
- ios::eof (determine if at the **e**nd **o**f a **f**ile)

Problem Set

Question 1

Write a function **checkFile** that opens up a file. The function takes one parameter, the name of the file *filename*. The function should return true if the file was opened successfully otherwise it should return false.

For example, if a file called *myFile.txt* exists in your workspace, then `checkFile("myFile.txt")` would return true because the file exists in the system.

Question 2

Write a function **fileLoadWrite** that opens a file and writes to it the squares of the numbers from 1 to 10, each on its own line. The function takes one parameter, the name of the file *filename.* It should open the file in write mode and write 10 lines to the file, each containing the square of the number line it is on (e.g. line 1 will contain 1, line 2 will contain 4, etc.).

The function should close the file and return 0 if the file was opened successfully, else it should return -1.

An example of a function call to **fileLoaderWrite** is fileLoadWrite(“myFile.txt”) which should successfully open a new file called *myFile.txt* and write ten lines of squared numbers.

Note: If a file called *myFile.txt* is already associated **to a stream** then the open(…) method fails.

Question 3

Write a function **fileLoadRead** that reads a file. The function takes a parameter, the name of the file *filename,* and returns the number of lines in the file. It should open a file in read mode, check for success, and read the lines until the end of the file. Remember to close the file.

If the file was not opened successfully, return -1.

If the file exists but is empty, return 0.

If the file exists and is not empty, return the number of lines. Include all the lines in your count, even those that are blank.

For Example:

Given a file called *myFile.txt* with the following content:

```
hola
ciao


hello
hallo
```

the function call `fileLoadRead("myFile.txt")` would return 6.

Question 4

Write a function **getLinesFromFile** that reads from a file and stores its content in an array. The function takes three parameters: a string *fileName,* a string array *wordArray,* an integer *sizeArray*.

getLinesFromFile should open the file in read mode, read each line, store each line in the array. The function should return the number of lines placed into the array.

If any empty lines are encountered, those should not count towards the count of the number of lines and should not be added to the array.

If the file does not exist, return -1.

Example: if *fileName.txt* has the following contents:

```
sky
night

78


ski season
```

The function call

```
getLinesFromFile("fileName.txt", wordArray, 4)
```

would return 4 and wordArray would look like

```
["sky", "night", "78", "ski season"]
```

Question 5

After a program finishes executing, intermediate results are lost unless we save them somewhere. Frequently, this can be accomplished by writing the results to a file. For this problem, you must write a function ***saveData*** which takes four arguments: the filename, an array of strings, an integer n which represents the number of values in the array, and the size of the array. The given array of strings has the following structure:

The first *n* entries of the array will be values, and the *n+1* entry will be a name.

```
["value1","value2","value3","name"]
```

This function will convert the first *n* values of the array into double and compute their average.

Your function should then write to the file so that the result is in the following format:

```
Name: name at n+1 position
Avg: average of n numbers
```

If `n=0`, then print just the first line to the file and do not include the line with Avg:

```
Name: first value from array
```

If the input is not in valid format the function should not write to the file.

```
void saveData(string fileName, string data[], int n, int size) {
  //Your code here
}
```

Example:

```
string data[4] = {"2.3", "-1.5", "0.8", "Garth"};


saveData("my_data.txt", data, 3, 4);
```

The file “my_data.txt” should have the following two lines:

```
Name: Garth
Avg: 0.5333
```