CIS 130 - Intro to Programming - Fall 2005
Homework Assignment #9 - **REVISED #5 on 11-07-05**

Homework #9:
HW #9 PART 1 is due by **12:00 NOON** on **Wednesday, November 9, 2005;**
HW #9 PART 2 is due by **12:00 NOON** on **Friday, November 11, 2005**

Week "11" Lab Exercise - due at end of your registered lab section
on either 11-4 or 11-7

WEEK "11" LAB EXERCISE

1. For each separate, unrelated code fragment below, write in the space provided what will be printed; if there is an infinite loop involved, write "infinite loop" in that space instead. Pay attention to how **cout** will format the output!

(You should answer these by reading the code; do not type them in and run them right now. And don't ever run any that happen to be infinite loops, if any... 8-))

(a)
```
int ct;

ct = 0;

cout << "begin" << endl;

while (ct < 0)
{
    ct = ct + 1;
    cout << ct << endl;
}

cout << "end" << endl;
```

____

____

____

____

____

(b)
```
int sum;
int ct;

sum = 0;
ct = 1;

cout << "begin" << endl;

while (ct < 7)
{
    sum = sum + ct;
    ct = ct + 1;
}

cout << ct << endl;
cout << sum << endl;
```

____

____

____

____

____

(c)
```
int ct;

ct = 0;

while (ct < 5)
{
    cout << "Ni!" << endl;
}
```
____

____

____

____

____

(d)
```
int sum_of_squares;
int ct;

sum_of_squares = 0;
ct = 0;

while (sum_of_squares < 20)
{
    ct = ct + 1;
    sum_of_squares = sum_of_squares + (ct * ct);
}

cout << ct << endl;
cout << sum_of_squares << endl;
```
____

____

____

____

2. Consider **asking_profits.cpp**. It uses a sentinel-controlled loop to ask a user for a ticket price, and then prints to the screen the profit for that ticket price; it continues until the user enters a ticket price of **-1** (that's the **sentinel** value, this case). Read over this carefully, and make sure that you understand what it is doing and how it is doing it.

On cs-server, get to the directory where you want to work on the lab/homework. Then copy over **asking_profits.cpp** into this directory within the cs-server command:

```
cp ~st10/130lab11_public/asking_profits.cpp .

                                            ^
```
note this period!!!!!

If you need to, you can also copy over the profit/revenue/cost/attendees functions to this same directory using:

```
cp ~st10/130lect11_public/*   .

                              ^
```
note this period!!!!!

(a) Copy over these files, compile them, and compile/link/load/create-executable for **asking_profits**.

Run **asking_profits**, and immediately enter a ticket price of **-1**. What gets printed to the screen?

(b) Run **asking_profits**, and type at least three different ticket prices. Below, write the ticket price you tried, and the profit it printed out for it.

ticket_price: _______________ profit: _________________

ticket_price: _______________ profit: _________________

ticket_price: _______________ profit: _________________

Now enter a ticket price of **-1** to end the program. Does the program try to print the profit from a ticket price of **-1**?

____

3. I don't want you to think that loops only belong in main() functions; they certainly might be contained in non-main functions, too. Consider example **spam2.cpp** from lecture; it could be rewritten as follows:

in a file **how_many_spam_msgs.cpp:**

```
// Contract: how_many_spam_msgs: int -> void
//
// Purpose: Print the message "I LIKE SPAM!" to the screen <num_times>
//          times, once per line.
//
// Examples: how_many_spam_msgs(3) should cause the following to be
//          written to the screen:
// I like Spam!
// I like Spam!
// I like Spam!
//
// by: Sharon M. Tuttle
// last modified: 11-3-05

#include <iostream>
using namespace std;

void how_many_spam_msgs(int num_times)
{
    int ct;

    ct = 0;

    while (ct < num_times)
    {
            cout << "I LIKE SPAM!" << endl;
            ct = ct + 1;
        }
    }
```

