CIS 130 - Homework #6

Due:

Friday, April 2nd, 11:59 pm (due one day later because of the Cesar Chavez holiday, March 31)

Purpose:

More C++ practice writing `if` statements, writing `bool` and `main` functions, and using local variables, interactive input, and interactive output.

How to submit:

When you are done with the following problems:

- (Assuming that you are still ssh'd and logged into to **nrs-labs**, since you will be writing and testing these C++ functions on nrs-labs using the `funct_play2/funct_compile/expr_play` tools)
- Make sure your current working directory on **nrs-labs** is the one where your homework files are: do the command:

  ```
  ls
  ```

  ...and make sure you see the names of your homework files listed.
  - If you are not in the proper directory, use `cd` *`directory_name`* to go to the proper directory. (For example, `cd 130hw05`)
- use `~st10/130submit` to submit all of your `.cpp` and `.h` files in the current directory.
  - Remember, I don't mind if extra `.cpp` and `.h` files get submitted as well.
  - Make sure that `~st10/130submit` shows that it submitted all of your homework files.
- (ASK ME if this is not clear, or if you have any problems with submission!)

Important notes:

- Each student should work individually on this assignment.
- Remember, you are still expected to follow the Design Recipe for all functions that you write. (Remember to use C++ type names in C++ function contracts, and to write C++ specific examples/tests as discussed in class.)
- Remember to follow the class style guidelines (more have been added for C++ -- see the in-class examples and postings).

Homework problems:

Problem 0

Create, protect, and change to a directory `130hw06` -- type the following from your home directory on nrs-labs:

```
[you1@nrs-labs ~]$ mkdir 130hw06
```

```
[you1@nrs-labs ~]$ chmod 700 130hw06
[you1@nrs-labs ~]$ cd 130hw06
```

(If you log out and come back later, remember to `cd 130hw06` each time to return to this directory!)

Problem 1

Recall the function from Homework 4, Problem 5 that computes number of hours of exercise required to counter the excess fat from eating pizza. Use `funct_play2` to develop a C++ version of this function named `workout` . (`workout` expects expects a number that represents daily pizza consumption, in slices, and produces a number, in hours, that represents the amount of exercise time that you need.

For a daily intake of :	You need to work out for :
0 slices	1/2 hour
1 to 3 slices	1 hour
>3 slices	1 hour +1/2 hour per slice above 3)

Submit your resulting `workout.cpp`, `workout.h`, and `workout_ck_expect.cpp` files.

Problem 2

Now, a little practice with just boolean logic in C++, with a bool function that does not require an `if` statement. Use `funct_play2` to develop a C++ bool function `is_an_op` that expects a single character (so, type `char`, rather than type `string`), and produces whether or not that character is `'+'`, `'-'`, `'*'`, or `'/'`.

While I would prefer that you write a single boolean expression that will be true if the expression given is one of those 4 characters, and will be false otherwise, you may use an `if` if you must...

(Note: it is a COURSE STYLE STANDARD that you use the C++ type `bool`, and the `bool` literals `true` and `false`, when you are dealing with boolean values. Using `int` or `1` or `0` (or other integers) instead within your code will cause you to lose points.)

Submit your resulting `is_an_op.cpp`, `is_an_op.h`, and `is_an_op_ck_expect.cpp` files.

Problem 3

For some more C++ branching practice, and to use `is_an_op`: use `funct_play2` to develop a C++ function `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 is required to appropriately use the C++ `if` statement
- it is required to appropriately use Problem 2's `is_an_op` 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 `do_op.cpp`, `do_op.h`, and `do_op_ck_expect.cpp` files.

Problem 4

To practice with interactive input, write a small function `get_balance` that expects nothing, interactively asks the user to enter a balance owed, and then returns that entered balance.

(Note: you can use `funct_play2` to create this function, although its `ck_expect` will need editing before it will work, as we discussed in class.)

To test `get_balance`, either edit `get_balance_ck_expect.cpp` or write a small testing `main` function in a file named `get_balance_test.cpp` to print a message to the screen saying that you are testing `get_balance`, and then print out the value of a testing call of `get_balance`.

