CSE-165-Lab 4

Write a separate .cpp file for each of the following tasks. For all the programs that you write, make sure there is no memory leak.

1. **(10 points)** Consider the file Stash.h. This file contains the Stash data structure from Chapter 4 of the textbook. Your task is to write a program that declares an instance of Stash, fills it up with double numbers and then prints out the numbers. You have to use the Stash member functions to complete this exercise.

   You need to read a file called input.txt, which contains floating points numbers at each line. Read in these values, store them in your Stash object and print out the values.

   After the floating points values have been printed out, use the appropriate member function of Stash to free up the memory your object occupied.

   Input input is in `input.txt`
   Output 16 2.3 -4.5 22 -0.4 87 34.9 -40.1 -10.5 8.8 101.4 2.56 -3.14 9.78 12.3 -3.1 1.2 freeing storage

2. **(10 points)** Consider the file Stack.h. This file contains the Stack data structure from Chapter 4 of the textbook. Your task is to write a program that declares an instance of Stack, fills it up with double numbers and then prints out the numbers. You have to use the Stack member functions to complete this exercise (except for printing).

   You need to read a file called input.txt, which contains floating points numbers at each line. Read in these values, store them in your Stack object and print out the values in the reverse order without deleting them from the stack (for printing you can directly access member variables). After the double values have been printed out, free the Stack by removing all elements with the pop() function. When the stack is empty, call the cleanup() function.

   Input input is in `input.txt`
   Output 1.2 -3.1 12.3 9.78 -3.14 2.56 101.4 8.8 -10.5 -40.1 34.9 87 -0.4 22 -4.5 2.3 16

3. **(15 points)** You are now going to create a LinkedList structure that will work very similarly to the Stack structure seen in the book (and used in the previous exercise). The starter code is provided in `LinkedList.h`. After you complete the `LinkedList.h`, create another program which will use your `LinkedList.h` similar to the previous exercise. That is, your code should read in floating point numbers from `input.txt`, and store them in a LinkedList. Then it should iterate through the LinkedList, and print out the stored values. Then it should clear up LinkedList, and free all the dynamically allocated memory.

   Input input is in `input.txt`
   Output 16 2.3 -4.5 22 -0.4 87 34.9 -40.1 -10.5 8.8 101.4 2.56 -3.14 9.78 12.3 -3.1 1.2

4. **(10 points)** One of the problems of our Stack and LinkedList structs is that if we make them handle generic types with void* pointers, they will not know how to delete the elements we insert in them. We will see later how templates will solve that, but for now let's practice using pointers to functions. Extend the book's Stack struct with one more member variable that will hold a pointer to the following function prototype:

   ```
   void deletecb (void *pt)
   ```

   You will then add a member function to set this pointer:

   ```
   void Stack::setDeleteCallback ( void (*delcb) (void * pt) )
   ```

After you do this, then add the corresponding code in the cleanup method to traverse all elements of the stack, deleting the links and calling the delete callback once for each stored void pointer. The user will be responsible for providing the delete callback and implementing it with the correct delete call after converting the void pointer argument to its correct type.

You may download the Stack.h file, make the modifications described above and submit it. Your code will be evaluated using voidPointers.cpp.

Input
3
11
-4
2
Output
Deleting: 2
Deleting: -4
Deleting: 11

5. **(10 points)** Create a struct named Entry that stores an entry of an address book. It should have fields for first name, last name and email address. Provide the appropriate getter and setter functions for each one of the fields. In addition provide a function called print. This function should print out the information in the class in the way it appears in the sample.

   Your code should go into a file named `Entry.h` with all the member functions implemented inline (in the header file). Your code will be tested using the file `address_book_entry.cpp`.

   Input
   Ann
   Bob
   annbob@merced
   Output
   First Name: Ann
   Last Name: Bob
   Email: annbob@merced

6. **(15 points)** Create a struct named AddressBook that stores multiple Entry structs (that you created in the previous exercise). Your code must be saved in a file named AddressBook.h and should use Entry structs from Entry.h.

   Member functions for AddressBook should be able to add an entry and print all the entries in the address book. Your code will be tested with the file addressBook.cpp, which reads in several entries from standard input, stores them in an AddressBook instance and prints them all out.

   Input
   2
   Ann
   Williams
   ann@merced
   Bob
   Williams
   bob@merced
   Output
   First Name: Ann
   Last Name: Williams
   Email: ann@merced

   First Name: Bob
   Last Name: Williams

Email: bob@merced

7. **(15 points)** How big is a structure? Write a piece of code that prints the size of the structures specified below, in the order they appear.
   - containing 1 char and 1 double
   - 2 chars and 1 double
   - 3 chars and 1 double
   - 4 chars and 1 double
   - an empty struct
   - 1 char, followed by 1 int and then 1 char
   - 2 chars followed by 1 int

8. **(15 points)** In the following exercise you will again use the Stash structure, however with two modifications:
   (a) you will have a new integer member to store the desired increment to be used during re-allocation (not a fixed 100 value as in the book)
   (b) you will have another integer member variable to count the number of re-allocations (calls to the inflate method) the Stash structure needed during its use.

   You will now write a variation of the run-length encoder you wrote in the previous lab using the Stash structure with each entry being one character. First of all you will read as input an integer that will tell how much the Stash should use as increment. Then you will read a sequence of pairs, each pair containing a character and a number. For each pair (C,N), add to a Stash object the character C N times without spaces. If a pair has a negative N number, then add the character C $|N|$ times and then add a newline character. If a pair (&,99) is read, then stop reading values, print the elements in the Stash in the order received, and then print two numbers: the number of calls that were made to inflate() inside your Stash object, and the total size in bytes that was allocated by the Stash object at the end (the value of the quantity variable).

   Input
   20
   a
   4
   b
   5
   c
   10
   &
   99
   Output
   aaaabbbbbccccccccc1
   20