Test 02

CSci 515 Spring 2015

2015-05-04

Dates:

Due: In Lab, Wednesday March 11, by 4:05 pm (lab end time)

Description

You have 2 hours (our regularly scheduled lab time) to complete the following tasks. Create a single file, named test02.cpp, in a visual studio project called Test02. Set up the Visual Studio projects using the normal settings for our class and labs, as we have practiced for the past 15 weeks. The file you submit to eCollege should be the resulting test02.cpp. It should contain only a single `main()` function, and any other functions and code you are asked to write for the following tasks.

Perform the following tasks:

1. Here is an implementation of the `bubbleSort()` function we used in class to sort an array of integers in ascending order:

```
/** Bubble Sort
 * Sort an array of integers using a Bubble sort.
 * Bubblesort works in this manner.  On the first pass we start at
 * index 0 and compare successive items.  We swap the items if they
 * are out of order.  The result is that on the first pass, the
 * largest item will be "bubbled up" to the largest index.  On the
 * next pass, we do the same thing, but since the last item is already
 * bubbled into place, we only pass through the N-1 items.  We do this
 * for N passes.  Bubble sort is very inefficient, it is an O(N^2)
 * algorithm.
 *
```

```
 * @param values An array of integers.  The array to be sorted.  The
 *   array is passed by reference and is sorted in place in memory.
 *   The array is sorted in ascending order.
 * @param size int The size of the array to sort.
 *
 * @returns void Nothing is returned explicitly but as a
 *   result of calling this function the array
 *   that is passed in will be sorted into ascending order.
 */
void bubbleSort(int values[], int size)
{
  // outer loop, perform N passes
  for (int pass = 0; pass < size; pass++)
  {
    // inner loop, bubble up items from index 0 up to size-pass-1 index
    for (int idx = 0; idx < (size - pass - 1); idx++)
    {
      // if the values are out of order, swap them
      if (values[idx] > values[idx + 1])
      {
        int tmp = values[idx];
        values[idx] = values[idx + 1];
        values[idx + 1] = tmp;
      }
    }
  }
}
```

Modify this function to accept an array of floating point values instead. Then modify the function to sort the values in descending rather than ascending order (e.g. The largest float should end up at index 0, the next largest at index 1, etc. and the smallest at the last index in the array). In your `main()` function, create an array of 5 floats, and initialize the array with the following values: `{-3.8, 4.2, 9.7, -2.5, 5.6}`. Demonstrate calling your modified `bubbleSort()` function with this array of floats, and use a loop in your `main()` function to display the values in the array after the array has been sorted. See the example output below for how you should format your resulting output for this task 1.

2. Create a structure called `Data`. This structure should have 3 member fields. The first field is called `speed`, and should be of type float, and the second field is called `rank` and should be of type int. The third field is a discrete category variable. You should define an enumerated type, called `Category`. The valid categories are `HIGH_PERFORMANCE`, `MID_PERFORMANCE`, `LOW_PERFORMANCE`. The `Data` structure should have a third field named `perfCategory` of type `Category`. Write a function called `generateData()` that takes an array of `Data` items as its first parameter, and an integer variable called `size` as its second parameter. This function should initialize all of the fields in each item of the given array of `Data` items with random values. For the `speed` float, create a value in the range from 0.0 to 10.0. For the `rank` integer, create a value in the range from 0 to 10. And generate a random number from 0 to 2 to use to randomly pick one of the performance categories for the `perfCategory` field. In your main function, create an array of 20 `Data` structures, and demonstrate calling your `generateData()` on this array to randomly initialize the fields of all of the `Data` items with random data.