Submit your files `get_balance.cpp`, `get_balance.h`, and either your edited `get_balance_ck_expect.cpp` or `get_balance_test.cpp`.

Problem 5

To practice writing a `void` function, write a function `make_bill` that expects a person's first name, a person's last name, and a balance owed, and it returns nothing, but it has the side-effect of printing to the screen a neatly-formatted personalized bill including the balance owed, in the following format:

```
**********************************************************
To: last name, first_name

Your current balance is: $current_balance

Please remit the balance owed by the 10th of the month
to avoid late fees on the last_name account. Thank you.
**********************************************************
```

That is, the call:

```
make_bill("Thomas", "Jefferson", 38.33);
```

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

```
**********************************************************
To: Jefferson, Thomas

Your current balance is: $current_balance

Please remit the balance owed by the 10th of the month
to avoid late fees on the Jefferson account. Thank you.
**********************************************************
```

Please note: We have not covered numeric formatting using `cout` yet, so you are not required to format the balance so that it prints to two fractional places. Whatever default formatting the `cout` uses for that balance is fine.

To test `make_bill`, write a small testing `main` function in a file named `make_bill_test.cpp` that:

- prints a message to the screen saying that you are testing `make_bill`
- calls `make_bill` at least twice, with different names and different balances
- (don't forget -- it needs to `#include "make_bill.h"` since it calls `make_bill`)

(Also remember: to compile and link the functions making up this small program `make_bill_test`, you need to execute an appropriate `g++` command at the nrs-labs prompt -- remember to include the `.cpp` files for both source code files involved, `make_bill.cpp` and `make_bill_test.cpp`. Remember, too, that you can use the `compile_helper` tool to help you "build" such a `g++` command, if you would like.)

Submit your files `make_bill.cpp`, `make_bill.h`, and `make_bill_test.cpp`.

Problem 6

Here is a rather odd function: `get_name` expects a string describing the "kind" of name it should ask for, it interactively asks the user to enter that kind of a name, and then it returns the name thus entered.

That is, for the expression:

```
get_name("first")
```

...it would ask the user to:

```
Please enter a first name:
```

and then read in, and return, the first name the user enters. If the user entered `Matilda`, then `get_name("first") == "Matilda"` would be true.

And, for the expression:

```
get_name("last")
```

...it would ask the user to:

```
Please enter a last name:
```

and then read in, and return, the last name the user enters. If the user entered `Jones`, then `get_name("last") == "Jones"` would be true.

But, it just trustingly asks for whatever kind of name it is given -- that is, for the expression:

```
get_name("hairy")
```

...it would ask the user to:

```
Please enter a hairy name:
```

and then read in, and return, the hairy name the user enters. If the user entered `Bigfoot`, then `get_name("hairy") == "Bigfoot"` would be true.

(Note: you can use `funct_play2` to create this function, although its `ck_expect` will need editing before it will work, as we discussed in class.)

To test `get_name`, either edit `get_name_ck_expect.cpp` or write a small testing `main` function in a file named `get_name_test.cpp` to print a message to the screen saying that you are testing `get_name`, and then print out the value of at least two testing calls of `get_name`, each with a different argument.

Submit your files `get_name.cpp`, `get_name.h`, and either your edited `get_name_ck_expect.cpp` or `get_name_test.cpp`.

Problem 7

Now write a `main` function in a file named `billing` that does the following:

- it should call `get_name` appropriately to obtain a customer's first name,
- it should call `get_name` appropriately to obtain a customer's last name,
- it should call `get_balance` appropriately to get a balance owed, and
- it should call `make_bill` appropriately to print out a bill for that customer.

Remember: every function called by a function needs to `#include` for that function's header file. Also remember that all of the `.cpp` files used in a program need to be in its `g++` statement for compiling it, and that you can, if you would like, use `compile_helper` to help you "build" such a `g++` statement.

We don't have a good way to write testers for `main` functions, so test-run your `billing` program until you are satisfied that it works properly, and submit your file `billing.cpp`.