CSci 1113
Lab Exercise 6 (Week 7): Arrays & Strings

Strings

Representing textual information using sequences of characters is common throughout computing. Names, sentences, text, prompts, etc. all need a proper representation. We've been using *string literals* since the first week of the course when we discovered how to write "Hello World" to the computer display.

In addition to representing non-numeric and qualitative information, string objects are frequently used in engineering and scientific applications to input and process large text files containing measurements, experimental test data, and so forth.

Comma Separated Value files (CSV)

One common format for representing large data files is the "comma separated value" (CSV) format. For example, if you have data in an Excel spreadsheet, it is a simple matter to output it to a file in CSV format. The CSV format is simplicity itself: each data element is stored as a *text* string separated from the succeeding value by a single comma (' , '). If the file data is tabular (rows and columns), the *row*s are separated using a single *newline* character ('`\n`'). This makes it possible to use a standard text-editor to view the contents of any CSV formatted file in order to determine its organization.

Getline()

Processing the data in a CSV formatted file requires that you identify and separate the individual *values* in each row. Individual rows are read using the `getline(`*stream*, *string*`)` function which returns the *entire* comma-separated row as a C++ string object. The row string must then be parsed by your program, separating values from the comma separators. This requires some fluency with manipulating strings which we will explore in this Lab Exercise.

Converting Strings to Values

The "values" that are obtained from each CSV string (row) are character strings. Before they can be used in a computation, they must be converted to numeric values (*floating-point* or *integer*). There are many clever ways to do this in C++, but the simplest method is to use the functions atof and/or atoi, found in the standard C++ library: <cstdlib>. These functions both take a *c_string* as an argument and return a double or int respectively. Recall that *c_strings* and C++ string objects are not the same thing! If the character string you wish to convert to a value is stored in a C++ string object, you first need to convert it to a *c_string* using the string class method c_str:

Arrays

Arrays are our first real look at *data abstraction*. An array is an *ordered collection* of data values, but it can be described as if it were a singular "thing". So, for example, a "deck" of cards might be represented using a 52-element *array* of characters or integers. We could subsequently pass this "deck" to a function that might "deal" the cards one by one.

Arrays or *lists* of data are integral to problem solving in every programming language. In this lab, we begin exploring this often-used and important concept.

Mystery-Box Challenge

Here is your next mystery-box challenge. What is the output produced by the following code segment?

```
int someArray[4][4], i, j;
for( i = 0; i < 4; i++ )
    for( j = 0; j < 4; j++ )
        someArray[i][j] = i + j;
for( i = 0; i < 4; i++ )
{   for( j = 0; j < i; j++ )
        cout << someArray[i][j] << " ";
    cout << endl;
}
```

Warm-up

1) **Simple 2D array**

Represent the following matrix using a two-dimensional array and write a nested for loop to initialize the elements **(do not initialize the array in the declaration):**

```
10  9   8

7   6   5

4   3   2
```

Stretch

1) Returning Individual CSV Values

Write a function named `nextString` that will return a single 'value' (i.e. substring) from a Comma Separated Value" string. Your function will take two arguments: a string variable containing a comma separated list of values, and an integer containing the starting index; your function should return a single string object with the value that starts at that index and ends right before the next comma ',' (do not include the comma in the returned string!) :

```
string nextString(string str, int start_index);
```

If, however, the start index is after the last comma in the string, then the function should return the value starting at that index and continuing to the end of the string.

For example,

```
cout << nextString("my,cat,ate,my,homework",3);
```

will print

```
cat
```

and

```
cout << nextString("my,cat,ate,my,homework",8);
```

will print

```
te
```

and

```
      cout << nextString("my,cat,ate,my,homework",18);
```
will print
```
      work
```

When you have written your function, then write a short test program that will take in a simple comma separated string using `getline`:
```
      getline(cin,somestring)
```
and output values in the string using the `nextString` function.

2) **Split**

Now, extend your test program by adding a second function named `split` that will identify *all* the individual values in a comma separated value string and return them in an array of string objects:

```
      int split(string str, string a[], int max_size);
```

Your function will take three arguments: a comma separated value string `str`, an array `a` of string objects, and the maximum size of the array. You must use the `nextString` function from Stretch Problem (1) to obtain each value in the string and store it in the array starting, with the first element. Return the total number of values stored in the array.

For example:

```
      string varray[VALUES];
      int cnt = split("my,cat,ate,my,homework",varray,VALUES);
      for( int i=0; i<cnt; i++)
         cout << varray[i] << endl;
```

Should produce:

```
      my
      cat
      ate
      my
      homework
```

3) **Earthquake Data, Part 1**

Large repositories of recorded measurement data are available on the World Wide Web from a wide spectrum of applications such as stock market data, weather/climate data, etc.

