ECE 1305, Ian Scott-Fleming

Lab 02

Looping, and Binary Numbers

Two goals for this lab. First, to give you a better understanding of binary numbers, how computers store (integer) data, and some of the limitations imposed by fixed-size data representation. Second, to give you some experience with loops in C++.

Here are some sample programs that demonstrate basic file I/O and looping:

- hello_file.cpp (reads the file numbers.txt)
- count_down_while_loop.cpp
- count_down_for_loop.cpp
- print_table.cpp

(All the sample code for this lab are in this file: lab_02_sample_code.zip .)

Study these programs, then compile and run them, so you have some useful examples to refer back to if/when you get stuck on the exercises below.

Reading for this lab: Chapter 2 of the textbook, and my tutorials on binary and floating point numbers, found on the course materials page.

Part 1: Write a program to prompt the user for 6 integer numbers, add them up, and print out their average (as a floating point value)

This should be pretty straight forward. Your program should:

- declare a variable to sum the numbers into
- declare a variable to read the numbers into
- Repeat 6 times:
    - prompt for number
    - read the number
    - add it to the sum
- Calculate the average, and print out the sum and average

(Suggestion: have your program print out the number and sum each time through the loop to help you test/debug.)

Part 1 b: Modify your program to handle an arbitrary number of input values. Have it prompt the user first for the number of values to read, then loop that many times, reading in the values and summing them.

Call your program ***add_numbers.cpp***

Part 2: How big are the different integers on your system?

Study, and then compile and `variable_sizes.cpp`, then answer the following questions (put your answers in a text file to turn in):

How long are the various integer data types on your computer system, in bits and in bytes? (This will be a function of your processor and your operating system, and can be limited by your compiler). The sizeof(...) function tells you how many bytes there are in the thing passed to it.

```
char, unsigned char
short, unsigned short
int, unsigned int
long, unsigned long

float
double
long double
```

What are the largest and smallest values you can store in each of the integer data types above? You'll need to calculate this from the sizes you get. You can check your answers using the windows or mac os x calculator; it will help to put it in "programmer" mode. (Recall that each byte has 8 bits, and with ***n*** bits, you can represent $\boldsymbol{2^n}$ different bit patterns, or values.)

What are the ranges for the floating point types on your system (what do the exponents range from/to)? The program won't tell you that, but you can find the answers with a little googling (they should also be in my tutorial on floating point numbers). How many digits of precision are there for each type?

Put your answers in a text file called ***variable_sizes.txt***

Part 3: Verify your answers for the integer types of part 1 with a C++ program.

First, study `char_ranges.cpp` and understand what it does. Then compile it and run it, and see if it gives you the output you expected. (If it tells you you have 16-bit chars, then use `char16_ranges.cpp`, which steps by 64, instead of stepping by 1's.)

Note that the example program demonstrates all 3 types of loops – a do_while, a while, and a for loop.

Then, write a program which demonstrates the range of values for signed and unsigned shorts and ints. You can have your program start outputting values 100 below the (expected) max value, and increment it 200 times, to see what happens. If it doesn't give you the expected results, make sure you know what the max value for your data types is.

This looping is a good place to use for(...) loops – generally used for counting loops – because you know how many times you want to go through the loop. However, use each type of loop in your program, so you are confident you know how each type works.

If your numbers don't wrap (back to 0 or back to a large negative number), that means you didn't get the right answer for the range in part 1.

Call your program ***int_ranges.cpp***

Part 3: What is the precision of floats, doubles, and long doubles on your system?

We'll figure that out by trying to store as many digits of PI as possible in each of the data types, output it, and then check to see where the output value goes wrong.

Study the program `floating_pt_precision.cpp` to get you started. Compile and run it, and notice that you don't get the exact values for f, d and ld that were assigned. Why not? Put your answer in your text file from part 1. Then modify the program to see how many digits you can get to PI in the three floating point data types. Assign the value in the comments for PI (it's good to more digits than we can represent in any standard data type).

Note the use of "setprecision" to control the number of decimal digits to put out.

(Be sure to put **your name** in it – you don't want to give ME credit for it! And it is common practice to give some credit to the original author when you start from someone else's code; and generally put the URL where you downloaded it, so others reading your code will know where you started from).

C++ doesn't have a constant defined for PI, so you have to create a variable and set it equal to PI in programs where you need it. The last few lines of the are a common way to get PI to double-precision. How does it work? (hint: look up 'trig functions' in the index of the textbook, and think about what the arc-cosine of an angle is.) Put your answer in your text file from part 1.

*Call your program **float_precision.cpp** (**not** floating_pt_precision.cpp !)*

Challenge Problem (optional):

This will test whether you really understand binary numbers, computer arithmetic, and looping. Write a program which prompts the user for a positive integer, reads it in, prints it out, and then prints it's representation in binary (i.e., 0's and one's).

There are several ways to get the state of any bit in a number. One way is to use bit-wise and'ing (look it up! & instead of &&). You can also do it with a little creative use of division and the modulus operator. Hint: what does the following do?

```
int n = 13;            // binary 0000 1101, 1*8 + 1*4 + 0*2 +1*1
bool b;

b = ( (n % 8) / 4);    // what does % 8 do?  then divide it by 4
```

Add some looping, and you should be able to figure out how to output a number in binary.

*Call your program **"binary_values.cpp"***

What to submit:

Submit only your source code files:
1. add_numbers.txt
2. variable_sizes.txt
3. int_ranges.cpp
4. float_precision.cpp

If you tackle the challenge, also submit
5. binary_values.cpp