CS 111 - Homework 10

Deadline

11:59 pm on **Friday, December 2, 2016**

How to submit

Each time you would like to submit your work:

- If your files are not already on nrs-labs, be sure to use sftp/WinSCP/FileZilla to get them TO nrs-labs (ideally in a folder/directory named `111hw10`).
- Then, if you are not already logged onto nrs-labs, then use PuTTY/`ssh` to do so, and use `cd` to change to the folder/directory where your homework files are -- for example,

  `cd 111hw10`
- Use the `ls` command to make sure your desired `.cpp` and `.h` files are really there:

  `ls`
- Use `~st10/111submit` to submit them, with a homework number of **10**
  - Make sure that `~st10/111submit` shows that it submitted **ALL** of your `.cpp` and `.h` files you were intending to submit!

Purpose

To use the design recipe to design and implement C++ functions, including `main` functions, involving local variables, assignment statements, `cout` and `cin`, and now also count-controlled `while` loops.

Important notes

- As always, start the homework problems early! Then you'll have time to e-mail me your `.cpp` and `.h` files along with the error message(s) if you run into an error you don't know how to handle.
- Be sure to follow the course coding style discussed in class and demonstrated in posted examples. In particular, remember to indent as shown and demonstrated in posted examples.
- You are *no longer required* to use `funct_play` to develop your C++ functions. You *may* still use it if you would like (keeping in mind its limitations for `main` and `void` functions!).
  - If you are not using `funct_play`, note that **you need to include all of the components shown in the given templates** (for `main` functions, non-`main` functions, and `.h` files) on the public course web page, whether you actually use those templates or not.
- Remember that you can JUST compile any single C++ function on nrs-labs with the command:

  `g++ -c` *filename*`.cpp`

  (If all goes well, this doesn't print anything to the screen, but it does result in a file *filename*`.o`, ready to be linked and loaded with other files to create a C++ executable file.)
- Remember that you can compile, link, and load to create a C++ executable program on nrs-labs using:

  `g++` *main_filename*`.cpp` *other1*`.cpp` *other2*`.cpp ... -o` *main_filename*

  ...being sure to include the `.cpp` files for ALL functions used in that program;

  - Also, to help you write these `g++` calls to compile, link, and load your C++ programs (to result in C++ executable programs), there is a little script `compile-helper` on nrs-labs that asks you to enter details about your program-to-be and then "builds" an appropriate `g++` command, which it runs as well as prints out so you can gradually learn how to write these yourself.
  - (NOTE: it assumes you have already written all of the `.cpp` and `.h` files for your program, and have transferred them all to your current working directory on nrs-labs.)

- You **are, of course, still expected** to follow the Design Recipe for all **functions** that you design/define, whether they are non-`main` or `main` functions.
  - Remember, you will receive **significant** credit for the signature, purpose, header, and examples/tests portions of your functions.
  - Typically you'll get at least half-credit for a correct signature, purpose, header, and examples/tests, even if your function body is not correct.
  - (and, you'll **lose at least half-credit** if you omit these or do them poorly, even if your function body is correct).

HOMEWORK 10 SETUP

I'm trying an experiment this homework, and putting copies of the `.h` and `.cpp` files for functions from previous in-class examples/homeworks/lab exercises that are also used in this homework into a file on nrs-labs where you can copy them from.

For example, some of these problems use `starline`, from the Week 13 Lab Exercise -- in case your lab partner has your pair's copy, here's how you can get a copy of my example solution's `.h` and `.cpp` files:

- use PuTTY/ssh to connect to nrs-labs
- `cd` to your Homework 10 directory:

  ```
  cd 111hw10
  ```

- use the following command to copy these "lab exercise example solution" versions of `starline.h` and `starline.cpp` to your Homework 10 directory,

  BEING CAREFUL TO NOTICE THAT THIS COMMAND ENDS WITH A BLANK, THEN A PERIOD (which is a nickname for your current directory):

  ```
  cp  ~st10/for-111hw10/starline*  .
  ```

