CS 112 - Homework 2

Deadline

11:59 pm on **Friday, September 9**

Purpose

To let you demonstrate that you understand the difference between a function returning a value versus printing a value to the screen, to practice more with separate compilation in C++, to practice some C++ branching, and to practice some C++ repetition.

How to submit

You will complete **Problems 1 and 2** on the course Canvas site (short-answer questions on return vs. print, and short-answer questions reviewing `bool` expressions in C++).

For **Problems 3 onward**, 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
- 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 - 10 points

Problem 1 is correctly answering the "HW 2 - Problem 1 - Short-answer questions or `return` vs print" on the course Canvas site.

Problem 2 - 10 points

Problem 2 is correctly answering the "HW 2 - Problem 2 - Short-answer questions reviewing `bool` expressions" on the course Canvas site.

Problem 3

As a separate-compilation warm-up, use the design recipe to design the following function.

Consider the game Rock-Paper-Scissors. In a classic game of Rock-Paper-Scissors, a player must choose one of either Rock, Paper, or Scissors.

As part of a larger program to play this game, it is decided that a helper function `legal_move` would be convenient. Within the program, it is decided that:

- `'r'` represents a choice of Rock,
- `'p'` represents a choice of Paper, and
- `'s'` represents a choice of Scissors.

`legal_move` simply expects a character, and returns `true` if that character is exactly one of `'r'`, `'p'`, or `'s'`, and it returns `false` otherwise.

Submit your resulting files `legal_move.cpp`, `legal_move.h`, and `legal_move_test.cpp`.

Problem 4

Now for a function to stretch your branching muscles!

Again consider the game Rock-Paper-Scissors. And, again, as part of a larger program to play this game, it is decided that:

- `'r'` represents a choice of Rock,
- `'p'` represents a choice of Paper, and
- `'s'` represents a choice of Scissors.

The next desired helper function is `get_winner`. `get_winner` expects two characters, the first representing a first player's choice, and the second representing a second player's choice, and it returns:

- `-1` if either (or both) player choice characters is not one of `'r'`, `'p'`, or `'s'`,
- `0` if the players tied (made the same choice),
- `1` if the first player's choice is the winner, or
- `2` if the second player's choice is the winner.

Write your function to decide based on the following rules:

- Rock defeats Scissors
- Scissors defeats Paper
- Paper defeats Rock
- When both choices are the same, it's a tie

For full credit, use the function `legal_move` appropriately within `get_winner`.

Submit your resulting files `get_winner.cpp`, `get_winner.h`, and `get_winner_test.cpp`. (You will have already submitted `legal_move.cpp` and `legal_move.h` as part of Problem 3.)

We *might* use these functions `legal_move` and `get_winner` in Homework 3, once we have had a chance to review C++ interactive input more and discuss some klugy-basic C++ pseudo-random-number generation.

Problem 5

But, some repetition practice is needed in the meantime; and we'll throw in practice writing a function with side-effects as well.

Write a function `show_multiples` that expects an integer, has the **side-effect** of printing to the screen, each on its own line, 1 times the given integer and that product, then 2 times the given integer and that product, then 3 times the given integer and that product, and so on, through 12 times the given integer and that product, and returns the sum of all of those products. For full credit, you must use an appropriate C++ loop statement to do this.

For example,

```
show_multiples(4) == 312
```

...and has the side-effect of printing to the screen:

```
1 x 4 = 4
2 x 4 = 8
3 x 4 = 12
4 x 4 = 16
5 x 4 = 20
```

```
6 x 4 = 24
7 x 4 = 28
8 x 4 = 32
9 x 4 = 36
10 x 4 = 40
11 x 4 = 44
12 x 4 = 48
```

And,

```
show_multiples(6) == 468
```

...and has the side-effect of printing to the screen:

```
1 x 6 = 6
2 x 6 = 12
3 x 6 = 18
4 x 6 = 24
5 x 6 = 30
6 x 6 = 36
7 x 6 = 42
8 x 6 = 48
9 x 6 = 54
10 x 6 = 60
11 x 6 = 66
12 x 6 = 72
```

Submit your resulting files `show_multiples.cpp`, `show_multiples.h`, and `show_multiples_test.cpp`.

Problem 6

Consider: a function `is_letter` was included along with the Week 2 Lab Exercise.

Also, consider these FUN FACTS:

- The C++ `string` class is set up so you can use `+` to concatenate (append) two string objects together!
- But -- as long as at least ONE of its arguments is indeed truly of type `string`, the other argument can be a `char*` literal or even a `char`, and `+` still works!

Let's wrap up with a function that uses a previously-written function, and involves both branching and repetition.

Consider a game like Wheel of Fortune, where the players are given a phrase, and guess letters in that phrase, until they can guess the entire phrase.

As a very early step in a simple program playing such a guess-the-letters-in-a-phrase game, a helper function `hide_letters` is desired that expects a phrase and returns a starting-display version of that phrase with each letter replaced with an underscore (but all blanks and other characters left as they are).

For example:

```
hide_letters("Corner Curio Cabinet") == "______ _____ _______"
```

```
hide_letters("tic-tac-toe") == "___-___-___"
```

For full credit, use `is_letter` appropriately in `hide_letters`.

Submit your resulting files `hide_letters.cpp`, `hide_letters.h`, `hide_letters_test.cpp`, `is_letter.cpp`, and `is_letter.h`.