Use your Web browser to view the following URL:
```
      http://earthquake.usgs.gov/earthquakes/map/
```

This site is maintained by the US Geological Service and reports recent earthquake activity around the world. If you examine the upper left corner of this web page, you will see a link marked 'download'. Select this link and then choose 'CSV' as the file format. Now save the downloaded file in your home directory and then use a text editor to examine it.

The file contains a large quantity of information that is detailed in the first line of the file. We will only be interested in the magnitude of the earthquake and the place (the string indicating where the event occurred, not the latitude/longitude). As you examine the file, note that the `place` is actually *two* comma separated strings. The first begins with a double-quote and the second ends with another double quote. The "general" location we are interested in is the *second* of the two strings that make up the place description.

First, write a program that will read the file and output the categories from the first line, along with their relative location in the line:

Example:

```
0  time
1  latitude
2  longitude
3  depth
4  mag
5  magType
6  nst
7  gap
8  dmin
9  rms
10 net
11 id
12 updated
13 place
14 type
```

You should use the `getline()` function to read the first line of the file and the `split` function from Stretch 2 to create the value array.

Workout

1) **Earthquake Data, Part 2**

Now, modify your program to print out the magnitude and location of each earthquake in the file. Compare your program results to the actual file data to verify that it works correctly.

2) **Bubble Sort**

Sorting a list of numbers is an important Computer Science problem that has been extensively studied. One of the simplest methods is known as "bubble sort". Given a list of numbers:

```
3  5  2  8  9  1
```

the basic idea is to compare the first two numbers in the list and swap them if the first one is larger than the second (assuming you wish to sort from low to high). Next, the second and third numbers are compared and swapped if necessary, then the third and fourth, fourth and fifth, and so on until the entire list has been examined. After the first pass through, the list would look like this:

```
3  2  5  8  1  9
```

Note that the largest value will *always* end up in the last position after the first pass.

Next, we repeat the process, “bubbling” the larger values up in the list on each pass. Note however, that for the second pass we don’t need to examine the last value because it’s guaranteed to be the largest. After the second pass, the last *two* values need not be examined, and so on.

The process ends when an entire pass through the list results in no values being swapped.

Write a program that will implement the Bubble-Sort algorithm. For this problem you need to do the following:

- Declare an integer array named *list* that contains 50 values.
- Using a loop, initialize *list* with the values 100 , 99, 98, … (in decreasing order)
- Construct a void function named `bsort` that implements the Bubble Sort algorithm as described above. Your function should accept two arguments: the integer array to be sorted and the number of elements in the array.
- Call the `bsort` function to arrange the elements of *list* in increasing order.
- Print out the elements of *list*, 5 per line as follows:

Example:
```
    51      52      53      54      55
    56      57      58      59      60
```
(and so on…)

[Hint: This problem will be much easier if you attack it in stages. First, write a `swap` function that will swap two integer values, then write another function that will make a single pass through the array, calling the `swap` function to correct any out-of-order elements. Finally, call this second function as many times as necessary to sort the array]

3) Steady State Temperature in a Thin Metal Plate

Imagine a thin metal plate surrounded by heat sources along each edge, with the edges held at different temperatures. After a short time, the temperature at each location on the plate will settle into a steady state. This can be modeled by dividing the plate into a discrete grid of cells and simulating the change in temperature of each cell over time:

100	100	100	100	100
90	$T(1,1)$	$T(1,2)$	$T(1,3)$	95
90	$T(2,1)$	$T(2,2)$	$T(2,3)$	95
90	$T(3,1)$	$T(3,2)$	$T(3,3)$	95
80	80	80	80	80

At each time step, the temperature in each interior cell (*i,j*) will be the average of four of its surrounding neighbors:

T(i,j) = (T(i-1,j) + T(i+1,j) + T(i,j-1) + T(i,j+1))/4

a) Part 1

Develop a program to determine the steady state temperature distribution in the plate by representing the plate as a two-dimensional array with `NROWS` rows and `NCOLS` columns. `NROWS` and `NCOLS` should be declared as global *constants* (suggestion: use 20 for both values).

