CS 346 – Lab 2 – "C" World

Today we'll take an excursion into the world of the C programming language.

Advantages to learning C:
1. C is an older relative of Java. Java was developed based on C. If you know how to program in Java, then picking up C will not be difficult. You should become multilingual in computer languages, and C is a fairly easy stepping stone for you. Knowledge of C will give you additional insight into how the Java language works.
2. The C language was created to facilitate the implementation of the Unix operating system, which itself was the precursor to Linux. Thus, if we are going to use the Linux operating system, it makes sense to know something about the language it is written in.
3. For decades, much compiler research has been done for the C language. Therefore, C programs are highly efficient.
4. C is commonly used in research and development, for example by the National Security Agency.

At first glance, a C program looks like Java and vice versa. It would not be worth you time to study how to compose such things as while loops or switch statements in C, because these are the same as in Java. It turns out that there are only a few significant differences that you should be aware of, and the focus of today's lab is to explore these differences. One difference I should mention right away (although we probably won't need it for today's lab) is that in C, local variables must be declared at the top of a function, rather than in the middle when you suddenly need to declare a new variable.

Log into the cs.furman.edu server as before. Create a new directory called lab02 and copy the files from my ~chealy/www/cs346/lab02 directory.

Performing Output

Just as you could include the Scanner class to handle I/O in Java, the corresponding header file responsible for I/O in C is called stdio.h. It's called "standard I/O" as opposed to file I/O, which we will not discuss today. The built-in file stdio.h contains predefined functions for I/O as well as some predefined identifiers. For example, "stdin" refers to standard input (the keyboard), and "stdout" refers to standard output (the screen). But one nice thing is that we only occasionally need to refer to stdin and stdout explicitly.

Simple interactive I/O in C is accomplished by 2 functions: printf() and scanf(). The letter 'f' in these names means the I/O may be *formatted*. The C printf() function works like the Java method System.out.printf() that you may already have experience with.

The printf() function takes as many parameters as you want. The first parameter is always a string, called the *format string*. If all you are doing is printing a string constant, then this will be the only parameter to printf(). However, if you want to print a combination of strings, integers, characters and/or floating-point numbers, you need to insert "place holders" inside the format string, and then add these parameters to the end of the printf() call.

Look at the program hello.c. Compile this program using the gcc command. (This stands for the GNU C Compiler.) The command you would type is `gcc hello.c –o hello`. The –o option says that the executable file should be called hello. It is customary for executable files not to have an extension in their names. After compiling, run the program by typing `./hello` (and hit enter).

If you look at the source code of hello.c, you'll notice that this program contains 4 printf() calls. The last three actually have the same effect. In the 2nd printf call, the number is part of the string. But in the 3rd call, we have placed a special "place holder" **%d** to indicate that we are going to put an integer in that place. And the second parameter is the value of this integer to be printed. The last printf() call shows that we can use a variable in place of a constant to do the same thing.

The technical term for "place holder" is *format specifier*, and it begins with a percent sign, ends with a formatting character code (**d** for decimal integer, **o** for octal integer, **x** for hexadecimal integer, **c** for character, **s** for string, **f** for float, and **lf** for double). In between the percent sign and the formatting character code you may put an integer value to specify how to format this value.

For example, see what happens if you change %d to %6d. (No, it does not mean sixpence!) This means we print the number with a field width of 6 characters, right justified. If you want to be left justified, put a minus sign in front of the number, as in %–6d. Don't you 'C' how easy it is to format output ☺ ?

Now, add a fifth printf() call that will use %f instead of %d, and make the second parameter of the printf() call 346.0 instead of 346. We literally want to print the number 346.0. To format floating-point numbers, you can specify both a field width *and/or* a precision. Experiment with %10.3f instead of just %f. Experiment with other numbers as well. Do you understand the role of each number? √

Input

Next, let's examine the file time.c. This small program performs both input and output. The input is handled with the scanf() function. We use the same formatting character codes in scanf() as we do for printf. The first parameter to scanf() is the format string, and the remaining parameters are the variables you are reading into. CAUTION – Notice that in each case, the variables are always preceded by a '&' character. This is because these parameters are technically addresses of variables, rather than the variables themselves. If you forget to put the '&' character, you will get a core dump when you try to run the program.

Compile and run the program time.c. (If you call your executable 'time', then you'll need to type `./time` so you don't inadvertently run the time system call.) Notice the formatting of the minute is %02d. The '2' means that we should prepare for a 2-digit number, and the leading '0' means to pad the front of the number with zeros. Thus, it is not necessary for the programmer to hard-code a check to see if the minute number is less than 10.

