Lab 2

Introduction to C programming

The purpose of this lab is to introduce the C programming language and working with different data types in C.

By the end of the lab, you will be able to:

- **compile** a program written in C into a binary executable and run the executable
- see the effect of using different data types on the same computation
- understand when to use a particular data type
- work with **for loops** and **while loops**

Compiling your first C program

Go to your ece120 directory and create a new subdirectory called `first-program`. Go inside this new directory and create a new file called `hello-world.c`.

Cut-and-paste the following code into `hello-world.c` (we didn't want to spoil the moment by giving it to you 😊)

```c
#include <stdio.h>

#define MESSAGE "Hello, x86 world!\n"

int main()
{
    printf(MESSAGE);
    return 0;
}
```

Step 1: Compile

There are many steps in the compilation of a program (which you will learn about in other classes), but in ECE 120 we can do them all with one invocation of the compiler "gcc" like so:
gcc -g -Wall -Werror -o hello-world hello-world.c

What does this mean?

- **-g** means compile with debugging symbols. This will allow the debugger to find the C source line corresponding to machine code (you'll learn more about the debugger later).
- **-Wall** means warn about all possible problems with your code. **Pay attention to warnings the compiler outputs.**
- **-Werror** means fail if there are any warnings. We recommend using this option unless you are absolutely sure that the warnings you are getting are harmless.
- **-o hello-world** indicates that the program to be built should be called hello-world
- **hello-world.c** is the input file implementing your program, written in C

It is not necessary to give the -o flag when you compile your C code. If you do not specify the name of your output file using the -o flag then the compiler gives the output in a file named a.out. Try compiling "hello-world.c" without the -o flag to see for yourself.

**Step 2: Execute**

Okay so after compiling, go ahead and run the program you produced:

```
./hello-world
```

This will load the program into memory and run it. You now have created a program that you can run just like any other Unix program.

**Why the ./ ?**

Recall that the Unix command conventions is:

- program [arg1] ...

While the program can be a path (containing slashes), it can also just be the name of a program (no slashes) to make a command line user's life easier. When you just specify a name, the operating system searches through standard locations for binaries (such as /usr/bin) to find the program with that name. By default, most Unix systems will not search the current working directory (which is generally a good thing, because you don’t want to be accidentally executing something you happen to be in the directory of). So you need to provide a path (something with slashes) to execute a program not in the standard locations. Adding ./ is usually the easiest way to do this. If you move to the parent directory, this would also work:

```
first-program/hello-world
```

**Data Types in C**

C has a concept of **data types**, which are used to define a variable before its use. The definition of a variable will assign storage for the variable and define the type of data that will be held in the location. There are several integer data types, floating point data types for holding real numbers and more. In addition you can define your own data types using aggregations of the native types (you'll learn more about that in ECE 220). Here are some that you will use in ECE 120.

**int**

The int data type represents a signed integer. For most of your simple computations, this will be your go-to data type. On the lab machines running on 64-bit Intel x86 processors, the int type is 32 bits, so it represent any integer between -2147483648 and 2147483647.

Notice that we say on 64-bit Intel x86 processors. This is because the size of the integer data types can vary from processor to processor. For example, on the LC-3 (a 16-bit processor), an int is 16 bits. In this author's opinion, lack of consistency of integer data types is a significant deficiency of the C language, and is something to be aware of if you write code that you want to run on different processors. (Floating point types are consistent and don't have this problem). For exercises in this class, we can assume that ints are 32-bits. The interested reader can also learn about a header `<inttypes.h>`, which provides a set of types that are guaranteed to be consistent across platforms.
float

The float data type allows you to store floating point numbers which the int data type cannot handle. Remember the IEEE 754 floating point representation of numbers? float denotes the 32-bit, single precision format, which has a range of \(-3.4 \times 10^{38}\) to \(+3.4 \times 10^{38}\). Many real world computations result in a floating point result. If you use the int data type to hold such results, you will not get accurate answers. This is because the int data type truncates everything after the decimal position. So if your answer is 3.14 and you try to hold this in a variable defined as int, you will get 3 as your answer. The 0.14 part will be truncated thereby resulting in inaccurate results.

char

This is an 8-bit signed integer (range -128 to 127) which is typically used to store simple characters that are available in the ASCII table. All the characters that can be represented as an ASCII number can be stored in the char data type.

Its not over yet!

C provides different data types within the categories of the basic data types. For example you have different kinds of integer data types, each having different properties but having one common link that they are all integers.

Integer data types

- unsigned int - Like the name suggests, can hold only positive numbers. Can hold a higher value than normal int as the same number of bits are used to represent only positive numbers now.
- long - On the lab machines, this a 64-bit value with a much bigger range when compared to the simple int data type.
- unsigned long - On the lab machines, 64-bit unsigned integer.
- short - 16-bit integer
- refer to C Programming Reference for the complete list

Float data type

- double - provides more precision than the simple float data type. Up to 10 decimal places of precision.

Numerical computations

Before starting to work on this part, make sure to update your local svn copy by typing the following line in your ece120 directory.

```
svn update
```

In your ece120 directory, you should now have a new directory called lab2. Inside lab2, you should find files lab2.txt and lab2.c which you will need to modify and commit as you follow the instructions provided below.

Download or cut-and-paste the program num_comps.c that contains the following code. You will modify this code during the following exercise.
#include <stdio.h>

int main()
{
    int val1 = -22;
    int val2 = 33;
    int res;

    res = val1 * val2;
    printf("%d\n", res);

    return 0;
}

Modify num_comps.c to perform the following three computations. You will perform each of the computations below 4 times, each time using a different data type given below. Write your answers to each computation in your lab2.txt file. Don't forget to commit to your SVN repository when you are done; otherwise, we will have nothing to grade.

