CIS 130 - Intro to Programming - Fall 2005
Homework Assignment #11

Homework #11:
HW #11 PART 1 is due by **12:00 NOON** on **Wednesday, December 7, 2005;**
HW #11 PART 2 is due by **12:00 NOON** on **Friday, December 9, 2005**

Week "14" Lab Exercise - due at end of your registered lab section
on either 12-2 or 12-5

WEEK "14" LAB EXERCISE

Now for something completely different...

For this week's lab, you are going to be introduced to something called an **integrated development environment (IDE)**. This is software that essentially sits on top of a compiler --- instead of typing your program in using pico and compiling it using the g++ command, you type it in in a window that is part of the IDE and compile it by clicking on a button or selecting an option from a menu within that IDE. Other tools are often provided as well.

You'll be introduced to an IDE for C++, called **DevC++**. It happens to be built on top of the g++ compiler, which is a definite advantage; it is also free software, another definite advantage. You might be expected to use it in CIS 230, so that's another reason to let you see it here first. What's the downside? It's only available for the Windows operating system --- which means the description below is being borrowed from an introduction by Professor Ann Burroughs. Any errors should be assumed to be mine!

If you do run Windows on your home computer, you can download it yourself for free if you'd like, from **http://www.bloodshed.net**. That will NOT be required for this lab, however, since you'll be doing it in GH 215, which should have DevC++ already installed. (You are not required to use it for HW #11, either. But if you do decide to try doing so, note that you would need to sftp your resulting files to cs-server to then submit them using ~st10/130submit.)

(For some reason, it is a bit tricky to find free C++ IDE's. I know of two very nice free ones for Java --- DrJava, at www.drjava.org, and BlueJ, at www.bluej.org. There's also a very powerful free IDE from IBM that can be used for a number of languages --- eclipse, at www,eclipse.org. It is not for beginners, I am told; it is supposed to have a bit of a learning curve. It is worth knowing about, however. All three of these latter IDE's work on a wide variety of platforms --- Windows, Mac, and Unix/Linux.)

That said, let's get started.

The following instructions are modified from a lab assignment by Prof. Ann Burroughs.

1. We're in the world of Windows now. Make a directory for yourself on one of the hard drives (probably c: in GH 215?) in a location you can find and remove later.

2. Along with this handout on the course web page, you will see links to five C++ source code files; these, together, make up a C++ program. You should see **show_grade_freq.h**, **show_grade_freq.cpp**, **grade_avg.cpp**, **grade_avg.h**, and **process_grades.cpp**.

There are also copies of these five files on cs-server, in the directory **~st10/130hw11_public**.

I'm not sure what you'll find to be the easiest way to get these from one of these sources into the directory that you created for #1. You could use **SSH File Transfer Client** (which really ought to be called **sftp**, but I'm not sure it is...) to get them from cs-server, or you could somehow copy the files from the course web page --- but that would mean copying and pasting.

3. Bring up DevC++ (in GH 215, it should be in the Start menu, probably under CNRS; if not there, I will announce during lab where it is.)

4. Choose **File -> New -> Project** and then choose **Console Application**. Name the project **process_grades** and click OK. You'll get a dialog window with a title **Create new project**.

Navigate to the directory you made in **Step 1** and click **Save**. You will see a window with some template code in it; DevC++ has given you a basic main() function skeleton. We're not going to use it, though.

5. Choose **Project -> Add to Project**.
Add these files to the project: **show_grade_freq.h**, **show_grade_freq.cpp**, **grade_avg.cpp**, **grade_avg.h**, and **process_grades.cpp** .

6. Choose **Project -> Remove from Project**.
Select **main.cpp** to be removed. You can answer "no" to the "save main.cpp?" question.

7. Open the file **process_grades.cpp** (double-click it in the left frame).
Add this statement to the program, right before the return of EXIT_SUCCESS:

```
system("PAUSE");
```

(This is an oddness of a number of graphical IDE's --- without this, the console window disappears so fast that you cannot see your program's results!)

8. Choose **Execute -> Compile and Run**
Your program compiles, and if there are no compilation errors, it goes into execution. Enter what it asks you to enter...

9. Let's close this project: **File -> Close Project**
Answer **Yes** to the Save Changes to process_grades dialog prompt.

10. Let's open this project again: **File -> Open Project or File**
The project is now called **process_grades.dev**. You can open individual files, but opening the project itself is the only way to open the whole "package".

11. If you wanted to enter files into a project from scratch, the command is **File -> New Source File**.

12. Put your name on the **Next:** list on the board; when it is your turn, I'll observe you running this program (to show that you successfully did so under DevC++).

NOTE: WHEN I COME to OBSERVE YOUR PROGRAM, I WILL ALSO ASK TO SEE YOUR ANSWERS TO THE FOLLOWING:

Consider:

```
int val1, val2, val3, val4, val5;

val1 = 15;
val2 = val1++;

val3 = 15;
val4 = ++val3;

val5 = 15;
val5 += 3;
```

What are the values of val1, val2, val3, val4, and val5 after this fragment has been run?

val1 _______ **val2** _______ **val3** _______ **val4** _______ **val5** _______

(Not sure? Why not write a little main() that does this, and then prints out the values of **val1**, **val2**, **val3**, **val4**, and **val5**?)