- And, some use function `cheer` during Week 12 Lecture 1 -- I've put copies of its `.h` and `.cpp` files in `~st10/for-111hw10`, also, so you can make copies of those for Homework 10 using the command:

  ```
  cp  ~st10/for-111hw10/cheer*  .
  ```

- While you are at it, also copy your function `salutation` (at least its `.h` and `.cpp` files) from Homework 8, Problem 2. BUT -- JUST in case -- I've put the example solution versions of these in `~st10/for-111hw10`, also, so you can make copies of those for Homework 10 using the command:

  ```
  cp  ~st10/for-111hw10/salutation*  .
  ```

- And, copy your function `get_worth` (at least its files `get_worth.h` and `get_worth.cpp`) from Homework 7, Problem 5. BUT -- JUST in case -- I've put the example solution versions of these in `~st10/for-111hw10`, also, so you can make copies of those for Homework 10 using the command:

  ```
  cp  ~st10/for-111hw10/get_worth*  .
  ```

- list your current directory's contents to see that all of these copy commands worked:

  ```
  ls
  ```

  - If you see copies of `cheer.cpp`, `cheer.h`, `get_worth.cpp`, `get_worth.h`, `salutation.cpp`, `salutation.h`, `starline.cpp`, and `starline.h`, all went well.

Problem 1

Problem 1 part a

(**NOTE** - this problem's new function does NOT directly involve a loop, although some of the functions it calls happen to contain a loop. It is to give you some additional practice with a function that calls multiple other functions, has some side-effects in addition to its return value, and maks use of a local variable.)

Recall that you wrote a function `starline` for the Week 13 Lab Exercise -- it expects a desired number of stars/asterisks, has the side-effect of outputting a line of that many asterisks to the screen, and returns the number of asterisks printed to the screen.

And, recall function `cheer` from Week 12 Lecture 1; it expects a desired number of `HIP`s, has the side-effect of printing to the screen that many `HIP`s then `HOORAY!`, and returns the number of `HIP`s printed.

And, recall function `salutation` from Homework 8, Problem 2, which expects a name, has the side-effect of printing a letter-style salutation to that name, and returns the length of the given name.

(Remember: the **HOMEWORK 10 SETUP** section above described how you can copy provided `.cpp` and `.h` files for these into your Homework 10 directory on nrs-labs.)

NOW: use the design recipe to write a function `encourage` that expects a name, returns the length of that name, and has the side-effects of:

- using `salutation` to print a salutation for that name,
- then using `cheer` to print a cheer with the number of `HIP`s equal to the length of the given name,
- then using `starline` to print a line of *'s the same length of the given name on the next line,
- then printing the given name on the line after that,
- then using `starline` again to print a line of `*`'s the same length of the given name on the next line, ending with a newline.

For example, `encourage("Carla") == 5` and should have the side-effect of printing to the screen:

```
Dear Carla,
HIP
HIP
HIP
HIP
HIP
HOORAY!
*****
Carla
*****
```

Here's one more requirement: you are required to **appropriately** declare, initialize, and use a **non-parameter local variable**, whose value is the length of the parameter name.

Hints:

- In `encourage.cpp`, remember to include the `.h` file for EACH function called by `encourage` .
- If you do this correctly, `encourage` should be a quite **short** function!
- Remember that, inside of a function, if you don't CARE about the value a function returns, but just want its side-effects, you can ignore the returned value by simply writing that function like a statement -- that is,

  `cheer(4);`
- Be careful with newlines here!

Problem 1 part b

To formally test your function, design a `main` function in a file named `encourage_test.cpp` that:

- prints a message saying that you are testing function `encourage`
- puts `boolalpha` into the `cout` output stream, so that `bool` values are printed as `true` and `false`
- because `encourage` has a desired side-effect in addition to its return value, for EACH of its examples/tests,
  - it should first print a message saying that what follows should be the name <print out the name> greeted, cheered and shown "in lights" followed by `true`,
  - and then put that example/test in its own separate `cout` statement, such that the result of that test will be printed on its own line.

