A Short Tutorial on C

(Last Updated 11/3/2019)

Structure of a C program

Probably the best way to start learning a programming language is by writing a program. Therefore, here is our first program:

```
// my first program in C
#include <stdio.h>
int main (void)
{
  printf("Hello World!");
  return 0;
}
```

Notice that each C code must include some pre-process directives or header files (e.g., stdio, stdlib, etc.) which allow us to use certain commands (or functions) in the code. We also must have a main function which is always the first function executed. Each C code line/instruction ends with ";"

How do you Compile and Run the program

You can write the above code in a file and save that as "hello.c". You can then open the file in any editor (Notepad, etc.) and then compile it using a C compiler (like gcc). The easiest way is to download and install something like DevCpp on your machine and then use it to open, compile and run the program. Alternately you can transfer (or even create from scratch) the file to Linux server and compile it (by using *gcc hello.c –o hello*) and run it (by using *./hello*).

Variables, Data Types

C allows you to define a variety of variables including integers, float, characters, strings, etc. to store data of different kinds. Integers are used to store natural numbers like 1, 2, 500, etc. and are declared using keyword, "int". Similarly you can use "double" to define floating point numbers like 0.34, 122.234, etc. Moreover, "char" defines a character variable (example: 'Q') and "string" defines a string like, "Hello World". Each variable must be defined before it is used.

The code below defines few variables, initializes them and displays them:

```
// my second program in C
#include <stdio.h.>
int main (void)
{
   int this_is_a_number;
   printf( "Please enter a number: " );
   scanf( "%d", &this_is_a_number );
   printf( "You entered %d \n", this_is_a_number );
   printf( "The decimal value is %d which is also %p in hexadecimal \n", this_is_a_number, this_is_a_number );
   printf( "Integer variable is %d bytes long ", sizeof(this_is_a_number) );
   printf( "and is stored at memory address %p", &this_is_a_number );
   getchar();
   return 0;
}
```

Getting an input from Keyboard and displaying it on Console /Screen

C allows you to get a user input from the keyboard and display data on screen. Any input can be stored in a variable as explained above. These variable values can then be displayed on console. "*scanf*" is generally used to receive an input from the user and "*printf*" is used to display the output. The code below initializes variables of 4 different kinds and then asks the user to provide values for each. The values provided by user are then displayed.

```
// my third program in C
#include <stdio.h>

int main ()
{
  int a; float b; char c;
  printf( "Please enter one integer value and press enter: " );
  scanf( "%d", &a );
  printf( "Please enter one floating point value and press enter");
  scanf( "%f", &b );
  printf( "Please enter one character value and press enter");
  scanf( " %c", &c );
  printf(  "\nYou entered: %d, %f, %c\n", a, b, c);
  return 0;
}
```

Few things to notice: you can't declare a string (e.g., *string str;*) because that doesn't exist in C without using a library hence we use a character array. Also, notice an extra space for *scanf* for the character, which is there to discard the extra whitespace (i.e., a space or enter key you entered) from prior inputs.

Apart from variables, other text as well as special characters can be displayed on console too. For example "\n" will allow you to use "end of line" (or move cursor to new line), "\t" prints a tab, "\\" prints back slash, and " \" " prints quotation marks.

Performing different Operations on Variables

C allows you to perform different operations on variables. For example, you can assign a value to a variable (i.e., *int a=5;*) or update its value. Similarly you could add two integers by using "+" (for example, *int c= a+b*). Other operations include subtraction ("-"), multiplication ("*"), division ("/") and so on.

C also allows the shorthand version of these instructions. For example, "*a=a+b*" can also be written as "*a+=b*" using shorthand notation. Other possibilities are "-=", "*=", "/=" and so on.

C also allows you to check if two variable values are same (or equal), greater, or smaller. To check if two variables have same value, you would use "==" (two equal signs). Similarly ">=" is greater than or equal and "<=" is used for less than equal. These are in addition to normal "greater (">") or smaller ("<") or not equal ("!=") operators. The answers to all these checks or conditional questions will be *true* or *false* which is also sometimes represented using 1 (*true*) and 0 (*false*). For example, to check to see if integer *a* is equal to 5 we will use (*a* == 5). Similarly, *!true* will results in *false*.

C allows several bitwise operations as well which include AND (&), OR (|), XOR (^), NOT (!), SHIFT (>>, <<), etc. The key to understanding these bitwise operations is to convert the value of the variables in binary and then perform these operations for each bit individually. Typical compilers use 32-bits to represent any number.

