C Programming Lab

Compiling Programs

The standard C compiler on Linux based systems is gcc; this application can be used to compile programs as follows (with an input file of “test.c”).

```
fly@VM:~/C$ ls
test.c
fly@VM:~/C$ gcc test.c
fly@VM:~/C$ ls
a.out  test.c
fly@VM:~/C$
```

The output from this compilation is a binary (executable) file “a.out”. Of course we would prefer to have a file with a specific name. In order to specify an output file name we can use the “-o” flag followed by the name of the output file, as follows (with the input file named “test.c” and the output file named “test”):

```
fly@VM:~/C$ gcc test.c -o test
fly@VM:~/C$ ls
test  test.c
```

The output from this compilation is a binary (executable) file “test”. We can then run binary files by either including the directory where they are stored in our PATH environment variable or by specifying that the binary should be executed from the local directory, as follows:

```
fly@VM:~/C$ ./test
```

There are some other flags that we will frequently include when compiling C programs, as summarised in the table below.

Flag	**Action**
-g	Provides additional information when using gdb (such as including the program code in the binary file)
-w	Suppress any warnings (which we might get when compiling vulnerable applications).
-m32	Compile for a 32 bit processor
-m64	Compile for a 64 bit processor

As an example, the following compilation command will compile a 32-bit application, prepare the binary for analysis in gdb and suppress any warning:

```
fly@VM:~/C$ gcc -o test test.c -g -m32 -w
```

(note: you may get an error if your system runs on a 64 bit processor; using the following should solve this problem

sudo apt install gcc-multilib

)

Basic Outline of a C Program

C is a programming language that uses functions – most other programming languages are either based on the structure of C or are written in C (for example, both JAVA and PHP are written in C). A function in C has a return value, a name and a list of variables that are input into the function. A required function in a C program is the "main" function. The main function will always return an integer value (as it finishes and returns control to the application or the operating system that started the program). The main function may take in some form of user input – this may be the command line arguments used when launching the program.

C uses curly braces "{" and "}" in order to denote the start and end of logic (respectively). The curly braces are also used to indicate the start and end of functions. The example below is the most basic outline of a C program that returns an integer value and accepts user input from the command line.

```
int main (int argc, char **argv)
{


}
```

The main function above takes in two variables. One of those variables is an integer value called "argc" (which is the number of command line arguments) and the other is a variable called argv, which is actually an array of pointers to character variables). In-between the curly braces is the code of the main function. The main function may call other functions (either from libraries/APIs or it may call functions that exist as a part of the program that is being written.

In order to give some basic functionality to our program we will start by including the stdio library (that allows us to use functions such as printf()). We will then use the printf() function to print out "Hello World". The semicolon at the end of the line should be noted (not including semicolons is a common source of compilation errors).

```
#include <stdio.h>

int main(int argc, char **argv){
        printf("Hello World!\n");
}
```

Data Types (variables)

A variable in C is technically an area of memory that is going to store some information. C is a strongly typed language, which means that variables must be declared as being a particular type (integer, float etc) before they are used.

Data Types: Integers

Integer variables are able to store whole numbers, e.g. -6, 0, 546, etc. The variable must be declared as an integer in the following way.

```
#include<stdio.h>

int main(int argc, char **argv)
{

int x;

}
```

Here a variable called “x” has been declared. The ‘int’ before it tells the C compiler that the variable is an integer. This allows the system to allocate the appropriate amount of memory. Multiple integer variables can be declared in the following way.

```
#include<stdio.h>

int main(int argc, char **argv)
{

int x,y,z;

}
```

Values can then be assigned to variables in the following way.

```c
#include<stdio.h>

int main(int argc, char **argv)
{

int x,y,z;
x=-6;
y=10;
z=19090;

}
```

Data Types: Floats

Float (real) variables are able to store any real numbers, e.g. -6.2, 0.0, 546.0, 0.657652, etc. Variables must be declared and initialised as floats in the following way .

```c
#include<stdio.h>

int main(int argc, char **argv)
{

float x,y,z;
x=-6.6;
y=5.301;
z=99.1231;

}
```

Data Types: Characters

Character variables are able to store any ASCII character, e.g. ‘a’, ‘b’, ‘f’, etc. Note that a variable declared as a character can only store one character. Variables must be declared and initialised as characters in the following way.

```c
#include<stdio.h>

int main(int argc, char **argv)
{

char x,y,z;
x='M';
y='\n';
z='5';

}
```

Note in the example above that y is equal to the new line character in ASCII. This is a valid definition, even though it looks like the variable y is given two characters. This is due to the special nature of the backslash when dealing with characters. In the following line, z is equal to the character 5 – which is different to the integer 5.

Data Types: Arrays

Multi-dimensional arrays can be declared in C. The important thing to remember is that there is a difference between the number of elements in an array as it is declared and the address of the maximum element in that array. This is because you start counting from 0 in C. An example of a single dimensional array is given below (declaration, initialisation and use).

```c
#include<stdio.h>

int main(int argc, char **argv)
{

int intArray[4];

intArray[0]=3;
intArray[1]=5;
intArray[2]=7;
intArray[3]=1;

printf("The second element of the array is %d\n", intArray[1]);

}
```

It is important to note from the above example that the maximum element in the array is “intArray[3]”. If you attempt to access “intArray[4]” you will get an error.

