CS 131 - Homework 9

Deadline:

5:00 pm on Friday, November 12

How to submit:

When you are done with the following problems:

- make sure that your current working directory on nrs-labs is the one where your C++ function files for this homework are;
  - for example, you might need:

    ```
    cd 131hw9
    ```

    ...to change to you directory `131hw9`, and
  - then look at what files are there using `ls`
- then use `~st10/131submit` to submit your `.cpp` and `.h` files for homework number **9**
- make sure that `~st10/131submit` shows that it submitted your `.cpp` and `.h` files for all of your C++ functions and classes for this homework

Purpose:

To practice using arrays, count-controlled loops using `while` and/or `for` statements, and `cout`

Important notes:

- Each student should work **individually** on this homework.
- You are still expected to follow the Design Recipe for all functions that you write.
  - Remember, you will receive **significant** credit for the signature, purpose, header, and examples portions of your functions.
  - (but remember to use **C++ types** in signatures for C++ functions)
  - (and, use == or < for your C++ example expressions -- for example,

    ```
    my_funct(3) == 27
    abs(my_dbl(4.7) - 100.43) < 0.01
    ```

    ...and note that these example tests are **expressions** rather than C++ statements, so do NOT end them with a semicolon!)
  - Typically you'll get at least half-credit for a correct signature, purpose, header, and examples/, 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).
- Be especially careful to include at least one specific example/check-expect for each "kind"/category of data, and (when appropriate) for boundaries between data. You can lose credit for not doing so.
- Remember that the C++ `cmath` library, included by the course C++ tools by default, includes

such goodies as an absolute value function (`abs`), `sqrt`, `pow`, and more.

The Problems:

Problem 0

Create, protect, and change to a directory `131hw9` -- type the following from your home directory on nrs-labs:

```
[you1@nrs-labs ~]$ mkdir 131hw9

[you1@nrs-labs ~]$ chmod 700 131hw9

[you1@nrs-labs ~]$ cd 131hw9
```

(If you log out and come back later, remember to `cd 131hw9` each time to return to *this* directory!)

Problem 1

We talked about the concept of **pseudocode** in the Week 11 Lab. Here is another example of pseudocode, the logic needed for a function that expects an array of numbers and its size, and produces the smallest element in that array:

```
set the smallest seen so far to be the first element
while there are more array elements to check
    if the latest array value is less than the smallest
    seen so far,
        make that the new smallest seen so far
    increment my count
return smallest seen so far
```

Using the design recipe (as always!), now use `funct_play2` to develop a C++ function `get_smallest` that indeed expects an array of numbers and its size, and produces the smallest element in that array.

As in the Week 11 Lab array-function example `sum_array`:

* for the specific examples step of the design recipe (using `funct_play2`), do include declarations and initializations for an example array and its size before your specific example call(s) of `get_smallest`; [style note: give this example array and its size **different** names than your parameter array and its size]
* after typing in the function using `funct_play2`, edit the `get_smallest_ck_expect.cpp` file to declare the constant for the example array's size and create and populate the example array(s) as local variables before all of the `cout` statements, and either edit out or delete the `cout`'s containing those declarations

Submit your files `get_smallest.h`, `get_smallest.cpp`, and `get_smallest_ck_expect.cpp`.

Problem 2

Use `funct_play2` to develop a C++ function `how_many` that expects a desired string, an array of strings, and the array's size, and produces the number of time the desired string appears in that array.

(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?)

As in the Week 11 Lab array-function example `sum_array`:

* for the specific examples step of the design recipe (using `funct_play2`), do include declarations and initializations for an example array and its size before your specific example calls of `how_many` (hint: for this one there had better be more than one example call!)
* after typing in the function using funct_play2, edit the `how_many_ck_expect.cpp` file to declare the constant for the example array's size and create and populate the example array(s) as local variables before all of the `cout` statements, and either edit out or delete the `cout`'s containing those declarations

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

Problem 3

To get some practice with `cout`...

Remember that the `string` class has a method `length` -- it expects no arguments, and produces how many characters are in that string.

