Unit-III

What is a Function

A function is a self-contained block of statements that perform a coherent task of some kind. Every C program can be thought of as a collection of these functions.

```
main( )
{
    message( ) ;
    printf ( "\nCry, and you stop the monotony!" ) ;
}
```

Here, **main( )** itself is a function and through it we are calling the function **message( )**. What do we mean when we say that **main( )** 'calls' the function **message( )**? We mean that the control passes tothe function **message( )**. The activity of **main( )** is temporarily suspended; it falls asleep while the **message( )** function wakes up and goes to work.

From this program a number of conclusions can be drawn:

- Any C program contains at least one function.
- If a program contains only one function, it must be **main( )**.
- If a C program contains more than one function, then one (and only one) of these functions must be **main( )**, because programexecution always begins with **main( )**.
- There is no limit on the number of functions that might be present in a C program.
- Each function in a program is called in the sequence specified by the function calls in **main( )**.
- After each function has done its thing, control returns to **main( )**.When **main( )** runs out of function calls, the program ends.

As we have noted earlier the program execution always begins with **main( ).**

Let us now summarize what we have learnt so far.

(a) C program is a collection of one or more functions.

(b) A function gets called when the function name is followed bya semicolon.

(c) A function is defined when function name is followed by a pair of braces in which one or more statements may be present.

(d) A function can be called any number of times.

(e) A function can be called from other function, but a function cannot be defined in another function.

(f) There are basically two types of functions:

Library functions Ex. **printf( )**, **scanf( )** etc.

User-defined functions Ex. **argentina( )**, **brazil( )** etc.

Passing Values between Functions

EX:

```
main( )
{
    int a, b, c, sum ;
    printf ( "\nEnter any three numbers " ) ;scanf ( "%d %d
    %d", &a, &b, &c ) ;
    sum = calsum ( a, b, c ) ;
    printf ( "\nSum = %d", sum ) ;
}

calsum ( x, y, z )
int  x, y, z ;
{
    int  d ;

    d = x + y + z ;return ( d ) ;
}
```

1. In this program, from the function **main( )** the values of **a**, **b** and **c** are passed on to the function **calsum( )**, by making acall to the function **calsum( )** and mentioning **a**, **b** and **c** in the parentheses:
2. The variables **a**, **b** and **c** are called 'actual arguments', whereas the variables **x**, **y** and **z** are called 'formal arguments'.
3. There are two methods of declaring the formal arguments.The one that we have used in our program is known as Kernighan and Ritchie (or just K & R) method.

```
calsum ( x, y, z )
int  x, y, z ;
```

Another method is,

```
calsum ( int x, int  y, int  z )
```

This method is called ANSI method and is more commonly used these days.

The **return** statement serves two purposes:

On executing the **return** statement it immediatelytransfers the control back to the calling program.

It returns the value present in the parentheses after **return**, to the calling program. In the above programthe value of sum of three numbers is being returned.

There is no restriction on the number of **return** statements that may be present in a function. Also, the **return** statement need not always be present at the end of the called function.

There is no restriction on the number of **return** statementsthat may be present in a function.

Scope Rule of Functions

Function scope	Function scope begins at the opening of the function and ends with the closing of it. Function scope is applicable to labels only. A label declared is used as a target to goto statement and both goto and label statement must be in same function

Calling Convention

Calling convention indicates the order in which arguments arepassed to a function when a function call is encountered. There aretwo possibilities here:

(a) Arguments might be passed from left to right.
(b) Arguments might be passed from right to left.

C language follows the second order.

Advanced Features of Functions

(a) Function Declaration and Prototypes
(b) Calling functions by value or by reference
(c) Recursion

1. Function Declaration and Prototypes

A function prototype is simply the declaration of a function that specifies function's name, parameters and return type. It doesn't contain function body. A function prototype gives information to the compiler that the function may later be used in the program.

A function declaration is any form of line declaring a function and ending with ; .