Note that you will lose points for **incorrect** values for **val1** through **val5** above (**-3** for each one incorrect); so, you need to be confident about your answers before you give them.

Once you've done this, you have received credit for this lab exercise (minus any points for incorrect values above).

13. Close the project again (**File -> Close Project**). If this weren't files you could grab from the course web page or cs-server whenever you'd like, I'd recommand that you copy any of the files that you need to save from the directory that you made in Step 1 to a more permanent location such as a flash drive or or to somewhere on cs-server (via sftp). If the work in an HSU lab is for homework, you should remove all of your files from the computer's hard drive before you leave.

14. Final note: In general, you need to have all of the pieces of your program in a single project.

HOMEWORK #11

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

PART 1: (30% of the HW #11 grade)

Complete problem #1 below, and submit the files it mentions (using **~st10/130submit**) by **12:00 noon** on **Wednesday, December 7th** to receive **any** credit for Part 1 of HW #11.

PART 2: <u>(70% of HW #11 grade)</u>
Complete problems #2 and #3 below. When you are ready, you must submit the files specified in each problem (using **~st10/130submit**) by **12:00 noon** on **Friday, December 9th** to receive **any** credit for Part 2 of HW #11.

1. We have discussed pass-by-reference parameters now; you need to write at least one function that uses them.

Consider a program which requires the user to enter **y** or **n** to a number of questions. Each time, it may *really* want to be sure that ONLY a **y** or **n** is entered by the user --- so much so that it should KEEP asking until a **y** or **n** is given.

This might be convenient to encode into a function; write a function **get_ans** that doesn't return anything, but has one pass-by-reference parameter, a **char**, that will be an output parameter, set to the user's answer. It asks the user to enter **y** or **n**, and reads in what they answer, until either a **y** or an **n** has been entered. Then it sets the output parameter accordingly, and should be finished.

How will you test this? A small testing-main **test_get_ans**.**cpp** should be written that calls **get_ans** at least twice, and then prints out, within a descriptive message, the value of its pass-by-reference argument **after** the call to **get_ans** each time. Make sure that you try entering values beside **y** and **n**, to see if it stubbornly keeps asking; and, make sure you answer **y** at least once and **n** at least once. (I won't be able to see if you've done this, but you should still do it...!)

When you are satisfied with them, submit **get_ans.cpp** and **test_get_ans.cpp** using **~st10/130submit**.

2. Another common use of pass-by-reference parameters is to return more than one result.

As a quick-n-sleazy example, consider an array of numbers. You might try to find out both its average and its lowest value.

Write a function **avg_and_min**, whose return type should be **void**, that requires **four** parameters: two **input** parameters, an array of double values and an int giving the size of the array, and two **output** parameters, that are hoped to be set to be the average of the values in the array and the smallest value of the values in the array. The two output parameters should, of course, be **pass-by-reference**.

Note, because of the use of output parameters, that will affect how you should write your Examples.

For example:

```
// Examples: for double vals[5] = {-5, 12, 1056, -909, 48},
//    and double thr_avg and double thr_min,
//    after the call avg_and_min(vals, 5, thr_avg, thr_min),
//        thr_avg == 40.4
//        thr_min == -909
//
//    (and insert at least one additional DIFFERENT example of your
//     choice)
```

How will you test this? A small testing-main **test_avg_and_min.cpp** should be written that sets up at least two different arrays for testing use (including all that you include in the **Examples** section for

avg_and_min). You can use the "1's == pass, 0's == fail" style of testing output, remembering that you want to compare what the output parameter arguments are after each call to what you expected them to be.

This one is suitable for creating an output file; so, when you are satisfied with these, create an example output file using:

test_avg_and_min > test_avg_and_min.out

and then submit **avg_and_min.cpp**, **test_avg_and_min.cpp**, and **test_avg_and_min.out** using **~st10/130submit**.

3. Now -- we really need some **switch** statement practice.

Let's say that people have assigned point values to all of the face cards in a deck of playing cards:

J - 10
Q - 12
K - 14
A - 16

Write a function **sum_hand** that takes no arguments, but returns an **int**. It should call function **get_ans** to ask the user if he/she has more cards to enter; while the answer is **y**, **sum_hand** should then ask the user to enter **J**, **Q**, **K**, or **A**. It should use a **switch** statement, then, to add to a running total the point value of each card entered. When the user answers **n** (with the help of **get_ans**), then **sum_hand** should simply return the sum of the face cards entered.

How will you test this? A small testing-main **test_sum_hand.cpp** should be written that calls **sum_hand** at least twice, and then prints out, within a descriptive message, the value of the sum of the hand entered. Make sure that you try entering at least one of **J**, **Q**, **K** and **A** before you are done, to "exercise" those switch cases; because this is so interactive, it is harder to do traditional "1==pass, 0==failed" testing, but you should interactively run your Example cases from **sum_hand**'s opening comment block, of course.

When you are satisfied with them, submit **sum_hand.cpp** and **test_sum_hand.cpp** using **~st10/130submit**.