CS 131 - Homework 10

Deadline:

5:00 pm on Friday, December 3

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 131hw10
    ```

    ...to change to you directory `131hw10`, and
  - then look at what files are there using `ls`
- then use `~st10/131submit` to submit your `.cpp` and `.h` files for homework number **10**
- 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 writing `void` functions, writing `main` functions, using `cin`, and writing sentinel-controlled and other kinds of loops

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 non-`void` function -- 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.
  - and now we also know that the C++ `iostream` library includes such goodies as `cout`, `cin`, `boolalpha`, and more.

The Problems:

Problem 0

Create, protect, and change to a directory `131hw10` -- type the following from your home directory on nrs-labs:

```
[you1@nrs-labs ~]$ mkdir 131hw10
[you1@nrs-labs ~]$ chmod 700 131hw10
[you1@nrs-labs ~]$ cd 131hw10
```

(If you log out and come back later, remember to `cd 131hw10` each time to return to *this* directory!)

Problem 1

Using `funct_play2`, develop a C++ function `line_of_X` that expects a desired number of X's, and doesn't produce anything, but has the side-effect of printing to the screen that many X's, followed by a newline character.

That is, the call:

```
line_of_X(3);
```

...would return nothing, but would have the side-effect of causing the following to be printed to the screen:

```
XXX
```

(remember that it SHOULD output a newline at the end of those 3 X's.)

Remember, though, that because `line_of_X` is a `void` function, the `_ck_expect` function `funct_play2` produces is not usable without extensive modification. So, for this problem, write your own small `main` function to test this `void` function.

Using `nano` (or your favorite text editor), write a `main` function in a file named `line_of_X_test.cpp`, using the `main` function template (`main_template.txt`) available from the public course web page as a template. (Helpful hint: make sure you have a `#include` line for `line_of_X.h` at the beginning of this `main` function)

This `main` function is required to meet the following requirements -- it must:

* ...call `line_of_X` at least 3 times, each time with a different argument value
* ...precede each `line_of_X` call with a `cout` saying how many X's ought to be seen on the next line
  * (Why? so someone looking at `line_of_X_test`'s results on-screen can tell if the results are reasonable! 8-))

(Remember: to compile and link the functions making up this small program `line_of_X_test`, you need to execute an appropriate `g++` command at the nrs-labs prompt -- remember to include the `.cpp` or `.o` files for both source code files involved, `line_of_X.cpp` and `line_of_X_test.cpp`)

Submit your files `line_of_X.h`, `line_of_X.cpp`, and `line_of_X_test.cpp`.

Problem 2

Using `funct_play2`, develop a C++ function `box_of_X` that expects desired number of rows and a desired number of X's per row, and doesn't produce anything, but has the side-effect of printing to the screen that many rows of X's, each with that many X's per row, ending up with printing a newline character. This function must appropriately call `line_of_X`.

That is, the call:

```
box_of_X(3, 5);
```

...would return nothing, but would have the side-effect of causing the following to be printed to the screen:

```
XXXXX
XXXXX
XXXXX
```

(remember that it SHOULD output a newline at the end of the "box".)

Again, because `box_of_X` is a void function, you are expected to write a own small `main` function to test this `void` function.

Using `nano` (or your favorite text editor), write a `main` function in a file named `box_of_X_test.cpp`, using the `main` function template (`main_template.txt`) available from the public course web page as a template. (Helpful hint: make sure you have a `#include` line for `box_of_X.h` at the beginning of this `main` function)

`box_of_X_test.cpp` is required to meet the following requirements -- it must:

* call `box_of_X` at least 2 times, each time with different argument values
* precede each `box_of_X` call with a `cout` saying how many rows of how many X's ought to be seen starting on the next line (so someone looking at just the results on-screen can tell if they are reasonable)

(Remember: to compile and link the functions making up this small program `box_of_X_test`, you need to execute an appropriate `g++` command at the nrs-labs prompt -- remember to include the `.cpp` or `.o` files for all three source code files involved, `box_of_X_test.cpp`, `box_of_X.cpp`, *and* `line_of_X.cpp`)

Submit your files `box_of_X.h`, `box_of_X.cpp`, and `box_of_X_test.cpp`.