Arrays are one way of dealing with strings. In C a string is an array of characters. It is even more important to note that a string is always terminated with a null character (\0). This can be explicitly written as in the example below.

```
#include<stdio.h>

int main(int argc, char **argv)
{

char string[5];

string[0]='T';
string[1]='e';
string[2]='s';
string[3]='t';
string[4]='\0';

printf("The elements in thr string are %s\n", string);

}
```

Pointers

A special type of variable in C is a pointer. Pointers are typically not data types that are available in any other language and hence some students have had very little exposure to the concept. If you are familiar with symbolic links in Linux then it may be easy to view pointers in C as a type of mechanism of achieving symbolic links.

Let's start with how to declare a pointer. Pointers are declared using the * character before the variable name, as shown below.

```
#include<stdio.h>
int main(int argc, char **argv)
{
char *x;

}
```

In the example above the variable 'x' holds a memory address that points to an area of memory reserved for a character. When thinking about pointers it is important to remember that with

every assignment of a variable we have two values, the variable name and the value. Let's take the following example;

int x = 9
We have the variable 'x' (which is commonly referred to as the lvalue) and we have the value '9' (which is commonly referred to as the rvalue). Clearly 'x' must in some way "point" to a memory address that is storing 9. If that is the case then surely we should be able to manipulate this pointing in some way (let's say to have more than one variable pointer to the same value). That is what a pointer can do.

Let's take the above example and declare a variable to store '9' and then use a special pointer operator to create a pointer that points to the same memory location (we will use what is referred to as the unary operator '&').

```
#include<stdio.h>
int main(int argc, char **argv)
{
int *x, y;
y=9;
x=&y;

}
```

In the above example the unary operator '&' is used to store the memory address of 'y' in the pointer 'x'. However, we cannot see anything working here. What we need to do is add some code that will print out the value at 'y' and the value that 'x' points to. In order to do this we are going to use some format specifiers in our printf() function – format specifiers specify the type of variable to print out. The format specifier that we are going to use is the "%d" or the integer specifier. We are also going to have to dereference our 'x' pointer. We can dereference our pointer by using '*' in front of our variable, as shown below.

```
#include<stdio.h>
int main(int argc, char **argv)
{
int *x, y;
y=9;
x=&y;

printf("x = %d and y = %d\n", *x, y);
y=10;
printf("x = %d and y = %d\n", *x, y);

}
```

If the above code is compiled and run, you will notice that both the value stored at 'y' and at 'x' is changed to 10 (even though we only change the value of 'y'). This is because we have made the pointer 'x' point to the memory location of 'y'.

Pointers make C a very powerful language to use (because we can directly manipulate memory and the information that is sent through device drivers). However, with this great power comes the risk of errors in memory management – leading to vulnerabilities existing in the code.

The last thing that we are going to note about pointers is the way that they are typically used in programs to perform run-time allocation of memory. Let's say that we don't know the size of the input that a user will make so we cannot predefine how large an array will be to store the user input. At run-time we might count the number of integer values input, say, and at run time allocate the appropriate size of memory. If we do this then we must also free() the memory up when we don't want to use it any more. Incorrect logic associated with this process leads to use-after-free vulnerabilities and also vulnerabilities associated with the use of the free() function. In order to use malloc() and free() we have to add the stdlib.h to our include directives.

```
#include<stdio.h>
#include<stdlib.h>

int main(int argc, char **argv)
{
int *input;
input = malloc(sizeof(int)*atoi(argv[1]));
input[0]=5;
input[1]=10;
input[2]=15;

free(input);

}
```

With the above code we now introduce errors into the program code at run time. If the user doesn't specify the number of integer values to hold then the program will crash because the initialisation of input[0] fails (as input hasn't been allocated any memory).

Conditional Statements: if

The conditional statement is very similar to almost all other programming languages. The if statement can test the truth of something and execute some code if the logic is true. The example below shows this:

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

int main(int argc, char **argv)
{
        int x;
        x=6;

        if (x==6){
        printf("The value is 6\n");
        }
        else{
        printf("Something went wrong!\n");
        }

}
```

You should note the use of the curly braces to denote that program code that should be executed after the evaluation of logic (as well as the double equals sign used by the logic in the if statement).

Looping Construct: for

The for looping construct requires a variable (or more than one variable) to be initialised, it requires the looping condition and it requires the action that should be taken at the end of each iteration.

In the example below we are looping on the variable x, which is initialised to 0. We will continue looping as long as x<5 and at the end of each iteration x = x+1.

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

int main(int argc, char **argv)
{
        int x;

        for(x=0;x<5;x++){
        printf("x is %d\n", x);
        }

}
```

Looping Condition: while

Another looping condition is the while loop. There are many different ways that this construct can be used and I am only going to detail one way here. In the example below I initialise x to 0, I loop based on x<5 and at the end of each iteration I add one to x (i.e. it does the same as the for example above).

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

int main(int argc, char **argv)
{
        int x=0;

        while(x<5){
        printf("x is %d\n", x);
        x++;
        }

}
```

Functions

So far we have only used logic that it contained with the main() function of our C program. Suppose we wanted to make some logic self contained within its own function (for example how to calculate the area contained by a square). There are a couple of ways of declaring functions, one way is to just write the code before the main function, other ways include using function prototypes and including the code in separate files (that can then be joined when we "make" our application).

The program shown below is the most basic use of a function possible. The function takes in as an input an integer value (locally called 'x') and it returns an integer value (and the calculation for the return value is x*x).

```c
#include<stdio.h>
int square(int x){
        return x*x;
}

int main(int argc, char **argv)
{
        int x, square_x;
        x=5;
        square_x=square(x);

        printf("The square of %d is %d\n", x, square_x);

}
```

Task 1 (optional)

1. Write a C program that will take an input from a user and reply with whether or not the input is a prime number.
2. Compile all the examples provided within the practical.