Remember that you must include the `.cpp` files for ALL functions used in a program in its `g++` command:

```
g++ encourage_test.cpp encourage.cpp salutation.cpp cheer.cpp starline.cpp -o encourage_test
```

Submit your resulting files `encourage_test.cpp`, `encourage.cpp`, and `encourage.h`.

Problem 2

Problem 2 part a

Now, for a little imperative repetition/count-controlled `while` loop practice!

Consider -- what would you see on-screen if you called `starline` repeatedly?

Remember, `starline` expects a desired number of stars/asterisks, has the side-effect of outputting a line of that many asterisks to the screen, and returns the number of asterisks printed to the screen.

Using the design recipe, develop a C++ function `starbox` that expects a desired number of rows and a desired number of asterisks per row, has the side-effect of printing to the screen that many rows of asterisks, each with that many asterisks per row, and returns the total number of asterisks printed out. This function must appropriately call `starline`.

For example, `starbox(3, 5) == 15` and has the side-effect of causing the following to be printed to the screen:

```
*****
*****
*****
```

And, `starbox(4, 2) == 8` and has the side-effect of causing the following to be printed to the screen:

```
**
**
```

```
**
**
```

Problem 2 part b

To formally test your function, design a `main` function in a file named `starbox_test.cpp` that:

- prints a message saying that you are testing function `starbox`
- puts `boolalpha` into the `cout` output stream, so that `bool` values are printed as `true` and `false`
- because `starbox` has a desired side-effect in addition to its return value, for EACH of its examples/tests,
  - it should first print a message saying that what follows should be a box of stars with <num> rows and <num> columns, followed by `true`,
  - and then put that example/test in its own separate `cout` statement, such that the result of that test will be printed on its own line.

You should be able to compile `starbox_test.cpp` using the command:

```
g++ starbox_test.cpp starbox.cpp starline.cpp -o starbox_test
```

Submit your resulting files `starbox_test.cpp`, `starbox.cpp` and `starbox.h`.

Problem 3

Problem 3 part a

Consider: in the Week 10 Lecture 1 in-class examples `silly_length` and `count_vowels`, and on Homework 8 - Problem 4, we considered a `string` to be a "list" of `char`, and used recursion to "walk" through that string.

We can also use a `while` loop as another way to "walk" through a `string` -- indeed, we did, in the Week 12 Lecture 2 in-class example `vertical`.

Use the design recipe to design an alternate version of Homework 8 - Problem 4's `count_blanks` that expects a string, and returns the number of blanks in that string -- but this time, use a loop instead of recursion. (SO -- for *this* problem, you will not receive credit unless you attempt to use a loop instead of recursion, since loop practice is the goal here!)

(Hint: consider: for EACH character in a given string, how can you tell if it is a blank or not? And what should you do if it is? Write out pseudocode steps for your logic, then convert them into C++ statements.)

Problem 3 part b

NOTE - since this `count_blanks` is a function that returns a value and has no side-effects, `funct_play`'s `count_blanks_ck_expect.cpp` should be fine for your testing `main`, with no additional modification needed!

BUT **IF** you did **not** use `funct_play`, then do write a testing `main` for your `count_blanks` in a file `count_blanks_test.cpp` that:

- prints to the screen a message saying that you are about to test `count_blanks`, and that `true`s mean passed
- put `boolalpha` into the `cout` output stream, so that `bool` values are printed as `true` and `false`
- copy each example/test from your function's examples into its own separate `cout` statement, such that the

result of that test will be printed on its own line

You should be able to compile `count_blanks_test.cpp` using the command:

```
g++ count_blanks_test.cpp count_blanks.cpp -o count_blanks_test
```

Submit your resulting files `count_blanks.cpp`, `count_blanks.h`, and either `count_blanks_ck_expect.cpp` or `count_blanks_test.cpp`.