3. The following is the simple definition of a self-referential structure we used in class for creating linked lists

```
// A self-referential structure
struct Node
{
  int data;
  Node* nextPtr;
};
```

Add this structure definition to your test02.cpp file. In your `main()` function, create a linked list by hand of 4 nodes. Name the nodes `node1`, `node2`, `node3` and `node4`, and initialize them with the integer values 10, 20, 30, 40, respectively. Also link together the nodes into a linked list, such that `node1` is the head node, and it points to `node2` which points to `node3` which points to `node4`. `node4` should also be correctly initialized to be the final node in the linked list (using the NULL pointer convention). Create a pointer to a `Node` item, and set it so it is pointing to the head `node1` of your linked list. Demonstrate accessing the value in `node4` from your pointer to the head node using a single output statement (e.g. starting from your pointer to the head

node, follow the nextPtr pointers till you arrive at `node4` and then access its value). An example of the desired output for this task 3 is shown in the example output below.

4. Write a function to insert a node into a linked list of nodes at the end of the linked list. This function should be called `insertAtBack()`. This function will take a pointer to the head of a linked list of `Node` as the first parameter and to a single unlinked `Node` as the second parameter, which will be inserted on the end of the list. This function should insert the given `Node` on to the end of the linked list of nodes it is given. For this test, you can ignore the case were the given list of nodes is empty, and for now just assume you are always given a valid list of nodes with at least 1 node in the list. In your main function, create a new node called `node5` and initilaize it with the value 50. Demonstrate calling your function in `main()` by having it append this node `node5` to the end of the list you created by hand in task 3.

Your program output for the 4 previous tasks should look something close to the following when I run your program:

Test Submission

An eCollege dropbox has been created for this test. You should upload your version of the test by the end of test time to the eCollege dropbox named `Test 02`. Work submitted by the end of the allotted time will be considered, but after the test ends you may no longer submit work, so make sure you submit your best effort by the test end time in order to receive credit.

Requirements and Grading Rubrics

Program Execution, Output and Functional Requirements

1. Your program must compile, run and produce some sort of output to be graded. You will loose at least 1/3 of the total points (33) if your program does not compile and run when submitted.

2. 10 pts (1 letter grade). Up to 1 letter grade will be awarded for formatting and style issues for the test. Your program must meet (most) all of the standard class style/formatting guidelines that we have been practicing and using in our labs and assignments for this course.

3. 20 pts. Task 1. You must successfully modify the sort function as required in task 1 and demonstrate calling it.

4. 25 pts. Task 2. You must define the structure and enumerated type as described. Your function must work to correctly initialize an array of `Data` structures with random values as described.

5. 20 pts. Task 3. You must correctly create the indicated linked list by hand as described. You must demonstrate following pointers to get the value at `node4` from the head node as required.

6. 25 pts. Task 4. You must correctly define the insert at back function as described, and it must be implemented correctly.

Program Style

Your programs must conform to the style and formatting guidelines given for this course. The following is a list of the guidelines that are required for the lab to be submitted this week.

1. The file header for the file with your name and program information and the function header for your main function must be present, and filled out correctly.

2. A function header must be present for all functions you define. You must document the purpose, input parameters and return values of all functions. Your function headers must be formatted exactly as shown in the style guidelines for the class.

3. You must indent your code correctly and have no embedded tabs in your source code. (Don't forget about the Visual Studio Format Selection command).

4. You must not have any statements that are hacks in order to keep your terminal from closing when your program exits (e.g. no calls to system()).

5. You must have a single space before and after each binary operator.

6. You must have a single blank line after the end of your declaration of variables at the top of a function, before the first code statement.

7. You must have a single blank space after , and ; operators used as a separator in lists of variables, parameters or other control structures.

8. You must have opening `{` and closing `}` for control statement blocks on their own line, indented correctly for the level of the control statement block.

9. All control statement blocks (if, for, while, etc.) must have `{ }` enclosing them, even when they are not strictly necessary (when there is only 1 statement in the block).

    (a) You should attempt to use meaningful variable and function names in your program, for program clarity. Of course, when required, you must name functions, parameters and variables as specified in the assignments. Variable and function names must conform to correct `camelCaseNameingConvention` .

Failure to conform to any of these formatting and programming practice guidelines for this test will result in loosing 1 letter grade You can get a B for this test if you do it perfectly, but have bad or missing style/formatting. To get an A, however, you need to follow (most) of the style/formatting requirements for this course on your test code.