CS 111 - Homework 9

Deadline

11:59 pm on **Friday, November 11, 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 `111hw9`).
- 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 111hw9
  ```

- 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 **9**
  - 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`.

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 with the command:

  `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).

Problem 1

NOTE: you are reading some code and answering two questions about it, and then you are writing an odd little `main` function here! The purpose is to make a point about the impact of calling a function in slightly different ways.

Recall the function `say_sound` from Week 11 Lecture 1 and 2. Along with this homework handout, I have posted a VARIATION of this, called `say_sound2`, that, like `say_sound`, expects an animal name and its desired animal sound, and has the side effect of printing that that animal says that sound to the screen -- BUT, to make a point, it ALSO returns something, in this case the LENGTH of the animal sound (how many characters are in the string representing that sound).

(That is,

```
say_sound2("cow", "oink") == 4
```

...AND has the side effect of printing:

```
The cow says, "oink".
```

... to the screen.

```
say_sound2("raccoon", "screech") == 7
```

...AND has the side effect of printing:

```
The raccoon says, "screech".
```

... to the screen.)

FIRST: to try to make sure the differences between `say_sound` and `say_sound2` are clear to you, create a file `problem1.txt`, start it off by including your name, and then answer the following questions as **Problem 1 part a** and **Problem 1 part b**, preceding each answer with the question part you are answering:

Problem 1 part a

Look at `say_sound`'s function header in `say_sound.h` (posted with the Week 11 Lecture 1 and 2 examples), and `say_sound2`'s function header in `say_sound2.h` (posted along with this homework handout).

Answer in `problem1.txt`: BESIDES the function name, what is the OTHER difference between these two headers?

Problem 1 part b

Now look at `say_sound`'s function body in `say_sound.cpp` (posted with the Week 11 Lecture 1 and 2 examples), and `say_sound2`'s function body in `say_sound2.cpp` (posted along with this homework handout).

Answer in `problem1.txt`: What is in `say_sound2`'s function body that is NOT in `say_sound`'s function body?

Problem 1 part c

True C++ fact: when you call a function, IF you don't CARE about the value a function being called returns, but JUST want its side-effects, you can **ignore** the returned value by simply writing that function like a statement -- for example:

```
say_sound2("cat", "meow");   // the returned 4 is just ignored here
            //     (but the The cat says, "meow"! will still get
            //     printed to the screen!)
```

What you are to do for Problem 1 part c:

First: Make copies of at least `say_sound2.cpp` and `say_sound2.h` in your current folder/working directory.

Then: use the design recipe to write a `main` function in a file named `say_sound2_play.cpp` whose purpose is to hopefully help you see some of the differences that can result from how you call a function. This experimenting-style `main` function should do the following (and I am pretty much giving you pseudocode here!):

- declare three local variables:
  - `desired_animal`, able to hold a string
  - `desired_sound`, able to hold a string
  - `say_sound2_result`, also able to hold an integer
- ask the user to enter a desired animal, and read what the user enters into `desired_animal`, and

  ask the user to enter a desired sound for that animal, and read what the user enters into `desired_sound`
- print to the screen "`calling say_sound2 BY ITSELF`" on its own line,
  - ...and then call `say_sound2` with `desired_animal` and `desired_sound` as its arguments, ended by a semicolon, similar to the way it is called above.
- print to the screen "`calling say_sound2 WITHIN cout`" on its own line,
  - ...and then write a `cout` statement that prints the result of calling `say_sound2` with `desired_animal` and `desired_sound` as its arguments.

  - [OBSERVE: how is the resulting output different in this case? THINK ABOUT: Why is it different?]
- print to the screen `"calling say_sound2 in assignment statement"` on its own line,
  - ...and then write an assignment statement setting `say_sound2_result` to the result of calling `say_sound2` with `desired_animal` and `desired_sound` as its arguments.
  - then print to the screen `"say_sound2_result is: "` followed by the value of `say_sound2_result`
- you can compile/link/load this program with the command:

  ```
  g++ say_sound2_play.cpp say_sound2.cpp -o say_sound2_play
  ```
- THINK ABOUT (you don't turn in answers to these, but you should think about them):
  - When should you call a (non-`void`) function all by itself, followed by a semicolon?
  - When should you call a (non-`void`) function within a `cout` statement?
  - When should you call a (non-`void`) function on the right-hand-side of an assignment statement?

Submit your resulting files `problem1.txt` and `say_sound2_play.cpp`.

Problem 2

Problem 2 part a

This next function calls some other functions, has some side-effects in addition to its return value, and happens to make use of a local variable.

Consider functions `say_sound` from Week 11 Lectures 1 and 2, and `say_sound2` from Problem 1. For Problem 2, you may use either one of these, your choice! Decide, and make sure you have copies of your choice's `.h` and `.cpp` files in your Homework 9 directory.

Then, also copy your function `salutation` (at least its files `salutation.h` and `salutation.cpp`) from Homework 8, Problem 2 part a. (You can also find an example solution for this on the course Moodle site, under "Selected solutions", in the "Homework 8 Example Solutions (in-progress)" folder.)

Then: use the design recipe to write a function `animal_letter` that expects a name, an animal, and an animal sound, that returns the length of that name, and has the side-effects of:

- using `salutation` to print a salutation for that name,
- then using `say_sound` or `say_sound2` to print a statement of what animal sound that animal makes.

For example, `animal_letter("Carla", "cow", "moo") == 5` and should have the side-effect of causing the following to be printed to the screen:

```
Dear Carla,
The cow says, "moo"!
```

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 `animal_letter.cpp`, remember to include the `.h` file for EACH function called by `animal_letter`.

- If you do this correctly, `animal_letter` should be a quite **short** function!
- Be careful with newlines here!

Problem 2 part b

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

- prints a message saying that you are testing function `animal_letter`
- puts `boolalpha` into the `cout` output stream, so that `bool` values are printed as `true` and `false`
- because `animal_letter` 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> sent a letter about <animal> and <animal sound>, 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 can compile/link/load this program with this command if you are using `say_sound2`:

```
g++ animal_letter_test.cpp animal_letter.cpp say_sound2.cpp salutation.cpp -o animal_letter_test
```

...and with this command if you are using `say_sound`:

```
g++ animal_letter_test.cpp animal_letter.cpp say_sound.cpp salutation.cpp -o animal_letter_test
```

Submit your resulting files `animal_letter_test.cpp`, `animal_letter.cpp`, and `animal_letter.h`.

Problem 3

As some more simple practice with `cin`, write a `main` function in a file named `animal_letter_ask.cpp` whose purpose is to serve as an interactive front-end for the function `animal_letter`. It should ask the user to:

- enter a name CONTAINING NO BLANKS that they would like to send an animal letter to,
- enter an animal name CONTAINING NO BLANKS, and
- enter an animal sound CONTAINING NO BLANKS,

...and then it calls `animal_letter` with whatever the user enters.

(Why the "containing no blanks" part? Because, as we discussed during the Week 11 Lab, `cin`, when reading into a `string` variable, considers any white space, such as a blank, to be the end of the input...!)

You can compile/link/load this program with this command if you are using `say_sound2`:

```
g++ animal_letter_ask.cpp animal_letter.cpp say_sound2.cpp salutation.cpp -o animal_letter_ask
```

...and with this command if you are using `say_sound`:

```
g++ animal_letter_ask.cpp animal_letter.cpp say_sound.cpp salutation.cpp -o animal_letter_ask
```

Submit your resulting `animal_sound_ask.cpp` -- you've already submitted `animal_sound.cpp` and `animal_sound.h` as part of Problem 2.

Problem 4

Consider 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.

First: copy at least `pig_lite.cpp`, `pig_lite.h`, `is_vowel.cpp`, `is_vowel.h`, `first.cpp`, and `first.h` to your current folder/working directory. (You can also find an example solution for `pig_lite` on the course Moodle site, under "Selected solutions", in the "Homework 8 Example Solutions (in-progress)" folder.)

An interactive front end for function `pig_lite` might be fun -- so, write a `main` function in a file named `piggify.cpp` whose purpose is to serve as an interactive front-end for the function `pig_lite`. It should ask the user to enter a word CONTAINING NO BLANKS that he/she would like to see a Pig-Latin-ish version of, and then prints to the screen the result of calling `pig_lite` with whatever the user enters.

(Why the "containing no blanks" part? Because `cin`, when reading into a `string` variable, considers any white space, such as a blank, to be the end of the input...! And, because `pig_lite` really is intended to be used for a single word.)

Now -- does this seem exactly like Problem 3? It is pretty similar -- BUT there is a **small but significant difference**, so be careful! Hint: if you actually did the thinking suggested in Problem 1, the difference should be apparent.

You can compile/link/load this program with the command:

```
g++ piggify.cpp pig_lite.cpp first.cpp is_vowel.cpp -o piggify
```

Submit your resulting `piggify.cpp`, `pig_lite.cpp`, and `pig_lite.h`. (I'm asking you to also submit `pig_lite` because there were some optional variations, and I want to be able to run your `piggify` program with *your* version of `pig_lite`.)