Your program should do the following:

- Declare 2 separate two-dimensional arrays named `temp` and `old.` Array `old` will be used to maintain the *current* values of the grid temperatures and array `temp` will be used to compute the *new* values obtained at each successive time step using the equation above.
- Prompt the user and enter temperature values for the top, bottom, left, right sides. Also prompt and enter an initial temperature T(i,j) for the interior cells. (For this lab, you may initialize all the interior cells to a single, user-input temperature.)
- Initialize `temp` using these values and display the initial contents of the `temp` array on the console.
- Obtain a convergence criterion from the user (say, 0.001)
- Use a convergence loop to continue updating the `temp` array until the temperature in every cell converges using the following method:
    1. Set a *boolean* variable named `steady` to `true`
    2. Copy `temp` to `old`
    3. Loop over all the *interior* cells (but not the edge cells!)
       `temp[i][j] = 0.25*(old[i][j-1] + . . .)`
       If `|temp[i][j] – old[i][j]| > convergence_criterion,`
       then set `steady` to `false`

  Repeat this 3-step process again and again until all the cells simultaneously satisfy the convergence criterion.

This is a longer program than usual, so here are a number of hints:

- Before writing any code think about what the code should look like: What will an outline look like? Will it involve loops? If so, how many, where, and of what type? Will there be if statements? If so, where and how many? What variables will your program need to keep track of? Which of these will be arrays? Think about these questions and discuss them with your partner before actually writing any code.
- Write, test, and debug your code incrementally, rather than all at once.
- Create a function named `void display(double temp[][NCOLS]).` You can use this function to print out that array at any time, and so the function will be useful in debugging your program.
- Make sure you can explain the difference between the `old` array and the `temp` array, and why your program needs to use both.
- Make sure you understand how to have your program loop through only the *interior* of the array.
- Make sure you understand what the convergence criterion test is, and what role it plays in your program.

b) Part 2 (optional)

Now you will *visualize* the steady state temperature data by creating a 3D plot using Matlab. Begin by modifying your program so that the final temperature data is written to a file named **`temp.dat`** instead of (or in addition to) the terminal screen. Then open Matlab by typing the following commands:

```
%module load math/matlab

%matlab
```

When the application opens, move the cursor to the Command window and enter the following Matlab commands in order and observe what happens:

```
nrows = 20

ncols = 20

[x,y]= meshgrid(1:nrows,1:ncols)

load temp.dat

mesh(x,y,temp)

surf(x,y,temp)
```

The mesh function produces a mesh or wireframe plot of the data. The surf function produces a surface plot. Try using the rotate button from the menu bar to have a good look at the data!

Check

Individually, list one important things you learned in lab today, one question you still have about the lab material, and one way arrays are used in your major. When you are done, share your list with your partner.

Challenge

Here is an extra challenge problem. You should try to complete the warm-up, stretch and workout problems in the lab. Try this challenge problem if you have extra time or would like additional practice outside of lab.

1) Tic-Tac-Toe, part 1

The game of Tic-Tac-Toe (or Naughts and Crosses) is an ancient game in which 2 players alternate turns placing either X's or O's on a 3x3 square grid. One player places X's on the grid and the opposing player places O's. The first player to place 3 consecutive X's (or O's) on any row, column or diagonal of the grid wins the game. If all 9 grid cells are occupied without either player winning, the game is declared a draw.

Write a C++ function that takes a 3x3 array of characters representing a tic-tac-toe game in progress and determines the current game state: player X has won, player O has won, the game is a draw, or none of those.

Your function should return a single character as follows: 'X' if player X has won the game, 'O' if player O has won the game, 'D' if the game is a draw (all cells occupied but neither player has won) and '*' if none of these conditions exist. (Hint: use a `for` loop that compares each element of the diagonal with the elements of its intersecting row and column, then check the diagonals)

Write a `main()` driver to verify that your function is correct. It should (1) declare a 3x3 character array, (2) use a loop to read in values for each of the 3 rows of the grid (use '-' to indicate a blank cell), (3) print out the contents of the grid as 3 rows of 3 values, (4) call your function with the array as an argument, and (5) output an appropriate message declaring the state of the game (as returned by your function).

Test your program using the 4 following cases (at a minimum):

```
X – O
X O –
X – O

O – X
X O –
X – O

- X O
X O –
X – O

O X O
X X O
X O X
```

2). **Earthquake Report**

Modify the bubble-sort function to sort the rows of a two dimensional array in descending order of the first element in each row. That is, as it sorts the first element in each row, it should move entire rows so that each row of data stays as a row throughout the sort.

Next, read the earthquake data and store only the magnitude and location in a two dimensional array with *n* rows and two columns.

Now sort the earthquake data in descending order of magnitude, and print out a list of all the earthquake magnitudes and their associated locations in order from the highest to the lowest. Note you will need to store the magnitude data as a string object in the array, but use its equivalent floating-point *value* for the comparison. Can you describe why?

[Hint: you can use the `atof() or stod()` function in your sort comparison.]