Problem 4

Problem 4 part a

Recall that, in Homework 7, Problem 5, you wrote a function `get_worth` that expects a character representing a coin:

- 'Q' or 'q' -- quarter
- 'D' or 'd' -- dime
- 'N' or 'n' -- nickel
- 'C' or 'c' or 'P' or 'p' -- cent/penny

...and it returns the decimal worth of that coin.

Also recall the `string` class method `at`, that expects the position of a desired character in that string, and returns the `char` at that position in the string (remembering, also, that it considers the position of the first character in the string to be position **0**, not 1).

Think about this -- do you see that you could use a count-controlled loop with this `at` method to do something with each `char` within a string? (Hint: consider function `vertical`, that uses a count-controlled loop and the `at` method to "grab" each character in a string and output it on its own line (thus outputting it in a "vertical" style).

Likewise, consider function `sum_ints`, that shows how you can use a count-controlled loop to sum all of the integers from 1 to a given integer.

NOW consider: what if you had a string whose characters were coin values? For example:

```
"qDnNCdQpN"
```

Using the design recipe, write a function `string_worth` that expects a string of coin characters, and returns the sum of the decimal worths of the coin characters in that string. For example,

```
string_worth("qDnNCdQpN") == 0.25 + .10 + .05 + .05 + .01 + .10 + .25 + .01 + .05
string_worth("Qn") == 0.30
```

For full credit, `string_worth` must also:

- appropriately call and use `get_worth`
- use a count-controlled loop

Problem 4 part b

NOTE - since this `string_worth` is a function that returns a value and has no side-effects, `funct_play`'s `string_worth_ck_expect.cpp` should be fine for your testing `main`, with no additional modification needed!

BUT **IF** you did **not** use `funct_play`, then do write a testing `main` for your `string_worth` in a file `string_worth_test.cpp` that:

- prints a message saying that you are testing function `string_worth`, and that trues mean passed
- puts `boolalpha` into the `cout` output stream, so that `bool` values are printed as `true` and `false`
- copy each example/test from `string_worth`'s examples into its own separate `cout` statement, such that the result of that test will be printed on its own line

You should be able to compile `string_worth_test.cpp` using the command:

```
g++ string_worth_test.cpp string_worth.cpp get_worth.cpp -o string_worth_test
```

Submit your resulting files `string_worth.cpp`, `string_worth.h`, and either `string_worth_ck_expect.cpp` or `string_worth_test.cpp` .

Problem 5

Consider -- any of your functions from part a of Problems 1-4 might benefit from a lovely interactive front end!

Decide: for which of `encourage`, `starbox`, `count_blanks`, or `string_worth` would you like to have an interactive front end?

Create a `main` function in a file named `encourage_ask.cpp`, `starbox_ask.cpp`, `count_blanks_ask.cpp`, or `string_worth_ask.cpp`, depending on your choice, that provides an interactive front end for your chosen function (and gives you a little bit more practice using `cin`, along with some more practice with another local variable):

- it should interactively ask the user for the information needed by your chosen function,
- and then it calls your chosen function with the information entered by the user.
  - NOTE that for `encourage` or `starbox`, you just want their side-effects -- simply calling them will give the user the desired results.
  - BUT for `count_blanks` or `string_worth`, you'll need to print to the screen the value they return in a descriptive message.

ADDITIONAL NOTES:

- IF you choose `count_blanks`, you need a way to read in a whole LINE from the user, since the user needs a way to enter a string with blanks! The `iostream` provides a function `getline` that can do this:

```
string entered_line;
cout << "type a line, and then enter: " << endl;
getline(cin, entered_line);
```

  - WARNING: this behaves oddly if used AFTER `cin >>`, but you shouldn't need `cin >>` for `count_blanks_ask.cpp`, I think. ASK ME if you have issues with this!
- Feeling adventurous? Want to call more than one of these (for example, `starbox` AND `encourage`?) That's fine -- just include all function names used in your `main` function's file name, ending with `_ask`, so I'll know it is your Problem 5 program. ASK ME if you are not sure what I mean by this!

Remember when you compile your program to include the `.cpp` files for ALL files involved in that program!

Submit your resulting `.cpp` file.