CSCI 1300 - Starting Computing
Instructor: Fleming
Recitation 7

This assignment is due **Friday, October 19th, by 11:55 pm**

- ***All components*** **(Cloud9 workspace and moodle quiz attempts) must be completed and submitted by Friday, October 19th, by 11:55 pm for your solution to receive points.**
- **Recitation attendance is required to receive credit.**

Please follow the same submission guidelines outlined in Homework 3 description regarding Style, Comments and Test Cases. Here's a review below on what you need to submit for Recitation 7.

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 Cloud 9 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 **Rec7** and place all your file(s) for this assignment in this directory.
   - Make sure to *save* the final version of your code (File > Save). Verify that this version displays correctly by going to File > File Version History.
   - The file(s) should have all of your functions, test cases for the functions in main function(s), and adhere to the style guide. Please read the **Test Cases** and **Style and Comments** sections included in the **Homework 3** write up for more details.
2. ***Submit to the Moodle Autograder:*** Head over to Moodle to the link **Rec 7**. You will find one programming quiz question for each problem in the assignment. Submit your solution for the first problem and press the Check button. You will see a report on how your solution passed the tests, and the resulting score for the first problem. You can modify your code and re-submit (press *Check* again) as many times as you need to, up until the assignment due date. Continue with the rest of the problems.

CSCI 1300 - Starting Computing
Instructor: Fleming
Recitation 7

Two-Dimensional Arrays

Two-dimensional arrays are also arrays, but they are arrays of arrays. When you define a one-dimensional array of integers, it will look like this:

```
int integer_array[10];
```

For an overview of one-dimensional arrays and pass by value vs pass by reference refer back to last week's recitation writeup. To make a two-dimensional array, simply add a new dimension as follows,

```
int integer_array[10][20];
```

which will create a 10 * 20 matrix. The (i, j) element of this matrix is accessed by

```
integer_array[i][j];
```

Initializing values for two-dimensional arrays is basically the same as in the one-dimension case. For example, if you want to initialize a two-dimensional array with some values, here's what you might want to do:

```
int integer_array[2][2] = { {10, 20}, {30, 40} };
```

However, be careful when you write something like this,

```
int integer_array[2][2] = { {10, 20} };
```

which does not give the matrix as

```
10 20
10 20
```

Instead, it gives you

```
10 20
0 0
```

When we pass a one-dimensional array as an argument to a function we also provide its length. We do the same thing for two-dimensional arrays. In addition to providing the length (or number of rows) of a two-dimensional array, we will also assume that we know the length of each of the subarrays (or the number of columns). A function taking a two-dimensional array with 10 columns as an argument then might look something like this:

```
void twoDimensionalFunction(int matrix[][10], int rows){ … }
```

CSCI 1300 - Starting Computing
Instructor: Fleming
Recitation 7

File Input/Output

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 from and write to 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`	Y	Y

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 giving 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())
    {
        myfilestream << "Writing this line to a file". << endl;
        myfilestream << "Writing this second line to a file". <<endl;
    }
    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();`

