CS 112 - Homework 1

Deadline

11:59 pm on **Friday, September 2**

Purpose

To make sure you have read over the course syllabus, to reinforce some of the CS 112 course style standards, and to practice writing C++ programs using separate compilation, such that the resulting non-`main` function could be used in more than one `main` function.

How to submit

You will complete **Problems 1 and 2** on the course Canvas site (syllabus confirmation and reading questions, and some short-answer questions related to some CS 112 course style standards).

For **Problems 3 and 4**, you will create several `.cpp` and `.h` files on the CS50 IDE, and then submit those to the course Canvas site.

NOTE: While I list the separate files you need to submit for each problem below, I am going to set up Canvas to *also* accept `.zip` files.

That is,

- you can submit each `.cpp` and `.h` file to Canvas as you submitted each of your files for the Week 1 Lab Exercise.
- OR, if you prefer, you may compress your files to be submitted into a single `.zip` file and submit that `.zip` file to Canvas.

Problem 1 - 20 points

Problem 1 is correctly answering the "HW 1 - Problem 1 - required syllabus confirmation and reading questions" on the course Canvas site.

Problem 2 - 10 points

Problem 2 is correctly answering the "HW 1 - Problem 2 - Short-answer questions on CS 112 course style basics" on the course Canvas site.

Problem 3 - 35 points

The purpose of this problem is to help you get used to separate compilation by starting with a C++ program that consists of several functions contained in a single `.cpp` file, and "breaking" it up into a set of separately-compilable `.cpp` and `.h` files.

Remember that you have:

- `letter_match.h`, `letter_match.cpp`, and `letter_match_test.cpp` posted with the Week 1 Lab posted examples,
- a `main` function `.cpp` file template, a non-`main` function `.cpp` file template, and a non-`main` function `.h` file template, posted on the home page of both the public course web site and the course Canvas site.

Consider the provided source code for a C++ program whose functions are all within the file **`112prob3.cpp`**.

Within this file you will see source code for a function **`check_within`** and a function `main`.

"Break" this up into three files, **check_within.h**, **check_within.cpp**, and **check_within_test.cpp**, such that you could now separately compile check_within.cpp and check_within_test.cpp,

and, if you copied check_within.h and check_within.cpp into another folder, you could call check_within within any other function in that folder by simply including:

```
#include "check_within.h"
```

...within its #includes, and including check_within.cpp in its g++ command compiling the program involved.

Your resulting files should be able to be successfully compiled using:

```
g++ check_within.cpp check_within_test.cpp -o check_within_test
```

...and the resulting program should be able to be successfully run using:

```
./check_within_test
```

Submit your resulting files check_within.h, check_within.cpp, and check_within_test.cpp.

(And, by the way: you might find function check_within to be useful when you are testing a function that returns a double value, because == can be too "strong" a test when comparing two double-precision floating point values for equality! Feel free to make use of it throughout the semester as you find it useful.)

Problem 4 - 35 points

The purpose of this problem is to give you practice writing a separately-compilable function from the start, using the **design recipe steps** described during Week 1, which can be summarized as:

- Think about the data involved in the function.
- In the function's opening comment, give the function's signature (following CS 112 course style).
- In the function's opening comment, give the function's purpose statement (following CS 112 course style).
- Write the function header (followed by an empty body, for now)
- Back up in the function's opening comment, write at least two tests (and at least one test for each "category" of data involved), written (whenever possible) as bool expressions that should be true if specific calls of that function work as they should.
- Complete the function body
- Write a testing main function to print the results of actually running those tests.
- Compile the resulting program, see if the tests run.

To practice this, **choose any ONE of the suggestions below** for your function. (You are encouraged to try as many of them as you like, but I will only grade one of them, in the interests of time.)

Problem 4 - choice 1: five_letter_str

REMINDER: The C++ string class has a method length, that expects nothing and returns the length of the calling string.

Use the design recipe to design a separately-compilable function five_letter_str that expects a string, and returns whether it is a string of 5 characters precisely. (This might be convenient, for example, in making sure a user guess for a Wordle-style program is allowable...)

For example:

```
five_letter_str("moo") == false
```

Submit your resulting files `five_letter_str.h`, `five_letter_str.cpp`, and `five_letter_str_test.cpp` (that contains a `main` function printing to the screen the result of running function `five_letter_str`'s tests).

Problem 4 - choice 2: pos_in_bounds

REMINDER: The C++ `string` class has a method `length`, that expects nothing and returns the length of the calling string.

Use the design recipe to design a separately-compilable function **`pos_in_bounds`** that expects a string and what should be a position within that string, and returns `true` if that position is indeed a position in that string. (That is, verify that the position given is in [0, length of that string minus 1], since the position of the first character in a string is 0.)

For example:

```
pos_in_bounds("angle", 2) == true    // a string of length 5 has
                                     //     positions 0, 1, 2, 3, 4

pos_in_bounds("who", 3) == false     // a string of length 3 has
                                     //      positions 0, 1, 2
```

Submit your resulting files `pos_in_bounds.h`, `pos_in_bounds.cpp`, and `pos_in_bounds_test.cpp` (that contains a `main` function printing to the screen the result of running function `pos_in_bounds`'s tests).

Problem 4 - choice 3: dist_btwn

Use the design recipe to design a separately-compilable function **`dist_btwn`** that expects the *x* and *y* coordinates of two points in a 2-dimensional coordinate system, and returns the distance between those points.

For example:

```
dist_btwn(0.0, 0.0, 3.0, 4.0) == 5.0
```

...since the distance between the point (0.0, 0.0) and the point (3.0, 4.0) would be 5.0.

Submit your resulting files `dist_btwn.h`, `dist_btwn.cpp`, and `dist_btwn_test.cpp` (that contains a `main` function printing to the screen the result of running function `dist_btwn`'s tests).

(Hints:

- It is perfectly reasonable to look up the formula for the distance between two points.
- if you have trouble getting `dist_btwn`'s tests to pass, feel free to use `check_within` in `dist_btwn_test.cpp`'s `main` function! You can also write its testing `bool` expression seeing if the absolute value of the diference between the actual function call and expected function call is small enough.)

Problem 4 - choice 4: your choice!

Think of a non-`main` function that is "pure" -- that is, it expects one or more values and returns a result, but does not have any side-effects such as printing to the screen -- that you would like to write, and use the design recipe to design a separately-compilable version of that function.

Submit `.h` and `.cpp` files named based on your chosen function's name, and a file whose name is your function's name followed by `_test.cpp` (that contains a `main` function printing to the screen the result of running your function's tests).