CS 111 - Homework 11

Deadline

11:59 pm on **Friday, December 9, 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 `111hw11`).
- 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 111hw11
  ```

- 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 **11**
  - Make sure that `~st10/111submit` shows that it submitted **ALL** of your `.cpp` and `.h` files you were intending to submit!

Purpose

To design and implement C++ functions, including `main` functions, and including some involving arrays, `for`-loops, sentinel-controlled `while` loops, and "question"-controlled `while` loops.

(To encourage you to practice with file input/output before the Final Exam, and to possibly bolster your homework grade via some additional programming, there are also **three BONUS problems** below.)

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 11 SETUP

I did not hear complaints about having copies of `.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, so we'll try that again for this homework, also.

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 11 directory:

  ```
  cd 111hw11
  ```

- use the following command to copy these "lab exercise example solution" versions of `starline.h` and `starline.cpp` to your Homework 11 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-111hw11/starline*  .
  ```

- 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-111hw11`, also, so you can make copies of those for Homework 11 using the command:

  ```
  cp  ~st10/for-111hw11/get_worth*  .
  ```

- You will also want your function `string_worth` from Homework 10, Problem 4. JUST in case you are not confident in your version, there's an example solution version of this in `~st10/for-111hw11`, also, so you can make copies of those for Homework 11 using the command:

  ```
  cp  ~st10/for-111hw11/string_worth*  .
  ```

- Also, you will want your function `do_op` from Homework 7, Problem 6. JUST in case you are not

confident in your version, there's an example solution version of this in `~st10/for-111hw11`, also, so you can make copies of those for Homework 11 using the command:

```
cp  ~st10/for-111hw11/do_op*  .
```

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

  ```
  ls
  ```

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

Problem 1

As an array warm-up, write a `main` function in a file named `array_play.cpp`.

This `main` function's purpose is simply to allow you to try creating an example array and then do something with its contents.

Here are the requirements:

- decide on how big an array you would like -- any size **larger than 5** is acceptable.
  - This size should be written as a **named constant**.
- decide what type of value you would like for your array to contain.
- declare an array of that type and size, and initialize it to values of your choice.
- then write a `for` loop to do something to/with each value in your array. You can choose -- it can be as simple as simply printing each value in the array, or something more elaborate (you can modify each value, call some function of your choice for each value, etc.)
  - Whatever you choose, make sure it has some output/result printed to the screen.

Submit your resulting `array_play.cpp`.

Problem 2

Problem 2 part a

Develop a C++ function `how_many` that expects a desired string, an array of strings, and the array's size, and returns the number of times the desired string appears in that array. It is required to appropriately use a `for` loop.

(suggestion: during the design recipe, after making your specific examples/tests, develop pseudocode for how you figured out how many times your example string appeared in your example array -- how did you know when a match was found? how do you keep track of how many you've seen so far?)

- (hint: for this one there had better be more than one example call! What are the different cases you should check? But as an additional hint -- if you design it well, you can probably use the **same** example array for more than one example call. What should be different in those calls, then?)

Problem 2 part b

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

- prints a message saying that you are testing function `how_many`, and that trues mean passed
- puts `boolalpha` into the `cout` output stream, so that `bool` values are printed as `true` and `false`

- declares and initializes at least one appropriate array, using a named constant for its size
- copies each example/test from `how_many`'s examples into its own separate `cout` statement, such that the result of that test will be printed on its own line

Submit your files `how_many.h`, `how_many.cpp`, and `how_many_test.cpp`.

Problem 3

Problem 3 part a

Consider, again, function `starline`, that 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.

You could use this to create a kind of horizontal bar chart, calling it for each of a set of values. And what is an array but a set of values?

Write a C++ function `bar_chart` that expects an array of integers and its size, returns the number of rows in the resulting bar chart (that is, the array's size...!), and has the side-effect of printing to the screen a horizontal bar chart with the help of `starline`, printing a line of *'s the length of each array value. This function must appropriately call `starline` and it must use an appropriate `for` loop.

For example, for:

```
const int NUM_MSRS = 7;
int measures[NUM_MSRS] = {3, 1, 6, 2, 8, 4, 5};

bar_chart(measures, NUM_MSRS) == NUM_MSRS
```

...and has the side-effect of causing the following to be printed to the screen:

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

Problem 3 part b

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

- prints a message saying that you are testing function `bar_chart`
- puts `boolalpha` into the `cout` output stream, so that `bool` values are printed as `true` and `false`
- calls `bar_chart` at least 2 times, each time with different arrays of different sizes
- because `bar_chart` 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 rows of stars of lengths <list them>, 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.

