CS 131 - Homework 11

Deadline:

5:00 pm on Friday, December 10

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 131hw11
    ```

    ...to change to you directory `131hw11`, and
  - then look at what files are there using `ls`
- then use `~st10/131submit` to submit your `.cpp` and `.h` files for homework number **11**
- 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 with pointers, file input/output, dynamic array allocation, and pass-by-reference.

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,
  - and that the C++ `fstream` library can be used for file input/output

The Problems:

Problem 0

Create, protect, and change to a directory `131hw11` -- type the following from your home directory on nrs-labs:

```
[you1@nrs-labs ~]$ mkdir 131hw11
[you1@nrs-labs ~]$ chmod 700 131hw11
[you1@nrs-labs ~]$ cd 131hw11
```

(If you log out and come back later, remember to `cd 131hw11` each time to return to *this* directory!)

Problem 1

For this and several other problems on this homework, you will need to have the `rhino` class available; you can find versions of `rhino.h` and `rhino.cpp` from the Homework 8 example solutions link on the course Moodle site, if you need or would like to. You will need to have copies of `rhino.h` and `rhino.cpp` in your `131hw11` directory.

For a little pointer practice, write a function `show_rhino` that expects a **pointer** to a rhino, and it produces nothing, but it has the side-effect of printing that rhino's 3 data fields' values to the screen in a pleasing format.

Either modify `show_rhino_ck_expect.cpp` or write a `main` function in `show_rhino_test.cpp` to test this function on at least 2 different `rhino` instances of your choice.

Submit your resulting `.cpp` and `.h` files.

Problem 2

Before we continue with rhinos -- let's take a small break. Consider `bar_chart` from Homework 10, Problem 4.

What if you had a file formatted as follows:

* its first line contains the number of values in the file,
* followed by exactly that many integers, one per line.

(You may assume, for this problem, that the file is ALWAYS formatted precisely as described above.)

Write a function `file_chart` that expects the name of a file, and produces nothing, but as a side effect it calls `bar_chart` for the values in this file, by:

* opening that file for reading (hint: use the `c_str` method as seen in Week 14 Lab example

`file_io_ex.cpp`)

* reading the first line from the file
* dynamically allocating an array of `int`'s of that size
* filling the array with the rest of the values from that file
* appropriately calling `bar_chart` with the result
* don't forget to then free that dynamically-allocated array's memory! (And for good measure and for practice, set the pointer involved to `NULL` after doing so.)

To test this, write a small `main` function that appropriately calls `file_chart` at least once.

Submit your resulting `.cpp` and `.h` files, as well as at least one example data file (that meets the stated requirements) whose name ends in `.txt` (or else `~st10/131submit` won't "see" it!)

Problem 3

Now, back to rhinos -- we are going to want to call this for a number of arrays of rhinos in an upcoming problem. It would be useful to have a function to streamline this. Write a function `show_all_rhinos` that expects an array of rhinos and its size, and it produces nothing, but it does call `show_rhino` for each rhino in that array.

Be careful -- how can you give `show_rhino` the *address* of each rhino in that array, since it expects a *pointer* to a `rhino`?

To test this, write a small `main` function that meets the following specifications:

* it dynamically-allocates an array of rhinos of size 3 or more
* it calls `show_all_rhinos` with that array and its size
* don't forget to then free that dynamically-allocated array's memory! (And for good measure and for practice, set the pointer involved to `NULL` after doing so.)

Submit your resulting `.cpp` and `.h` files.

Problem 4

Write a function `calm_rhinos` that expects an array of rhinos, its size, and the desired amount to calm all of them, and it produces nothing, but it does calm each of the rhinos in that array by that amount. (Remember that the `rhino` methods includes a `calm` method...!)

To test this, write a small `main` function that meets the following specifications:

* it dynamically-allocates an array of rhinos of size 3 or more
* it calls `show_all_rhinos` to display the original version of these rhinos
* it calls `calm_rhinos` appropriately
* it calls `show_all_rhinos` again to display the calmed version of these rhinos
* don't forget to then free that dynamically-allocated array's memory! (And for good measure and for practice, set the pointer involved to `NULL` after doing so.)

Problem 5

Now, assume that an input file has been set up that is organized as follows:

* on the first line, it has the number of rhinos whose information is included in the file;
* on each 3 lines after that, it has a rhino's weight, color, and irritability index, for precisely that many rhinos

(For this particular problem, you are NOT responsible for what happens if the file is NOT set up in this way!)

Write a `main` function `calm_file_rhinos.cpp` that meets the following specifications:

* it asks the user to enter the name of a properly-formed file of rhino data (as described above), and sets up an input file stream and opens up this file for reading
* it reads the number of rhinos from the first line of the input file, and dynamically allocates an array of that many rhinos
* then it reads in the rhino information from the file, setting the array elements to be rhinos with that information
* it asks the user how much to calm this set of rhinos, and then it uses `calm_rhinos` to calm the rhinos in the array by that amount
* it asks the user for the name of the file to write the calmed rhino information to, and sets up an output file stream and opens this file for writing
* then it writes the number of rhinos to the first line of this file,

  followed by the data for each of the now-calmed rhinos, one piece of rhino data per line.
* don't forget to properly free the dynamically-allocated array of rhinos when you are done! (And for good measure and for practice, set the pointer involved to `NULL` after doing so.)

Submit your resulting `calm_file_rhinos.cpp`, as well as at least one example rhino data file (that meets the stated requirements) named `my_rhinos.txt`

Problem 6

You need a little pass-by-reference practice. Using `funct_play2` (but writing your own `main` to test it), write a function `accelerate` that expects a single pass-by-reference parameter, representing a speed, and it increases the corresponding argument's value by 10%.

That is, if you did:

```
double curr_speed = 48;
accelerate(curr_speed);
```

...then after these two statements, the following expression should be true:

```
curr_speed == 52.8
```

(although you might need: `abs(curr_speed - 52.8) < .001 `)

Either modify `accelerate_ck_expect.cpp` or write a `main` function in `accelerate_test.cpp` that performs at least the above actions and then outputs the result of the expression shown (being sure to use `boolalpha` in `cout` before trying to print out the results of the expression comparing `curr_speed` to its expected value).

Submit your resulting `.cpp` and `.h` files.