

## CSCI 1300 - Starting Computing

Instructor: Fleming

### Recitation 6

This assignment is due **Friday, October 12th, by 11:55 pm**

- **All components (Cloud9 workspace and moodle quiz attempts) must be completed and submitted by Friday, October 12th, 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 6.

**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 **Rec6** 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 6**. 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 6

## Parsing and Arrays

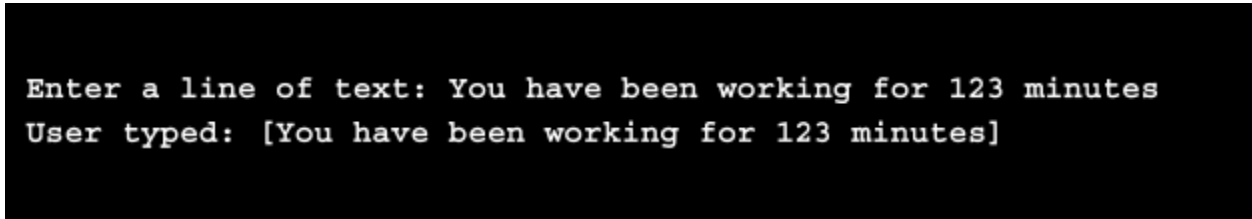
Computers can store information in many different formats. We have used integers, floating point values, strings and characters. When we print information to the console for a user to read, we convert the integer and floating point values to characters that can be displayed on the screen. When we get 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 substring 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.

## CSCI 1300 - Starting Computing

Instructor: Fleming

### Recitation 6

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 been 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.***

## CSCI 1300 - Starting Computing

Instructor: Fleming

### Recitation 6

## Arrays

An array is a *data structure* which can store primitive data types like floats, ints, strings, chars, and bools.

Arrays have both a **type** and a **size**.

### How to Declare Arrays

```
data_type array_name[declared_size];  
bool myBooleans[10];  
string myStrings[15];  
int myInts[7];
```

### How to Initialize Arrays (Method 1)

```
bool myBooleans[4] = {true, false, true, true};
```

If you do not declare the size inside the square brackets, the array size will be set to however many entries you provide on the right.

```
bool myBooleans[] = {true, false, true}; // size = 3
```

### How to Initialize Arrays (Method 2)

You can also initialize elements one by one using a for or while loop:

```
int myInts[10];  
int i = 0;  
while (i < 10)  
{  
    myInts[i] = i;  
    i++;  
}  
//{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
```

### How to Access Elements in an Array

We have essentially already had practice with accessing elements in arrays, as in C++, strings are arrays of characters.

```
//accessing character at position 1  
string greeting = "hello";  
cout << greeting[1] << endl;
```

## CSCI 1300 - Starting Computing

Instructor: Fleming

### Recitation 6

'h'	'e'	'l'	'l'	'o'
0	1	2	3	4

You can access elements in arrays using the same syntax you used for strings:

```
string greetings[] = {"hello", "hi", "hey", "what's up?"};  
cout << greetings[3] << endl;
```

"hello"	"hi"	"hey"	"What's up?"
0	1	2	3

Remember how we iterated through individual characters in a string:

```
string greeting = "hello";  
int i = 0;  
while (i < greeting.length()){  
    cout << greeting[i] << " ";  
    i++;  
}
```

**Output:**

h e l l o

You can similarly iterate through arrays of other types in the same way. Below we are iterating through an array of strings:

```
string greetings[] = {"hello", "hi", "hey", "what's up?"};  
int size = 4;  
int i = 0;  
while (i < size){  
    cout << greetings[i] << endl;  
    i++;  
}
```

**Output:**

hello  
hi  
hey  
what's up?

## Passing By Reference vs. Passing By Value

### Passing By Value

Up until now, when calling functions, we have always *passed by value*. When a parameter is passed in a function call, a new variable is declared and initialized to the value *passed* in the function call.

Observe that the variable `x` in **main** and variable `x` in **addOne** are *separate* variables in memory. When **addOne** is called with `x` on line 9, it is the *value* of `x` (i.e. 5) that is *passed* to the function. This *value* is used to initialize a new variable `x` that exists only in **addOne**'s scope. Thus the value of the variable `x` in `main`'s scope remains unchanged even after the function **addOne** has been called.

```
1    void addOne(int x){
2        x = x + 1;
3        cout << x << endl;
4    }
5
6    int main(){
7        int x = 5;
8        cout << x << endl;
9        addOne(x);
10       cout << x << endl;
11    }
```

#### Output :

5  
6  
5

### Passing By Reference

Arrays, on the other hand, are *passed by reference*. When an array is passed as a parameter, the original array is used by the function.

Observe that there is only *one* array `X` in memory for the following example. When the function **addOne** is called on line 9, a *reference* to the original array `X` is *passed* to **addOne**. Because the array `X` is *passed by reference*, any modifications done to `X` in **addOne** are done to the original array. These modifications persist and are visible even after the flow of control has exited the function and we return to `main`.

