CS 131 - Homework 8

Deadline:
5:00 pm on Friday, November 5

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 131hw8
    ...
    then look at what files are there using ls
  – then use ~st10/131submit to submit your .cpp and .h files for homework number 8
• then use ~st10/131submit to submit your .cpp and .h files for homework number 8
• 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 some more using C++ if statements, and to add zero-argument constructors, modifier methods, and other methods to C++ classes

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++ \texttt{cmath} library, included by the course C++ tools by default, includes such goodies as an absolute value function (\texttt{abs}), \texttt{sqrt}, \texttt{pow}, and more.

\textbf{The Problems:}

\textit{Problem 0}

Create, protect, and change to a directory 131hw8 -- type the following from your home directory on nrs-labs:

\begin{verbatim}
[you1@nrs-labs ~]$ mkdir 131hw8
[you1@nrs-labs ~]$ chmod 700 131hw8
[you1@nrs-labs ~]$ cd 131hw8
\end{verbatim}

(If you log out and come back later, remember to \texttt{cd 131hw8} each time to return to this directory!)

\textit{Problem 1}

Consider: a character '+' cannot be used to actually add two numbers together in C++ -- but if you were given that character, and two numbers, you could write logic that would see if the character was '+', and if that is so, then add those numbers together.

So, for a function that will require use of a C++ branching statement: use \texttt{funct\_play2} to develop a C++ function \texttt{do\_op} that expects an operator expressed as a character and two numbers, and produces the result of performing the specified operation on those two numbers. These are further requirements for this function:

* it should produce a value of 0.0 if it is called with an operator character besides '+', '-', '*', or '/'

* it should also produce a value of 0.0 if someone attempts to divide by 0

Submit your resulting \texttt{do\_op.cpp}, \texttt{do\_op.h}, and \texttt{do\_op\_ck\_expect.cpp} files.

\textit{Problem 2}

Now for some practice with other kinds of methods in classes: overloaded methods, zero-argument constructors, modifier methods, and "other" methods.

Consider the \texttt{rhino} class from Homework 7.

We added an overloaded zero-argument constructor and a modifier method for each data field to class \texttt{boa} during class. Now add an overloaded zero-argument constructor and a modifier method for each data field to class \texttt{rhino}.

Also add an "other" method, \texttt{calm}, that:

* expects an integer giving how much you have calmed the calling rhino;

* has the side-effect of reducing the rhino's irritability index by the amount it has been calmed EXCEPT not reducing it to less than 0 (don't allow the resulting irritability index to be less than 0);

* produces/returns the new irritability value for the rhino.
To test these new methods, modify `rhino_test` as follows:

* add a declaration using the new 0-argument constructor;
* create 3 `bool` variables to hold results of "sets" of tests (as is done in the posted example `boa_test.cpp`);
* modify the current `return` statement to instead set one of these `bool` variables, and to also test if the rhino created by the 0-argument constructor also has the data fields expected;
* call each of the modifiers at least once;
* set yet another `bool` variable to the result of testing if the rhino(s) modified by the modifiers has the data field values now expected;
* call `calm` at least twice, on two different rhinos, calming one less than its current irritability index, and calming the other more than its irritability index;
* set the 3rd `bool` variable to the result of testing if those rhinos' irritability indexes are as they should be after the `calm` calls; and
* return the result of the logical `and` of the three `bool` variables.

Remember that you can use nano to modify these rhino files, and that you can use `funct_compile` to recompile the modified `rhino_test`.

Submit your files `rhino.h`, `rhino.cpp`, `rhino_test.cpp`, and `rhino_test_ck_expect.cpp`.

**Problem 3**

Now consider the `taxi` class from Homework 7.

Add an overloaded zero-argument constructor and a modifier method for each data field to class `taxi`.

Also add an "other" method, `more_bags_than`, that:

* expects a `taxi` instance;
* produces/returns whether the calling taxi can hold more bags than the given taxi instance;

And, add an overloaded additional version of `more_bags_than` that:

* expects a number of bags;
* produces/returns whether the calling taxi can hold more than that given number of bags;

To test these new methods, modify `taxi_test` as follows:

* add a declaration using the new 0-argument constructor;
* create 3 `bool` variables to hold results of "sets" of tests (as is done in the posted example `boa_test.cpp`);
* modify the current `return` statement to instead set one of these `bool` variables, and to also test if the taxi created by the 0-argument constructor also has the data fields expected;
* call each of the modifiers at least once;
* set yet another `bool` variable to the result of testing if the taxi(s) modified by the modifiers has the data field values now expected;
* call each version of `more_bags_than` at least three times, with appropriate arguments,
  comparing each call to the expected value for that call; set the 3rd bool variable to the result
  of logical and'ing those 6 comparisons; and

* return the result of the logical and of the three bool variables.

Remember that you can use `nano` to modify these taxi files, and that you can use
`funct_compile` to recompile the modified `taxi_test`.

Submit your files `taxi.h`, `taxi.cpp`, `taxi_test.cpp`, and
`taxi_test_ck_expect.cpp`. 