Recitation 7

This recitation is due **Saturday, March 3rd, by 6 pm**

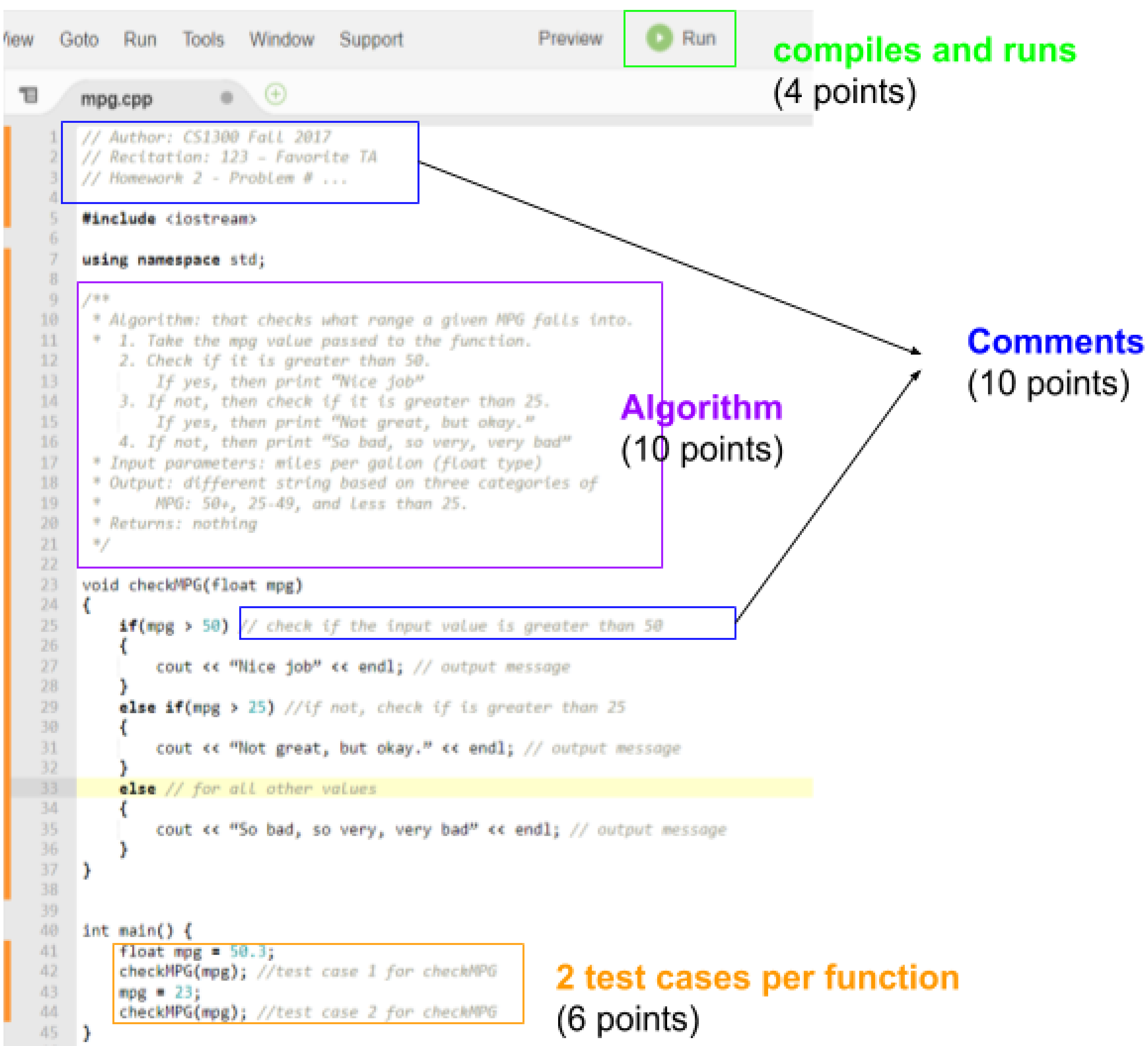
- All components (Cloud9 workspace, moodle quiz attempts, and zip file) must be completed and submitted by **Saturday, March 3rd 6:00 pm** for your solution to receive points.
- Recitation attendance is required to receive credit.
- **Develop in Cloud9:** For this recitation assignment, write and test 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 *Rec7* 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 7*. 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 7 (File Submission)* on moodle.

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

- 4 points for <u>code compiling and running</u>
- 6 points for <u>two test cases</u> for each function in main
- 10 points for an <u>algorithm description</u> 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/…
// Homework 3 - Problem # ...
```

  - comments for functions / test cases



Recitation 7: Parsing and 2D array

Computers can store information in many different formats. We have used integers, floating point values, strings and characters. When we print information to the console for a user to read, we convert the integer and floating point values to characters that can be displayed on the screen. When we get data from the user, they type individual characters and press the return key to send the data to our program.

The ***cin*** object has been interpreting the characters from the user and converting them into integer and floating point values for use in our programs. We are going to control the parsing of the string of characters typed in by the user. We will get data from the user in whole lines of text, that is a sequence of characters up to the end of line character. The ***getline*** function will allow us to read the sequence into a string variable.

The example below will read a line of text from the user (***cin*** specifies where to read from) and will echo the string back to the console.