## CSCI 1300 - Starting Computing

Instructor: Fleming

### Recitation 6

```
1      void addOne(int X[]){
2          X[0] = X[0] + 1;
3          cout << X[0] << endl;
4      }
5
6      int main(){
7          int X[4] = {1, 5, 3, 2};
8          cout << X[0] << endl;
9          addOne(X);
10         cout << X[0] << endl;
11     }
```

#### Output:

```
1
2
2
```

## Problem Set

### Problem 1

Write the code to **declare** the four arrays listed below. You are not writing a function; your code will be placed directly in to our `main()` function for evaluation. For testing/development in cloud9, simply write the code in your own `main()` function.

- `arrayOne` - an array of integers with 49 elements.
- `arrayTwo` - an array of floating-point values, one for each practicum
- `arrayTri` - an array of booleans for attendance in each of the 14 recitations.
- `arrayFor` - an array to hold the names of each month.

Note 1: Make sure that your four array names match the indicated names exactly.

Note 2: You do not need to fill in these array's values, only declare the appropriate type and length specified for each of the four arrays listed above.

### Problem 2

Write the code to **declare and populate** the three arrays listed below. You are not writing a function; your code will be placed directly in to our `main()` function for evaluation. For testing/development in cloud9, simply write the code in your own `main()` function.

- `temps` - an array of 10 floating point numbers initialized with -459.67 (absolute zero in Fahrenheit)
- `colors` - an array of the strings "Red", "Blue", "Green", "Cyan", and "Magenta", in that order.
- `sequence` - an array of the first 100 positive integers in order; 1, 2, 3, 4, ... etc.

Note: Make sure that your three array names match the indicated names exactly.



## CSCI 1300 - Starting Computing

Instructor: Fleming

### Recitation 6

#### Problem 3

Write a function **split** which takes four input arguments: a string to be split, a character to split on ("a delimiter"), an array of strings to fill with the split pieces of the input string, and an integer representing the maximum number of split string pieces. The function will split the input string in to pieces separated by the delimiter, and populate the array of strings with the split pieces up to the provided maximum number of pieces. Your function will return the number of pieces the string was split into.

- Your function should be named **split**
- Your function takes four input arguments:
  - The **string** to be split.
  - A delimiter **character**, which marks where the above string should be split up.
  - An array of **string**, which you will use to store the split-apart string pieces.
  - The **int** length of the aforementioned array, which indicates the maximum number of split-apart string pieces
- Your function returns the number of pieces the input string was split into.
- Your function does not print anything.
- If the input string is split into more pieces than the array of string can hold (more than the indicated length), your function should fill only as many words as it can, and return -1.

Note: It's possible that the string to be split will have the delimiter character more than once in a row. Sequences of repeated delimiters should be treated as a single delimiter. It's also possible for a string to start or end with a delimiter, or a sequence of delimiters. These "leading" and "trailing" delimiters should be ignored by your function. See the examples.

Example output:

Input	split returns	words value
<pre>string words[6]; split("one small step", ' ', words, 6);</pre>	3	{"one", "small", "step", ...
<pre>split(" one small step ", ' ', words, 6);</pre>	3	{"one", "small", "step", ...
<pre>split("cow/big pig/fish", '/', words, 6);</pre>	3	{"cow", "big pig", "fish", ...
<pre>split("cow/big pig//fish", '/', words, 6);</pre>	3	{"cow", "big pig", "fish", ...
<pre>split("unintentionally", 'n', words, 6);</pre>	5	{"u", "i", "te", "tio", "ally", ...
<pre>split("one small step", ' ', words, 2);</pre>	-1	{"one", "small"};

## CSCI 1300 - Starting Computing

Instructor: Fleming

### Recitation 6

#### Problem 4

Write a function **getScores** which takes three input arguments: a string of scores, an array of integers to fill with scores, and an integer representing the maximum number of values that array can contain. The function will find each substring of the given string separated by space characters and place the value represented by that string in to the array. Your function will return the number of integers extracted.

- Your function should be named **getScores**
- Your function takes three input arguments:
  - The **string** of space-separated scores.
  - An array of **int**, which you will use to store the score values.
  - The **int** length of the aforementioned array, which indicates the maximum number of scores you should fill.
- Your function returns the number of scores extracted.
- Your function does not print anything.
- You can assume that all scores are valid integers.
- If the input string has more scores than the array of int can hold (more than the indicated length), your function should fill only as many scores as it can, and return -1.

Note 1: To convert a string like "123" to the integer value 123, C++ has a built-in function, `stoi`. `stoi` can be called with a single argument, a string, and will convert that string to an integer value, which it returns. `stoi` examples can be found on page 3 of this document.

Note 2: We recommend that you include your `split` function from the previous problem in your solution to this problem.

Note 3: Your function should treat leading, trailing, and repeated instances of the space character ( ' ') in the same way as the `split` function (Problem 1) would with ' ' as the delimiter.

Example output:

Input	getScores returns	nums value
<pre>int nums[6]; getScores("15 2007 5", nums, 6);</pre>	3	{15, 2007, 5, ...}
<pre>getScores("123 456 789 000", nums, 6);</pre>	4	{123, 456, 789, 0, ...}
<pre>getScores("", nums, 6);</pre>	0	{...}
<pre>getScores(" 123  ", nums, 6);</pre>	1	{123, ...}
<pre>getScores("15 2007 5", nums, 2);</pre>	-1	{15, 2007}