Control Structures / Conditional Statements

C also allows you to control the flow of the code using conditional statements. These include "*if*", "*else*", "*else if*", "*switch*", etc.

The first such statement is "*if*" statement. The "*if*" statement helps you decide which block of code to execute if the condition is *true* or *false*. If the condition is *true* then the next line after "*if*" statement is executed, otherwise it is skipped. For example, if you wanted to check the current temperature and display the information, here is how you will use the C statements:

```
// my fourth program in C
#include <stdio.h>

int main ()
{
  int temp = 81;
  if (temp < 90)
     printf("The current temperature is less than 90");
  return 0;
}
```

Notice that if the condition in the "*if*" statement is not *true* then the "*printf*" will not be executed.

A better approach usually is to use "*if*" statement along with "*else*" statement. This allows us to clearly specify which block to be executed when "*if*" condition is *true* and which block to execute when "*if*" condition is *false* (or "*else*" is selected). Sometimes when you specifically need to check for multiple conditions, you can either use them all in any "*if*" statement or you can use an "*else if*" after the first "*if*" statement. For example, let's look at the code below which uses "*if*", "*else if*" and "*else*".

```
// my fourth program in C
#include <stdio.h>
int main ()
{
  int temp = 81;
  if (temp < 90)
     printf( "The current temperature is less than 90");
  elseif (temp < 80)
     printf( "The current temperature is less than 80");
  else
     printf( "The current temperature is greater than 80 and less than 90");

  return 0;
}
```

I would strongly encourage you to use brackets { } to identify the block within any "*if*" and "*else*" statement even when they are only one line code.

Another possible way to control the flow of control is by using a "*switch*" statement. Switch statement checks the condition and based on the results, several cases may be selected. For example, the code below checks if the grade is 'A' or 'B' or any other character and then prints the appropriate statement indicating the value by choosing the corresponding "*case*" statement.

You can think of switch-case statement similar to if-else when used with == comparisons.

```
// my fifth program in C
#include <stdio.h>

int main ()
{
  char grade = 'A';
  switch (grade)
  {
   case 'a':
   case 'A':
     printf ( "The current temperature is equal to 90");
     break;
  case 'B':
     printf ( "The current temperature is equal to 80");
     break
  default:
     printf ("The current temperature is neither 90 nor 80 degrees");
     break;
  }
  return 0;
}
```

Notice that *break* statement must be used to end each *case* and you should also include a *default* case. Failure to use *break* statement will allow code to continue executing until next *break* statement is found of *switch* statement ends. (Notice in the above example: when user enters 'a' or 'A', the first *printf* statement will be executed). The *break* statement can also be used to stop the conditional and iterative executions.

C also allows few more advanced conditional operations such as AND, OR, XOR, etc. AND operation checks two values and gives a result of *true* or *false* if both values are *true*. Similarly OR operation will give a results of *true* if any or both of two input variables are *true*. C instruction for AND operation is "&&" while OR can be performed using "||".

Iterative Structures / Loops or Repetitions

When you want to execute some code multiple times, C allows you to iterate using several instructions including "*while*", "*do-while*" and "*for*" loops. For example if you want to print your name 10 times (repeatedly), we can do that by using any of the iterative statements.

```
// my sixth program in C
#include <stdio.h>
int main ()
{
  int count = 0;
  while (count <= 10) {
     printf( "My name is Robert!\n");
     count++;
  }
  return 0;
}
```

Notice that if the conditional statement inside "*while*" stays *true*, the block of code underneath "*while*" keeps on executing. Once the execution of the block is done, the "*while*" loop checks the condition again and keeps on iterating (or repeating) until the condition stays *true*. That is why the condition is usually

updated inside the block, just like the count variable is being updated. The only way to stop the "*while*" loop is to ensure that the condition becomes *false*.

Similarly we can run the same code using "*for*" statement, which allows you to iterate in a slightly different manner. The above code can now be re-written using "*for*" loop as below:

```
// my seventh program in C
#include <stdio.h>

int main ()
{
    for (int count = 0; count <= 10; count++) {
        printf( "My name is Robert!\n");
  }
  return 0;
}
```

Notice that the initialization of the condition variable, the conditional check as well as updating is all done inside the "*for*" loop statement.

Another iterative approach is using "*do-while*" statement. It is slightly different that "*while*" statement.

```
// my eighth program in C
#include <stdio.h>

int main ()
{
  int count = 0;
  do{
    printf( "My name is Robert!\n");
    count++;
  } while (count <= 10);

  return 0;
}
```

Notice that the "*while*" statement **first** checks the condition and based on the condition being *true* or *false* it decides which block of code underneath will be executed. It then repeats the process. On the other hand, "*do-while*" statement **first** executes the block of code above "*while*" (and below "*do*" statement) ONCE and then checks the condition. If the condition is *true*, the block of code inside "*do*" keeps on repeating until the condition stays *true*. The updated version of previous "*while*" code is given above in the form of "*do-while*" loop.