Submit your resulting files `bar_chart.h`, `bar_chart.cpp`, and `bar_chart_test.cpp`.

Problem 4

You will practice writing a **sentinel-controlled** `while` **loop** in this problem.

Problem 4 IMPORTANT NOTE!!

FIRST: the C++ `iostream` library includes a function, `getline`, that lets you read everything the user types -- even blanks! -- up to but not including enter/return:

```
getline(cin, string_to_read_into);
```

(It expects an input stream, for our purposes `cin`, and a local `string` variable, and it has the side-effect of setting that local `string` variable to what the user types up to but not including enter/return, read in and treated as a `string`. It behaves a bit oddly when you mix it with `cin <<`, BUT when used just with other calls to `getline` it is quite reasonable.)

For our purposes here, note that if the user immediately types enter/return, an empty string -- `""` -- is read into that local `string` variable that is its second argument.

...back to YOUR Problem 4 task:

A user might like a version of Homework 10 Problem 4's `string_worth` that allows them to enter more than one string of coin characters. So...

...for practice writing a **sentinel-controlled-style** `while` **loop**, write a `main` function in a file named `count_change.cpp` whose purpose is to serve as a **repeated**-interactive front-end for the function `string_worth`.

That is, your `main` function will repeatedly prompt the user to enter a string of coin characters, and then use `string_worth` to determine the worth of the coins in the entered string and print to the screen a polite, readable message telling the user the worth of the coins in the entered coin-characters string, until the user indicates that he/she would like to stop by entering the special sentinel value.

For this program, the user entering an **empty** coin-string -- just typing enter/return when prompted -- would make for a quite reasonable sentinel value.

Here are **two additional requirements**:

- this program should allow the user to enter an empty coin character string by using the function `getline` to read in the entered coin-characters string
- **ALSO** keep a **running total** of the worth of all of the strings entered by the user, and after the user enters the special sentinel value, print a final polite, readable message telling the user the total worth of all of the entered coin-character strings they entered during that program run.

Make sure that this `main` function uses a properly structured sentinel-controlled-style `while` loop (as described in Week 14 - Lecture 2 and in the Week 14 Lab) in asking the user to enter the next coin-characters string, and then displaying total worth of the entered string. Also, be sure to include what the user needs to type in to quit as part of your interactive prompt to the user.

Test-run your `count_change` program on an appropriate variety of test input until you are convinced that it is working properly. (In this case, it is also wise to test it on a case in which the user immediately enters the sentinel value, the first time they are asked. What should happen in that case?)

Submit your resulting `count_change.cpp`.

Problem 5

Consider the `do_op` function from Homework 7 - Problem 6, that expects an operator expressed as a character and two numbers, and returns the result of performing the operator corresponding to that character to those two numbers.

To practice writing a **"question"-controlled style** `while` **loop**, write a `main` function in a file named `calc.cpp` whose purpose is to serve as a repeated-interactive front-end for the function `do_op`.

That is, your `main` function will repeatedly ask the user if they would like to do another computation, and if they answer yes, it prompts the user to enter a character and two numbers, and then prints to the screen, in an appropriate descriptive message, the value that `do_op` returns for those arguments. It will keep doing this until the user **doesn't** answer yes.

Make sure that this `main` function uses a properly structured "question-controlled" loop style as described in Week 14 - Lecture 2 and in the Week 14 Lab, and make sure that it appropriately calls `do_op`. You get to **decide** what will be considered as a "yes" answer; make sure it is clear to the user what they need to type to continue and to stop.

Test-run your `calc` program on an appropriate variety of test input until you are convinced that it is working properly. (In this case, it is also wise to test it on a case in which the user immediately answers no, the first time they are asked. What should happen in that case?)

Submit your resulting `calc.cpp`.

BONUS PROBLEMS

BONUS PROBLEM 1 - worth up to 10 POINTS

We will be discussing simple file input/output during Week 15. For some practice writing to a file...

BONUS Problem 1 IMPORTANT NOTE!!

You can use the `string` method `c_str` to generate a `char*` string FROM a `string` object:

your_string_variable_name.`c_str()`

...which turns out to be very useful when a `string` local variable contains the name of a file to be opened, since the input file stream's `open` method is one of the rare situations where an old-style C-string of type `char*` is required.