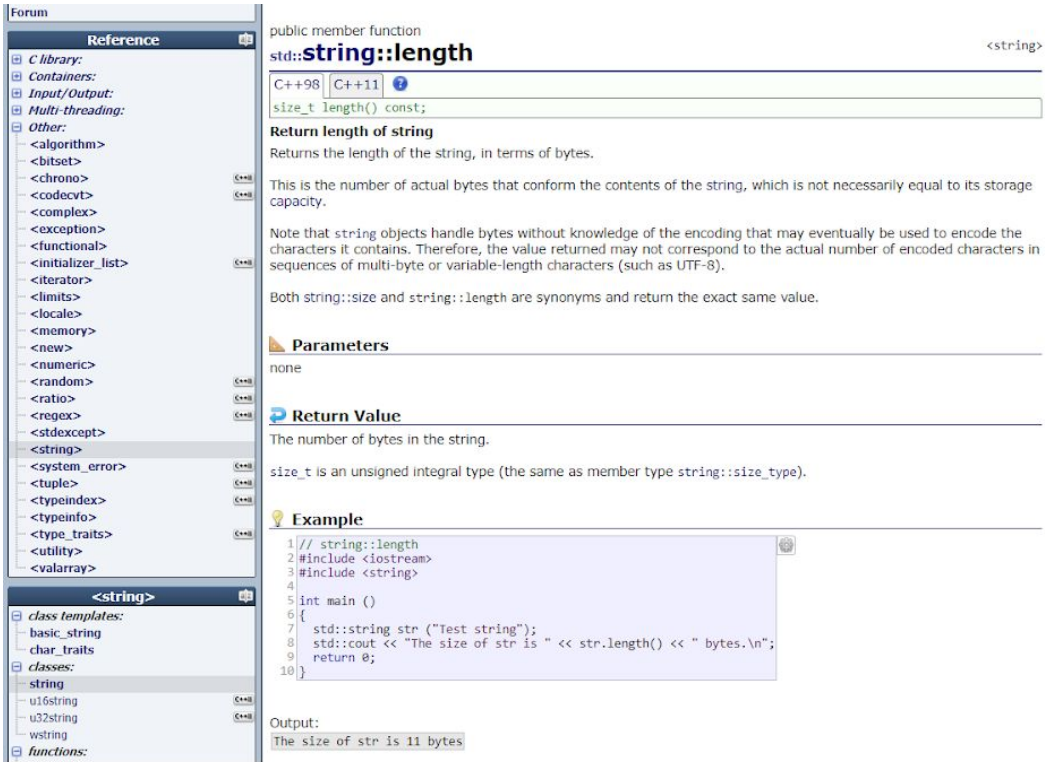
Other functions

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)

CSCI 1300 - Starting Computing
Instructor: Fleming
Recitation 7

- ios::eof (determine if at the end of a file)



CSCI 1300 - Starting Computing
Instructor: Fleming
Recitation 7

Problem Set

Problem 1

Write a function **replaceNums** that replaces each element except the first and last by the larger of its two neighbors.

- Your function should be named
- Your function should take two parameters
    - An **integer array**
    - The **length** of the given array
- Your function should not return or print anything

Examples

```
int myArray[2] = {1, 5};              →    {1, 5}

int myArray[1] = {1};                 →    {1}

int myArray[3] = {1, 3, 5};           →    {1, 5, 5}

int myArray[3] = {5, 3, 1};           →    {5, 5, 1}

int myArray[5] = {1, 2, 3, 4, 5};     →    {1, 3, 4, 5, 5}

int myArray[5] = {5, 4, 3, 2, 1};     →    {5, 5, 4, 3, 1}
```

Problem 2

Write a function to print all the integers in a two dimensional array. Each row of the array should be printed on a separate line with the integers separated by commas. Assume that the array has 5 columns.

- Your function should be named **printTwoDArray**
- Your function should take two input arguments
    - The two dimensional **integer array** to be printed
    - The **length** of the given array
- Your function should not return anything
- Your function should print the values in the array

Edge Cases:
If the length is 0 your function should print 0.
If the length is a negative number your function should print -1.

Example:

```
int myArray[3][5] = {{1,2,3,4,5},{6,7,8,9,10},{11,12,13,14,15}};
printTwoDArray(myArray, 3);
```

Output:

```
1,2,3,4,5
6,7,8,9,10
11,12,13,14,15
```

Problem 3

Write a function **floodMap** which takes three arguments, a two-dimensional array of doubles indicating the height of the terrain at a particular point in space (assume that there are 4 columns), an int indicating the number of rows in the map, and a double indicating the current water level. The function should print out a "map" of which points in the array are below the water level. In particular, each position at or below the water level will be represented with a * and each position above the water level will be represented with a _.

- Your function should be named **floodMap**
- Your function should take three parameters.
    - A two-dimensional **array of doubles** with 4 columns.
    - The **number of rows** in the array.
    - A **double** indicating the current water level.
- Your function should print a flood map as described.

Examples

```
double map[1][4] = {{5.0, 7.6, 3.1, 292}};  *_*_
                    floodMap(map, 1, 6.0);

double map[2][4] = {{0.2, 0.8, 0.8, 0.2},   ____
                    {0.2, 0.2, 0.8, 0.5}};  ____
                    floodMap(map, 2, 0.0);

double map[2][4] = {{0.2, 0.8, 0.8, 0.2},   _**_
                    {0.2, 0.2, 0.8, 0.5}};  __**
                    floodMap(map, 2, 0.5);

double map[2][4] = {{0.2, 0.8, 0.8, 0.2},   ****
                    {0.2, 0.2, 0.8, 0.5}};  ****
                    floodMap(map, 2, 1.0);
```

```
double map[4][4] = {{1.0, 5.0, 1.0, 1.0},   *_**
                    {1.0, 5.0, 5.0, 1.0},   *__*
                    {5.0, 1.0, 5.0, 5.0},   _*__
                    {1.0, 1.0, 1.0, 1.0}};  ****
                   floodMap(map, 4, 3.14);
```

Problem 4

Write a function **getLinesFromFile** that reads from a file of integers and stores its content in an array. Each line in the file will either contain a single integer or be empty.

- Your function should be named **getLinesFromFile**
- Your function should take three parameters
  - A **string** filename
  - An **array of integers**
  - The **length** of the given array
- Your function should fill the array with the integers in the file
- Once the array is full (once **length** integers have been added) subsequent integers in the file should be ignored
- Your function should return the number of integers added to the array if the file exists
- If the file does not exist, return -1

Example:

if `fileName.txt` has the following contents:

```
1
5
23

18
```

The function call

```
int arr[4];
getLinesFromFile("fileName.txt", arr, 4);
```

would return 4 and arr would look like

```
[1, 5, 23, 18]
```