When writing loops be sure to ensure that it doesn't turn into an infinite loop. If you need an infinite loop, you can use while(1); or while(1){;} in your code which is same as for(;;); or for (;;){;}

Functions

C allows us to separate code into different functions where each function can be written to do any specific job. Recall that to use any variable, you have to declare the variable, define / initialize its value and then use the variable. Similarly, a function needs to be declared, defined and then called (or used). The declaration of the function often is done at the top of the code (usually above *main*) and is needed to let the compiler know that the code will use a function of given name. The definition of the function can be written

above *main* or below *main*, which is essentially specifying what the function is supposed to do. Finally within your code, you will need to call (or use) the function in order to run it.

Here is an example of simple function which is just adding two numbers together and displaying their result.

```
// my ninth program in C
#include <stdio.h>

int myAddFunction( ); // function declaration also known as header

int main ()
{
  printf("Welcome to my 9th Function\n");
  myAddFunction(); // This is my function call
  return 0;
}

//Function Definition
int myAddFunction( )
{
  double a=5.0, b=6.1;
  printf( "SUM: %f", a+b);
  return 1;
}
```

Notice that the name of the function is myAddFunction and all functions use parenthesis.

Return Values and Arguments

The function can receive an input and can also provide an output. The values that are given as input to the function are called arguments and are written in parenthesis. The function can return a value (like an output) which will be written just before the function name. In our last example (9th program), the function was returning an integer value.

To return a value from the function, we use the *return* statement before the closing brace of the function.

The arguments are written inside the parenthesis and are passed from the code which had called our function. These arguments then become local variables for the function are can be used inside its definition. The scope of a variable is typically the area where the variable can be accessible. Typically, a set of braces ({ }) are used to limit the scope. C allows declaring global variables which can be defined outside main function. They will have file scope which means you can access the global variables any place in the file.

When you call a function with arguments, the calling function must provide a value for the argument which is then received by called function. This is known as *pass by value.* C uses *pass by value* for all arguments except array arguments. Array arguments are passed by reference. C also has *pass by pointer* which is similar to *pass by reference.*

The function call must ensure that the argument list matches the function prototype and definition.

Let's rewrite our last function in a way that the input values are now defined inside main functions and then passed to the function as arguments. The function is responsible for adding those and then returning the answer which is then displayed in main code. Here is the new version of our function.

```
// my tenth program in C
#include <stdio.h>

double myAddFunction (double a, double b ); // function declaration also known as header

int main ()
{
  printf( "Welcome to my 10th Function\n");
  double a=5, b=6, c;
  printf( "Var Values a %d, b %d, addresses: a %p, b %p", a, b, &a, &b);
  c = myAddFunction (a,b); // This is my function call
  printf( "SUM:" %f, c);

  return 0;
}

//Function Definition
double myAddFunction( double a, double b)
{
  double sum = a+b;
  printf( "Var Values a %d, b %d, addresses: a %p, b %p", a, b, &a, &b);
 return sum;
}
```

The above example shows two versions of *a* and *b* variables (one declared inside main function and the other inside the function). Their memory addresses confirm the fact that they are defined in different scopes.

Each time the function is called, local variables defined inside the function are re-created. They are automatically deleted when the function ends or when variables go out of scope. If you don't want these variables re-created on each function execution, you can use *static* keyword. When using *pass by reference* or *pass by pointer*, function accesses the variables defined in calling function and hence can modify the original variables. If arguments are passed with *const* keyword, function is not allowed to modify values of passed arguments.

Introduction to Pointers

Pointers is one of the most important concepts in C. Let's start with a variable before we discuss pointers.

```
// my eleventh program in C
#include <stdio.h>

int main ()
{
  int var = 10;
  printf( "var has the value: %d stored at memory address: %p\n", var, &var);
  printf( "var has the type \'int\' and needs %d bytes of storage \n", sizeof(var));

  return 0;
}
```

The code above declared an integer variable (type is *int*), whose name is *var* and takes 4 bytes of storage (size), stored at memory address that can be found by using *&var*. *The important point to remember is that* ***variables are containers for values*** *and their contents are values*.