...back to YOUR BONUS Problem 1 task:

Write a **`main`** function in a file named `fill_file.cpp` whose purpose is to allow a person to interactively create a file and fill it with desired contents, line by line. It needs to meet the following requirements:

- It should ask the user to enter a desired file name, which should be read into a local variable of type `string`. (You will be writing to a file with this name.)
- Then it should interactively ask for **lines** of file content that should be written to a file with that name in the current directory.
  - You are expected to use function `getline` to read in the lines that are to be written to the file.
  - Remember, `getline` expects the input stream as its first argument -- so, here, reading from the user,

that will be `cin` -- and the string to be set to what is typed in as its second argument.

For this problem, you get to **choose** whether to use a `for`-loop (to ask for a set number of lines from the user), a sentinel-controlled loop style, or a "question"-controlled loop style. Don't forget to close your output file stream when you are done!

This being a `main` function, we don't write a separate tester for it! But do happen to use it to create at least two files whose names include `bonus-prob1` in their names, and that end in `.txt` (for example, `bonus-prob1-1.txt` and `bonus-prob1-2.txt`).

Submit your resulting `fill_file.cpp` and at least two resulting `.txt` files from running `fill_file`.

BONUS PROBLEM 2 - worth up to 25 POINTS

We will be discussing simple file input/output during Week 15. For some practice reading from a file...

Consider again Homework 8 - Problem 3's function `pig_lite`, which uses `is_vowel` and `first`, and which expects a word and returns a Pig-Latin-ish version of that word.

Recall that function `pig_lite` expects a word, and if it starts with a vowel, it returns a string that is that word with `-ay` added to its end -- otherwise, it returns a string that is that word with `-` and its first letter and `ay` added to its end (although you had the option of making a more-sophisticated version if you chose to).

Write a **`main`** function in a file named `piggifile.cpp` whose purpose is to ask the user to enter a desired file name (stored in a local variable of type `string`), and then it attempts to call `pig_lite` for each "word" in that file, outputting each resulting piggified "word" to the screen on its own line. (For our purposes, a "word" is *any* set of non-white-space characters separated by white space. So,

```
How are you?
```

has 3 words, "`How`" and "`are`" and "`you?`".)

For example, if a file `testy.txt` contains:

```
How are you
today, Sarah?
```

...then, if the user typed `testy.txt` as the file name when prompted by `piggifile`, then the following would be printed to the screen (assuming the default un-fancy version of `pig_lite` -- yours might be "nicer"):

```
How-Hay
are-ay
you-yay
today,-tay
Sarah?-Say
```

(Well, we did call it `pig_*lite*`... 8 -))

Create two input files, named `pig-test1.txt` and `pig-test2.txt`, that each contain at least 2 lines and at least 5 "words" each. (You may have additional input files if you would like, of course -- and note that you can use Bonus Problem 1's `fill_file` or `nano` or Notepad++ or TextWrangler, etc., to create these files.)

Test-run your `piggifile` program on at least these two input files until you are convinced that it is working properly.

Submit your resulting `piggifile.cpp`, `pig-test1.txt`, and `pig-test2.txt`, along with the `.cpp` and `.h` files for `is_vowel`, `first`, and `pig_lite` (since you might be using a "fancier"

pig_lite).

BONUS PROBLEM 3 - worth up to 25 POINTS

We will be discussing simple file input/output during Week 15. For some practice reading from a file...

Bonus Problem 3 part a

Consider bar_chart from Problem 3.

What if you had a file formatted as follows:

* its first line contains the number of values in the file,
* followed by exactly that many integers, one per line.

(You may assume, for this problem, that the file is ALWAYS formatted precisely as described above.)

Write a function file_chart that expects one string parameter, the name of a file, and returns nothing, but as a side effect it calls bar_chart for the values in this file, by:

* opening that file for reading
* reading the first line from the file
* declaring an array of int's of that size
* filling the array with the rest of the values read from that file (using an appropriate for loop)
* appropriately calling bar_chart with the result
* explicitly closing the input file stream when it is done

Bonus Problem 3 part b

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

- prints a message saying that you are testing function file_chart
- calls file_chart at least 2 times, each time with a different input file, each of which contains a different number of integers
  - you need to submit these input files along with file_chart_test.cpp so I can run your tests. So that ~st10/111submit will accept them, make sure their names end in .txt
  - you should create these input files yourself -- note that you can use Bonus Problem 1's fill_file or nano or Notepad++ or TextWrangler, etc., to create these files
  - make sure your example test input files each contain a first line giving how many integers are in the file, followed by that many integers, one per line
- because file_chart has a desired side-effect (and NO return value), for EACH of its examples/tests,
  - it should first print a message saying that what follows should a bar chart with rows of stars of lengths <list them>,
  - followed by the example/test call.

Submit your resulting files file_chart.h, file_chart.cpp, and file_chart_test.cpp.