<u>in a file **spam3.cpp**:</u>

```
// Contract: main: void -> int
//
// Purpose: Ask the user how many times they'd like to see the
//          message I LIKE SPAM! written to the screen, and then
//          do so.
//
// Examples: If the user enters 3, then printed to the screen will
//           be:
// I LIKE SPAM!
// I LIKE SPAM!
// I LIKE SPAM!
//
// by: Sharon Tuttle
// last modified: 11-03-05

#include <iostream>
#include "how_many_spam_msgs.h"
using namespace std;

int main()
{
    int num_times;

    // WHERE DO YOU START?
    int counter = 0;

    // HOW LONG DO YOU GO?
    cout << "How many times would you like to see it? ";
    cin >> num_times;

    // DO IT
    how_many_spam_msgs(num_times);

    return EXIT_SUCCESS;
}
```

Read these over carefully; be prepared to answer the following questions:

1. Why is there no **cin** in **how_many_spam_msgs**?

2. How does **spam3.cpp** differ from the **spam2.cpp** posted on the public course web page?

Copy these over from **130lab11**_public:

```
cp ~st10/130lab11_public/*spam*   .
```

...and compile them/compile/link/load/create-executable so that you can run **spam3**.

You will demonstrate that your **spam3** runs for this part of the lab exercise, and you may be asked the above questions, also.

HOMEWORK #9

You are to work **individually** on this assignment.

PART 1: (30% of the HW #9 grade)

Complete problems #1 and #2 below, and submit the files they mention (using **~st10/130submit**) by **12:00 noon** on **Wednesday, November 9th** to receive **any** credit for Part 1 of HW #9.

PART 2: (70% of HW #9 grade)

Complete problems #3 - #6 below. When you are ready, you must submit the files specified in each problem (using **~st10/130submit**) by **12:00 noon** on **Friday, November 11th** to receive **any** credit for Part 2 of HW #9.

1. You should remember the square root function, **sqrt**, from the **cmath** standard library; it takes a **double** value as an argument, and returns its square root.

Here's a simple interactive interface for **sqrt:**

in file **asking_sqrt.cpp**:

```
// Contract: main: void -> int
// Purpose: provide an interactive interface to the cmath sqrt function;
//          it interactively asks the user for a number, reads in what
//          he/she types, and print to the screen the square root of that
//          value.
// Examples: If, when prompted, the user types 4, this would print to the
//          screen:
// The square root of 4 is 2.
//
// by: Sharon M. Tuttle
// last modified: 11-03-05

#include <cmath>
#include <iostream>
using namespace std;

int main()
{
    double input_val;

    cout << "Enter the value for which you'd like the square root: ";
    cin >> input_val;

    cout << "The square root of " << input_val << " is "
         << sqrt(input_val) << endl;
}
```

You can copy this file from **130lab11_public** if you would like to "play" with it:

```
cp ~st10/130lab11_public/asking_sqrt.cpp .

                                          ^
                                     note the period!
```

One use of loops (amongst many) is to provide a more repetitive user interface.

As simple practice of count-controlled loops, write a main function in a file named **ctd_sqrt.cpp**. This should first ask the user how many square roots he/she wishes to compute; then, it must use a **count-controlled while loop** to ask for precisely that many values, and for each one entered, it should print that value's square root to the screen on its own line within a descriptive message (that is, one indicating what kind of value is being printed, such as the one above in **asking_sqrt.cpp**'s main function).

When you are ready, use **~st10/130submit** to submit your version of **ctd_sqrt.cpp**.

2. Or, perhaps your user would prefer another style of repetitive user interface.

As simple practice of sentinel-controlled loops, write a main function in a file named **sentl_sqrt.cpp**. It should repeatedly ask the user for values for which square roots are desired --- but it should also instruct the user to enter a sentinel value of **-1** he/she would like to stop. As long as he/she has just entered a value that isn't -1, it should print that value's square root the screen on its own line within a descriptive message.

Note that you are required to use a **properly-structured sentinel-controlled while loop** in your function; it must not try to compute the square root of the sentinel value of -1, and it may not use an **if** statement in this particular while loop.

When you are ready, use **~st10/130submit** to submit your version of **sentl_sqrt.cpp**.

3. And, some repetitive user interfaces are neither count-controlled nor sentinel-controlled; for example, they might be more-generically **event-controlled**.