You can think of a ***pointer as a container for memory addresses***. Pointers are variables too. When you are declaring a pointer, you are basically creating a variable which is supposed to contain (or store) a memory address. We try to match the type of the pointer with the type of the address that it is supposed to store (i.e., if you have an integer pointer, it is supposed to contain an address of an integer variable)

```
// my twelveth program in C
#include <stdio.h>

int main ()
{
  int var = 10;
  int* ptr_var; /* other declarations: //int *ptr_var; //int*ptr_var; //int * ptr_var; /*
  ptr_var = &var;
  /* Note: you can combine two lines as well: // int* ptr_var = &var; */

  printf( "var has the value: %d stored at memory address: %p\n", var, &var);
  printf( "ptr_var is a pointer that contains var\'s memory address and the value of var can be accessed through the pointer by dereferencing the pointer, which is %d \n", *ptr_var);

  return 0;
}
```

Notice that value stored inside the pointer is an address of another variable (hence the contents of a pointer variable are a memory address of another variable). That another variable may contain the actual value. In order to access that value using the pointer variable, you will first have to find out the address of the variable and then access that address to get to the value. This process is called *dereferencing* and is some by using * operator with the pointer variable (**ptr_var*). You should also not forget that the pointer variable is itself a variable and hence it will be stored at some memory location as well. In order to know where the pointer variable is stored, you will need to use & operator again (e.g., *&ptr_var*).

In summary, *&ptr_var* is the memory address where a pointer variable was stored when we created it. At that memory address you stored address of another variable *var* (which could be printed using *&var*). You will find the actual integer value stored at that memory address.

Pointer Arithmetic

Each variable has a type which closely relates to the amount of memory needed for storing that variable. For example, on a 32-bit computer, an *integer* variable needs 4 bytes, a *character* variable needs 1 byte, a *short* variable needs 2 bytes and *double* needs 8 bytes of memory. You can find the required memory size by using *sizeof* function defined in <*stdlib.h*>. Each pointer variable has a type which matches the type of pointer variable in which memory address of that variable can be stored.

When you increment an integer variable (e.g., *var += 1;*), you increment the value of the integer by 1. When you increment a pointer variable, the address stored in that pointer variable is incremented. *The increment of an address is NOT same as an increment of a value*. When you add 1 to an address you ***move*** your address to the next variable of ***that*** type. For example, incrementing an integer pointer moves the address to next integer variable (which is found 4 bytes after the previous integer). Similarly incrementing a character pointer moves the address to next character variable (which is found 1 byte after the previous character). Hence an increment of a pointer variable (or a memory address) depends on the type of the pointer. ***A single increment of the pointer (e.g., ptr_var++) will change the address of an integer pointer by 4, address of a double pointer by 8 and an address of a character pointer by 1 byte***.

```
// my thirteenth program in C
#include <stdio.h>

int main ()
{
  int varI = 1; char varC = 'a'; double varD=1.0;
  int* ptr_varI; char* ptr_varC; double* ptr_varD;
  ptr_varI = &varI; ptr_varC = &varC; ptr_varD = &varD;
  printf( "Contents of (or Values stored in) the pointer Variables:\n");
  printf( "Integer: %p, Char: %p, Double: %p \n", ptr_varI, ptr_varC, ptr_varD);
  ptr_varI ++; ptr_varC ++; ptr_varD ++;
  printf( "Contents of (or Values stored in) the pointer Variables (Updated):\n");
  printf( "Integer: %p, Char: %p, Double: %p \n", ptr_varI, ptr_varC, ptr_varD);

  return 0;
}
```

Introduction to Dynamic Memory Allocation

Most programs allocate variables and arrays statically. If you want to allocate memory at run time (or dynamically), you will use *malloc* (or *calloc*) functions which will reserve some memory from *heap* and return the starting address to you. You can deallocate that memory using *free* function. All of these are declared in <*stdlib.h*>

```
// my fourteenth program in C
#include <stdio.h>
#include<stdlib.h>

int main ()
{
  int* ptr_int = malloc (1*sizeof(int));
  int* ptr_int_array = malloc (5 * sizeof(int)); //5 integers
  *ptr_int = 11;
  for (int i=0; i<5;i++) {
   ptr_int_array[i]=i;
  }
  for (int i=0; i<5;i++) {
   printf("%d ", *(ptr_int_array+i)); /* Can also use // print("%d ", ptr_int_array[i]); */
  }
  printf("\n%d ", *ptr_int);
  free(ptr_int); free(ptr_int_array);
  return 0;
}
```

You can also create a double pointer where a pointer contains an address of another pointer which will contain the address where the value is stored. You will need to use dereferencing twice to access values from a double pointer.