Now I will also note that + can be used with two `string` instances to concatenate, or combine, those two strings -- that is, using the `string` constructor method,

```
(string("be") + string("bop") == string("bebop"))
```

Using the design recipe, now use `funct_play2` to develop a C++ function `greeter` that expects a name, has the side-effect of printing a greeting of your choice that includes that name to the screen (ending with a newline), and produces the resulting length of the greeting-string including that name.

(Note that, when writing out its examples, you should also describe the side-effects each example call should have -- and then delete those `cout`'s from the resulting `greeter_ck_expect.cpp`.)

Submit your files `greeter.h`, `greeter.cpp`, and `greeter_ck_expect.cpp`.

Problem 4

Consider our `rhino` class and the function `is_safe_to_feed` from Homework 7.

Let's write a simple function using `cout` -- using the design recipe, use `funct_play2` to develop a C++ function `show_rhino` that expects a `rhino`, has the side-effect of printing to the screen the `rhino`'s data field values in the following format:

```
weight:        this rhino's weight
color:         this rhino's color
irritability:  this rhino's irritability index
```

(ending with a newline), and produces whether it is safe to feed this rhino right now. For full credit, appropriately use the function `is_safe_to_feed` in `show_rhino`.

That is, the call:

```
show_rhino(rhino(3000, "grey", 3)) == true
```

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

```
weight:        3000
color:         grey
irritability: 3
```

(Note that, when writing out its examples, you should also describe the side-effects each example call should have -- and then delete those `cout`'s from the resulting `show_rhino.cpp`.)

(Here are a few tips: don't forget that this function uses both the `rhino` class and the function `is_safe_to_feed` -- and copies of these need to be in the same directory as `show_rhino`.)

Submit your files `show_rhino.h`, `show_rhino.cpp`, and `show_rhino_ck_expect.cpp`.

Problem 5

You can have an array of class instances as well as an array of `int` or of `string` or of `double`. Consider an array of `boa` instances. You can declare an array of boas using something like:

```
const int NUM_BOAS = 3;

boa boa_bunch[NUM_BOAS] = {boa("green", 15, "pizza"),
                           boa("red", 10, "pie"),
                           boa("gray", 2, "pita")};
```

Since an element of this array is a `boa`, you can call methods on that `boa` instance in the usual way -- it just looks a little unusual since that instance is an array reference:

```
boa_bunch[0].get_length() == 15
```

...is a `true` expression, for example.

Now consider the `taxi` class and the function `fillup_cost` from Homework 7.

If you had an array of `taxi`'s, you might be interested in knowing how much it would cost to fill up all of their tanks with gasoline.

Using the design recipe, now use `funct_play2` to develop a C++ function `fillup_all_cost` that expects an array of `taxi`, its size, and a price per gallon of gasoline, and produces how much it would cost to fill up all of those taxis at that gasoline price. For full credit, appropriately use the function `fillup_cost` in `fillup_all_cost`.

(Here are a few tips: don't forget that this function uses both the `taxi` class and the function `fillup_cost` -- and copies of these need to be in the same directory as `fillup_all_cost`.)

Submit your files `fillup_all_cost.h`, `fillup_all_cost.cpp`, and `fillup_all_cost_ck_expect.cpp`.

Problem 6

Remember that the `string` class has a method `length` -- it expects no arguments, and produces how many characters are in that string.

Using the design recipe, now use `funct_play2` to develop a C++ function `show_in_lights` that expects a string, has the side-effect of

* printing a line of `*`'s the same length of that string on one line,
* then printing the string on the next line,
* and then printing a line of `*`'s the same length of that string on the next line, ending with a newline,

...and produces the length of that string.

(Note that, when writing out its examples, you should also describe the side-effects each example call should have -- and then delete those `cout`'s from the resulting `show_in_lights_ck_expect.cpp`.)

Hints:

* this is an example where you'd like a `cout` without an `endl` -- to get several things to print on the same line, deliberately;
* ...but when something needs to be done just once, it doesn't go IN a loop, but outside of it

Submit your files `show_in_lights.h`, `show_in_lights.cpp`, and `show_in_lights_ck_expect.cpp`.