1. \(-22 \times 33\)
2. \(12 / 3\)
3. \(10 / 3\)

The * operator in C multiplies two numbers. The / operator divides two numbers.

Carry out each of the above mentioned computations by storing the first and the second values and the result as the following data types. To do that, you will need to update num_comps.c accordingly. Make sure that you update the printf command to match the data type!

1. int
2. unsigned int
3. float
4. double

Perform the following two computations once using the int data type. Write your answers to each computation in your lab2.txt file

1. \(12 \% 3\)
2. \(-11 \% 3\)

Make sure to update the printf specifiers like %d to match the data types you are using. For float and double use %f for unsigned int use %u

Iterative Constructs in C

In every programming language, there are circumstances where we want to do the same thing many times. For instance, you want to print the same words ten times. You could type ten printf function calls, but it is easier to use a loop. The only thing you have to do is to set up a loop that executes the same printf function ten times.

There are three basic types of loop constructs in C:

- for loop
- while loop
- do while loop

We are intentionally going to skip introducing the do while loop for now.

The for loop

The for loop loops from one number to another number and increases by a specified value each time.

It uses the following structure:
Here is a simple example of the \texttt{for} loop. This C program prints "Hello World !" 3 times on the screen.

```c
#include <stdio.h>
int main()
{
    int i;
    for (i = 0; i < 3; i = i + 1)
    {
        printf("Hello World !\n");
    }
    return 0;
}
```

This \texttt{for} loop is equivalent to the following infinitely nested \texttt{if} statement:

```c
{
    int i = 0;
    if (i < 3)
    {
        printf("Hello World!\n");
        i = i + 1;
        if (i < 3)
        {
            printf("Hello World!\n");
            i = i + 1;
            if (i < 3)
            {
                printf("Hello World!\n");
                i = i + 1;
                if (i < 3)
                {
                    ...
                    ...
                    ...
                }
            }
        }
    }
}
```

Let's trace through what happens at each iteration:
In iteration 0

```
i = 0 (first clause of for construct)
The program then checks whether i < 3
result = true, hence the program flow goes into the loop
print Hello World !
The for loop now increments i (because i = i + 1 is the update value that is mentioned)
therefore i = 0+1
```

In iteration 1

```
(i is now 1)
The program then checks whether i < 3
result = true, hence the program flow goes into the loop
print Hello World !
The for loop now increments i (because i = i + 1 is the update value that is mentioned)
therefore i = 1+1
```

In iteration 2

```
(i is now 2)
The program then checks whether i < 3
result = true, hence the program flow goes into the loop
print Hello World !
The for loop now increments i (because i = i + 1 is the update value that is mentioned)
therefore i = 2+1
```

In iteration 3

```
(i is now 3)
The program then checks whether i < 3
result = false, hence the program flow breaks out of the loop
```

Final output

```
Hello World !
Hello World !
Hello World !
```

```
for (index = 0; index < max_index; index=index+1)
{
    ...
}
```

The above usage of for is a time-honored way of iterating through integers in the interval \([0, \text{max\_index})\) \((0 <= \text{index} < \text{max\_index})\). It's
The while loop

The while loop does the same task as the for loop but is used when you don’t know how many times a particular task has to be done.

Here is a simple example of the while loop. In this C program, an integer value is entered by the user. This value then tells our program the number of times "Hello World !" has to be printed on the screen. This program effectively does the same task as the above program if the user enters 3

```c
#include <stdio.h>
int main()
{
    int count = 0, number_of_times;
    scanf("%d", &number_of_times);

    while(count < number_of_times)
    {
        printf("Hello World !\n");
        count=count+1;
    }

    return 0;
}
```

As you can see for simple looping operations such as shown above, using the for loop results in a much more concise and cleaner code. Hence, we suggest you use the for loop (instead of the while loop) to carry out simple tasks like this. Below, we show a much more appropriate use of the while loop.

A more appropriate use of the while loop

The code shown below is used to accept a series of characters from a user. When the user hits return (enter on the keyboard), the program stops accepting characters and displays how many characters the user has entered. In a real program you would store these characters in an array for further manipulation. We will introduce arrays and operations on them in ECE 220.
The rest of this lab's assignment

Your lab2.c file contains the following program, in which the iteration is done using a **for loop**.

```c
#include <stdio.h>
#include <math.h>

int main()
{
    int a;
    for (a = 256; a > 1; a = sqrt(a))
    {
        printf("%d
", a);
    }
    printf("%d
", 1);
    return 0;
}
```

Your task is to convert the above program and use the **while loop** construct instead of the **for loop** construct.

Edit this code in lab2.c to use only the **while loop** and commit the modified C file to your SVN repository.

You will probably see the error **undefined reference to `sqrt'** when you try to compile lab2.c. The problem is that the `sqrt()` function is defined in a C math library (with header file `<math.h>`), not in the standard C library (with header file `<stdio.h>`). This C math library provides other functions such as `pow()`, `log()`, `sin()`, `cos()`, `tan()`, etc.

But the compiler does not automatically know that it should link the math library during the compilation process. To use a math function provided in the `<math.h>` header file, you must give the `-lm` flag to the `gcc` compiler like so:

```bash
gcc -lm -g -Wall -Werror -o lab2 lab2.c
```
When rewriting the **for loop** into a **while loop**, make sure that the variable `a` is initially assigned the value 256.

Remember, the output of the series should be exactly the same as the output of the program given using **for loop**.

**Hint**

Try unrolling the **for loop** into a nested **if statement** as shown earlier in this lab. This will give you a clear idea of what is happening and will help you transform your code to use a **while loop**.

**Hand In Instruction**

Only submit the files which are required (**lab2.c, lab2.txt**). Do not submit hello-world.c or num_comps.c