```
string strX;
cout << "Enter a line of text: ";
getline(cin, strX);
cout << "User typed: [" << strX << "]" << endl;
```

Here is an example of the interaction when the program is run:

```
Enter a line of text: You have been working for 123 minutes
User typed: [You have been working for 123 minutes]
```

The numbers typed are just characters in the string, they have not been converted to an integer value in the program. However, we can parse out the numbers (get the sub-string with only the numbers) and convert them from string to integer or floating point values. In this case, we need to find the number within the string and then we can convert that substring into an integer.

You have learned many ways to search through a string and find substrings. One that will be very useful in the near future is to break a string up into words (an array of strings made up of the contiguous non-space characters) . If you performed that task on the string above, you could search the words for a string that was all digits and be able to convert it.

If we were to *chop* this sentence into words, we would end up with each of these strings:

```
You
have
been
working
for
123
minutes
```

Only one of these values is all digits and could be converted into an integer value.

In our examples, we will be using data that will come from text files. Data within files can be organized in many ways. Frequently consecutive values on the same line are separated by some specific character called a delimiter. In CSV (Comma Separated Values) files the delimiter is a comma. In TSV (Tab Separated Values) the delimiter is a tab (the escape character for a tab is '\t'). We can use the same type of function to break up a line of text into fields that are separated by a specific character. In our example above, we used a space character (' ') to delimit the words. Generically, we want to collect the characters until we encounter the delimiter.

Once the strings have be separated, we need to convert the values from the string into an integer or floating point value. We could write the code to interpret the individual digits and generate the correct value, but there are already functions to do this for us.

For example:

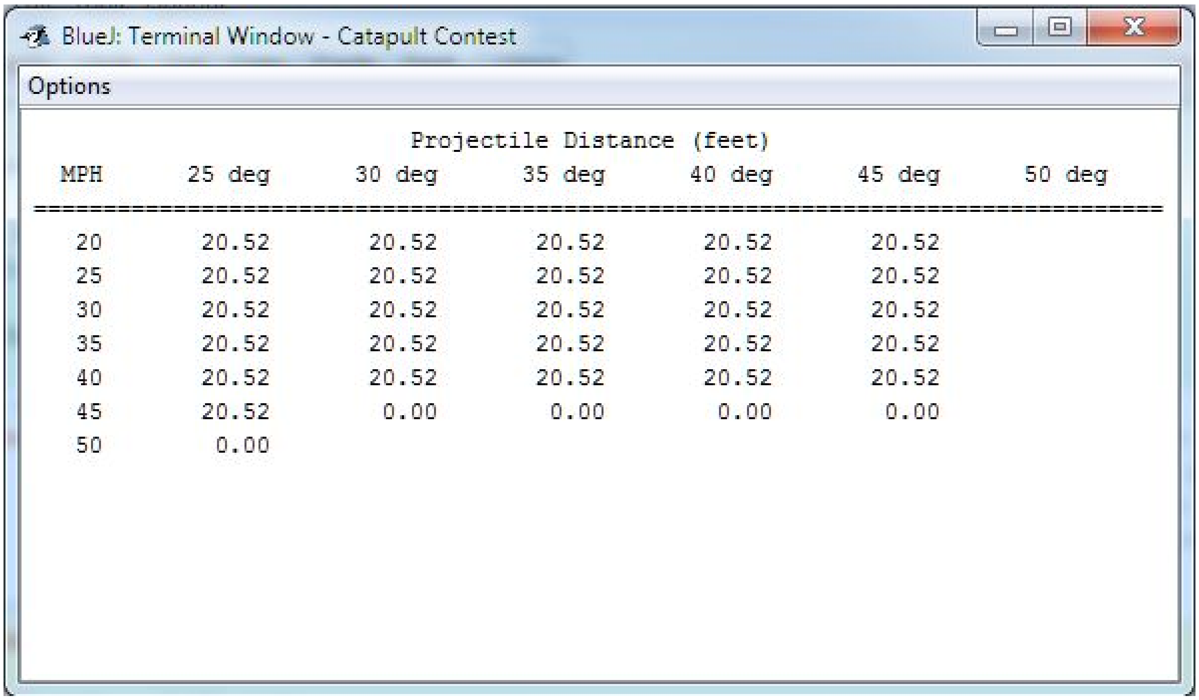
```
    string str = "123";
    int x = stoi(str);      //Converts the string into an integer.