Uh-oh! There is a mistake in the program. It reads in 2 integers and then a character, but the character it's reading is actually the newline character you typed right after you gave the second integer. Change the 3rd scanf() call so that you throw away the newline character and properly obtain the am/pm character.

Modify this program so that it also asks the user to enter the number of seconds (just like the minutes), and then prints out the time in hh:mm:ss format as well. √

It's good to know how to use scanf, but in the long run it is better to read input line by line. In C, the built-in function to read a line of text is called fgets(). Read the man page for fgets to learn its format. How would you call fgets if you wanted to read a line of text from standard input (stdin), into a character array called line that has a length of 80 characters?

____

Incidentally, there is a simplified version of fgets() called gets() that specifically works only for reading from stdin.

Functions

The statements in a C program are organized into functions much like Java. One feature that's especially convenient about C functions is the ability for it to change multiple values for you – as if you had the ability for the function to return more than one value without having to declare global variables.

The secret is to use "pointers". The purpose of a pointer is to hold the address of another variable. If a function is given a pointer, then it has the power to change the variable. For example, consider the function call: swap(x, y). We want the swap function to exchange the values contained in variables x and y. The only way to do it is for us to pass pointers to x and y – or else the function really would have no effect. So instead, we should say swap(&x, &y). In the function call, it is necessary to pass the address of x and y, because the value of a pointer is always the address of some variable.

Take a look at the file swap.c. In the header of the swap() function you see the formal parameter list to be: (int *x, int *y). This states that both x and y here are "pointers to int". If x is a pointer to int, then inside the function we use the * operator to obtain the actual value of what is being pointed to.

Modify this swap program so that the 4 statements (2 printf and 2 scanf) that get the 2 input values are modularized into a special input() function. This function, like swap, will have 2 parameters that are both pointers to int. The call to the input function will be much like the call to the swap function. √

Structures

Perhaps the most conspicuous difference between C and Java is that there is no object-oriented programming in C. C was developed in 1972, years before OO was invented. The C++ language was released in 1986 and supported OO as well as the traditional procedural programming paradigm. And Java, announced in 1993, is fully OO.

In C you cannot declare a class, so instead we use a `struct` – which stands for structure (which some people also call a "record"). A struct in C is like a class in Java or C++, but there are only data members, no constructors or member functions of any kind. Yes, you could write functions that manipulate structs, but they would act like "static" functions in Java. Also, we don't have the encapsulation that you see in OO – everything is public: you can't declare anything to specifically have public, protected or private access.

Examine carefully the program struct.c. At the beginning of the program we declare a struct that represents a new data structure to model information about a student. The program's main() function delegates the work to 3 functions: get_input(), compute_averages(), and output(). I have written the first and last of these, and you are to write the compute_averages() function.

Compile and run the program. I have provided an example input file called struct.in. Use the redirection operator < so you can run the "struct" program on this input. You will notice that the students' averages need to be calculated. Feel free to experiment further by adding more names and scores to the input file.

Once you have implemented the compute_averages() function, modify the format string in the output() function so that the names, scores and the average are in neat columns. Also, you should change the way the average is printed so that it only prints to 1 decimal place instead of 7 by default. √

By the way, this program struct.c makes use of two interesting built-in functions. In your own words, describe what they do. In particular, what is the equivalent in Java?

atoi ____

____

strtok ____

____

Revisiting the Range

Now that you have a taste of C, let's write a small program from scratch. Refer to the Range.java program you worked on in the previous lab. Write an equivalent C program.

Hints: To compare two strings in C, you need to include <string.h> at the top of your source file, and then use the strcmp() function. This function takes 2 strings, and returns an int value. The effect is similar to comparators that you may have worked on with Java: the return value of 0 means the two strings are the same. A positive (negative) return value means the first string would appear later (earlier) in the dictionary.

How to use command line arguments… You already so how to do these in Java, and the concept is similar in C. To make use of command line arguments, we declare our main() function as follows:

```
int main(int argc, char *argv[])
```

The first parameter is the number of command line arguments, and the second parameter is an array of strings (technically, an array of pointers to char – as there is no string type in C). However, you should note that, unlike Java, in C the [0] (first) command-line argument is the name of the executable program itself. The input proper begins at [1]. √

A Mystery

Before we leave today, I want to show you a fun surprise. Take a look at the program mystery.c. What do you think this program does? The programming style definitely leaves much to be desired. It is one of the most infamously obfuscated programs ever written. So, let's compile and run the program. How was this program even able to compile?

```
main() {
   if (you_love(C))
      honk();
}
```