Problem 3

Using `funct_play2`, develop a C++ function `triangle` that expects the number of rows desired for a triangle, and doesn't produce anything, but has the side-effect of printing to the screen a "triangle" of X's that many rows tall, with one X in the first row, two X's in the second row, and so on, ending up with the newline character. This function must appropriately call `line_of_X`.

That is, the call:

```
triangle(5);
```

...would return nothing, but would have the side-effect of causing the following to be printed to the screen:

```
X
XX
XXX
XXXX
XXXXX
```

(remember that it SHOULD output a newline at the end of the "triangle".)

Again, because `triangle` is a void function, you are expected to write a own small `main` function to test this `void` function.

Using `nano` (or your favorite text editor), write a `main` function in a file named `triangle_test.cpp`, using the `main` function template (`main_template.txt`) available from the public course web page as a template.

`triangle_test.cpp` is required to meet the following requirements -- it must:

* call `triangle` at least 2 times, each time with different argument values
* precede each `triangle` call with a `cout` saying how tall a triangle ought to be seen starting on the next line (so someone looking at just the results on-screen can tell if they are reasonable)

(Remember: to compile and link the functions making up this small program `triangle_test`, you need to execute an appropriate `g++` command at the nrs-labs prompt.)

Submit your files `triangle.h`, `triangle.cpp`, and `triangle_test.cpp`.

Problem 4

Consider `line_of_X` yet again. 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?

Using `funct_play2`, develop a C++ function `bar_chart` that expects an array of integers and its size, and doesn't produce anything, but has the side-effect of printing to the screen a horizontal bar chart with the help of `line_of_X`, printing a line of X's the length of each array value. This function must appropriately call `line_of_X`.

That is, the fragment:

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

...would return nothing, but would have the side-effect of causing the following to be printed to the screen:

```
XXX
X
XXXXXX
XX
XXXXXXXX
```

```
XXXX
XXXXX
```

For this `void` function, you are expected to write a own small `main` function to test this `void` function.

Using `nano` (or your favorite text editor), write a `main` function in a file named `bar_chart_test.cpp`, using the `main` function template (`main_template.txt`) available from the public course web page as a template.

`bar_chart_test.cpp` is required to meet the following requirements -- it must:

* call `bar_chart` at least 2 times, each time with different arrays of different sizes
* precede each `bar_chart` call with a `cout` describing what the user should see (so someone looking at just the results on-screen can tell if they are reasonable)

(Remember: to compile and link the functions making up this small program `bar_chart_test`, you need to execute an appropriate `g++` command at the nrs-labs prompt.)

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

Problem 5

This problem's very small function will allow you to have some interactive input practice and will also be usable later in this assignment.

Using `funct_play2`, develop a C++ function `request_weight` that expects nothing, and produces a numeric weight interactively entered in by the user. (That is, it asks the user to type in a desired numeric weight, reads in what the user types, and returns that value.) To make this function convenient for use in later problems, in your request to the user, ask him/her to enter a negative weight (less than 0.0) if he/she would like to quit.

This function is not a `void` function, although its `_ck_expect` does require editing to be a suitable tester. So, *either* edit `request_weight_ck_expect.cpp` or write a `request_weight_test.cpp` that EITHER:

* asks the user to enter a specific weight when prompted
* then prints to the screen whether than `request_weight()` call returned that value

OR:

* prints a message to the screen noting that whatever the user types in when prompted SHOULD then printed to the screen
* then prints to the screen the result of calling `request_weight()`

(By the way: you CAN run `request_weight` using `expr_play`, and at the end of `funct_play2` and `funct_compile`! Interactive input and output sometimes cause odd-looking results, but they don't freak these tools out like `void` functions do...)