As I cannot think of a really reasonable one, you'll write a rather goofy one.

Write a main function in a file named **goofy_sqrt.cpp**. It should continue asking the user for values whose square roots should be computed, for each one entered printing its square root to the screen on its own line in a descriptive message, but also accumulating a running sum of all of the square roots computed so far --- as long as this running sum does not exceed 100, it should continue asking the user for values. As soon as this sum gets to 100 or higher, the function should stop, printing a final message of "That's enough --- goodbye!".

When you are ready, submit your version of **goofy_sqrt.cpp**.

4. Consider the profit/revenue/cost/attendees family of functions. Assume that (for whatever reason) different ticket prices were used for different performances under this scenario. But now, you want the sum of the profits for all of these performances.

(a) Write a main function in a file named **total_profits1.cpp**. The user is repeatedly asked to enter the ticket price for each performance, and to enter a ticket price of **-1** when all of the performances' ticket prices have been entered. The function should compute the profit for each performance based on its ticket price, and keep a running total of the profits. When a ticket price of **-1** is entered, the sum of all of the performances' profits should be printed to the screen in a descriptive message.

Part of your grade for this function will be determined by whether your function uses the appropriate style of while loop, and whether it is appropriately structured.

When you are ready, submit your version of **total_profits1.cpp**.

(b) You have now practiced adding up a collection of values using a loop --- you should be able to add code to that to count how many values you have added together.

Copy **total_profits1.cpp** to a new file **total_profits2.cpp**; at the cs-server prompt, type:

```
cp total_profits1.cpp total_profits2.cpp
```

Now, modify **total_profits2.cpp** so that, in addition to computing the total profits for all of the performances, it will also **count** how many ticket prices have been entered so far. After the ticket price of **-1** is entered, the message printed (giving the sum of all of the performances' profits) should be modified to also include how many performances' profits are included in that total (how many ticket prices were entered).

Be careful --- do not count the sentinel value entered; you are only counting the actual number of ticket prices (actual number of performances) entered.

When you are ready, submit your version of **total_profits2.cpp**.

(c) One more small enhancement: if your loop has computed the sum of all of the performances' profits, and has counted how many performances are involved, then it should be a small step to now determine the **average** profit per performance.

Copy **total_profits2.cpp** to a new file **total_profits3.cpp**:

```
cp total_profits2.cpp total_profits3.cpp
```

... and modify **total_profits3.cpp** so that, after the loop, it computes the average profit per performance, and its closing message now includes the total profits, the number of performances, AND the average profit performance.

When you are ready, submit your version of **total_profits3.cpp**.

5. Now, for something completely different: you are going to write a non-looping, non-main function named **get_code** for use in problem #6.

get_code should not expect any arguments, but it will return an integer. It is to print a "menu" of options to the screen that looks like the following:

```
to determine:        please type:
-------------------------------
profit               1
revenue              2
cost                 3
attendees            4
(or, to quit)        any other number

your choice:
```

Then, it should read in the choice that the user types, and **return** it as the return value for this function.

REVISED 11-07-05:
This is not quite a "regular" function; it returns a value, but interactively gets information from the user. So a "plain" testing main will not quite work. However, you should be able to write **baby_test_get_code.cpp** that calls it three times, and prints to the screen what it returns.

When you are ready, submit your versions of **get_code.h**, **get_code.cpp**, and **baby_test_get_code.cpp**.

6. Now, write a main function in a file named **tkt_price_queries.cpp** that appropriately uses **get_code**.

This function is to use **get_code** to print its menu of options and return the code that the user enters, and to then, as long as the user hasn't asked to quit, ask the user for the ticket price to be used and then perform the computation corresponding to the code that he/she has entered (compute the profit for that ticket price if the user entered the code 1, compute the revenue for that ticket price if the user has entered the code 2, and so on). It should print the value computed to the screen in an appropriate descriptive message.

Part of your grade for this function will be determined by whether your function uses the appropriate style of while loop, and whether it is appropriately structured.

When you are ready, submit your version of **tkt_price_queries.cpp**.