    string strPi = "3.14159";
    float pi = stof(strPi);  //Converts the string into a float.
```

You should get to know how these functions will work on different strings. Test these functions out on your own. What happens when the string argument to ***stoi*** or ***stof*** is not directly representable as an ***int*** or ***float***? For instance, what do ***stoi("123.45")*** and ***stoi("123abcd")*** return? Do they work at all?

Challenge problem: create a program to convert the sequence of digit characters into an integer or floating point value.

2D arrays

Often it is useful to have lots of data associated with each index. Consider a table with data, such as wind chill values or the catapult data shown below. We may want to store multiple values for each row of information. If we consider the catapult data as a 2D array, the speed will specify which row we want to look at and the angle will determine which column to look at for that row. In this case, the speed and angle are the index values used to find the resulting distance.


BlueJ: Terminal Window - Catapult Contest

Options

Projectile Distance (feet)

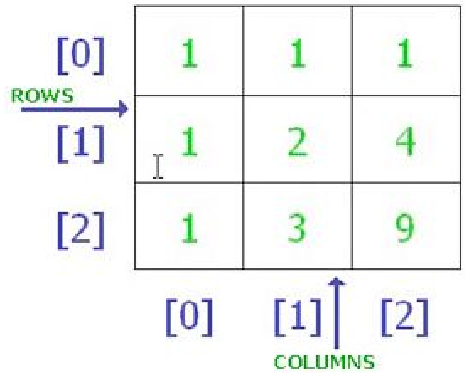
MPH	25 deg	30 deg	35 deg	40 deg	45 deg	50 deg
20	20.52	20.52	20.52	20.52	20.52	
25	20.52	20.52	20.52	20.52	20.52	
30	20.52	20.52	20.52	20.52	20.52	
35	20.52	20.52	20.52	20.52	20.52	
40	20.52	20.52	20.52	20.52	20.52	
45	20.52	0.00	0.00	0.00	0.00	
50	0.00					

We can create multiple dimensional data in C++ using arrays.
To create a 3 by 3 array shown below, we must tell the C++ compiler that we are creating a number of rows in an array AND that each row will have three elements in that row. We specify each dimension in its own set of brackets.

```
int table[3][3];
cout << table[0][0] << endl;
cout < table[2][1] << endl;
```

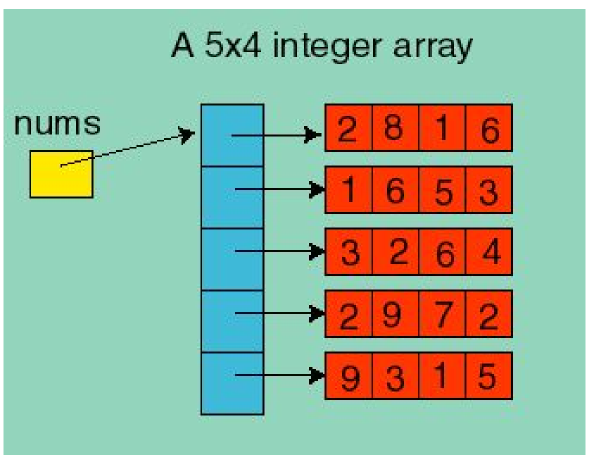
Output:

    1
    3



Multiple dimensional arrays are actually built as an array of arrays. In our two dimensional example, the program will build an array of three elements, where each element is an array of three integers.
For a 5x4 array example where *nums* is a multiple dimensional array, the layout of the memory will look like the figure below.

```
int nums[5][4];
```



To access the array we will need to use nested loops, one for indexing through each array:

```
int nums[5][4];
for (int row = 0 ; row < 5 ; row++)
{
    for (int col = 0 ; col < 4 ; col++)
    {
            cout << setw(4) << nums[row][col];
     }
    cout << endl;
}
```

When passing a multi-dimensional array as a parameter, *you must specify the size of the all dimensions except the first.* In our case, we have a two dimensional array, we can pass the array to a function, but we must specify the number of columns in each row.

```
float CalcAverage(int array[][4], int n_rows)
{
}
```

The function above is told it is a two dimensional array with 4 integer elements per row. The array itself could be any number of rows. Therefore we also send another parameter that specifies how many rows to process.

Problem Set

Question 1

Write a function **Split(...)** that is given a string, a separator character, an array to fill and the max number of elements in the array as parameters and returns an integer result. The function will separate each substring of the given string between separator characters and place it in the array. The function returns the number of substrings extracted.

For example:
```
string words[10];
Split("cow/chicken/fish", '/', words[], 10 );
```

would return the count of 3 and fill the array with "cow", "chicken", "fish".

Question 2

Write a function **GetScores(...)** that is given a string, an int type array to fill and the max number of elements in the array as parameters and returns an integer result. The function will find each substring of the given string separated by space characters and place it in the array. The function returns the number of integers extracted.

You should make use of the **Split(...)** function you wrote for Question 1.

For example:
```
int values[3];
GetScores("123 456 789", values, 3 );
```

would return the count of 3 and fill the array with integers 123, 456, 789.

Note: The write-up explains a built-in function available which converts a string into an integer.

Question 3

Write a function fillMatrix which takes 2 parameters- an int type 2-D array and number of rows. There will always be 4 columns in the array. The function will fill the array such that each element in the array is the sum of its indexes.

```
int matrix[10][4];
fillMatrix(matrix, 10);
```

For example, the value at matrix[3][4] will be 3+4 = 7.

Question 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 `"_"`.

An example map might look like this:

```
*_**
*__*
_*__
****
```