Submit your files `request_weight.h`, `request_weight.cpp`, and either `request_weight_ck_expect.cpp` or `request_weight_test.cpp`. (It's OK if you submit all 4...)

Problem 6

Since, in a properly-structured sentinel-controlled loop, you write statements "obtaining" the

next piece of data both before and inside at the end of a while-loop, can you see how `request_weight` could be used as part of such logic?

Using `funct_play2`, use `request_weight` in a function `sum_weights` that expects nothing, but produces a sum of weights -- it uses `request_weight` with a sentinel-controlled loop to interactively ask for weights from the user until the user enters a negative weight, and only then does it return the sum of those weights (all but the negative sentinel value, of course!)

For full credit, you are required to use a properly-structured sentinel-controlled loop and appropriate calls to `request_weight` in your function.

Either edit `sum_weights_ck_expect.cpp` or write a `sum_weights_test.cpp` that EITHER:

* asks the user to enter specific weights when prompted
* then prints to the screen whether that `sum_weights()` call returned the expected sum of those specific weights

OR:

* prints a message to the screen noting that the sum of whatever the user types in when prompted SHOULD then printed to the screen
* then prints to the screen the result of calling `sum_weights()`

Submit your files `sum_weights.h`, `sum_weights.cpp`, and either `sum_weights_ck_expect.cpp` or `sum_weights_test.cpp`. (It's OK if you submit all 4...)

Problem 7

For some more sentinel-controlled loop practice, write a `main` function in a file `many_lines.cpp` that provides an interactive "interface" for function `line_of_X` from Problem 1 using a sentinel-controlled loop.

That is, your `main` function will prompt the user for desired numbers of X's, and then use `line_of_X` to display lines with that many X's, until the user indicates that he/she would like to stop.

Make sure that this `main` function uses a properly-structured sentinel-controlled loop in asking the user to enter how many X's he/she would like to see in the next line, using `line_of_X` appropriately to then display a line of that many X's. What would be an appropriate sentinel value for this situation? Decide, and include what the user needs to type in to quit as part of your interactive prompt to the user. (You may certainly also write a small auxiliary function to handle this input, as `request_weight` did for `sum_weights`, if you would like.)

At this point in this course, we don't have a good way to write testers for `main` functions themselves, so test-run your `many_lines` program until you are satisfied that it works properly, and submit your file `many_lines.cpp`.

Problem 8

To practice writing a loop that takes a "question" approach (as discussed in the Week 13 Labs), write a `main` function in a file `many_boxes.cpp` that provides an interactive "interface" for function `box_of_X` from Problem 2 using a loop that takes such a "question" approach, each time asking the user if he/she would like to enter data for another "box" of X's before asking for

the necessary box-related information and calling `box_of_X` appropriately. When they answer negatively, your program should end.

Make sure that this `main` function uses a properly-structured "question" approach loop in asking the user whether he/she would like to request another "box" of X's. (You may certainly also write a small auxiliary function to handle this answer input, as `request_weight` did for `sum_weights`, if you would like.)

Test-run your `many_boxes` program until you are satisfied that it works properly, and submit your file `many_boxes.cpp`.

Problem 9

And, of course, you could have a count-controlled "interface" for some function, also -- it could ask the user how many "somethings" they want, and then that many times it will ask the user for the necessary information and do it.

To practice this approach, then, write a `main` function in a file `many_tris.cpp` that provides an interactive "interface" for function `triangle` from Problem 3 using a loop that takes such a count-controlled approach. That is, it should first ask the user how many triangles he/she would like, and then, that many times, it will ask for the next triangle size and using `triangle` to display a triangle of that size.

Make sure that this `main` function uses a properly-structured count-controlled loop.

Test-run your `many_tris` program until you are satisfied that it works properly, and submit your file `many_tris.cpp`.