A prototype is a function declaration where all types of the parameters are specified.

For example,

consider the following code,

```
int add(int, int);
```

Here, a function named add is declared with 2 arguments of type int and return type int.

Call by Value and Call by Reference

Call by value: A copy of the variable is passed to the function.

Call by reference: An address of the variable is passed to the function.

Call by Value means calling a method with a parameter as value. Through this, the argument value is passed to the parameter.

While Call by Reference means calling a method with a parameter as a reference. Through this, the argument reference is passed to the parameter.

Pointers

An Introduction to Pointers

A **Pointer in C** language is a variable that holds a memory address. This memory address is the address of another variable(mostly) of same data type.

Pointer Notation

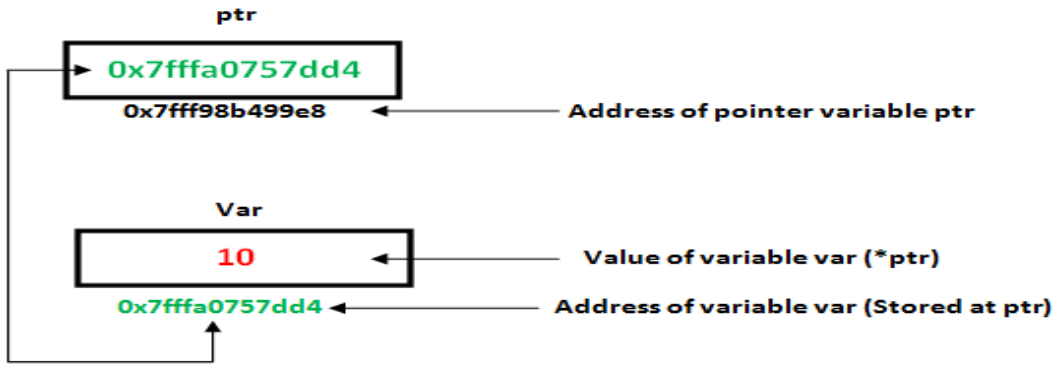
Consider the declaration,

```
int  var= 10 ;
```

This declaration tells the C compiler to:

- Reserve space in memory to hold the integer value.
- Associate the name **var** with this memory location.
- Store the value 10 at this location.

We may represent **var**'s location in memory by the following memory map



```c
#include<stdio.h>



void main()

{

    int var = 7;

    printf("Value of the variable var is: %d\n",
var);

    printf("Memory address of the variable var is:
%x\n", &var);

}
```

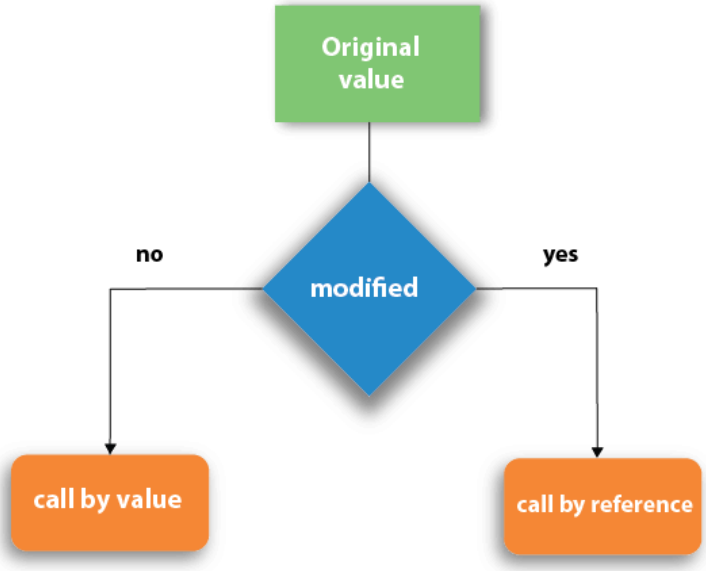
Output:

```
Value of the variable var is: 7
Memory address of the variable var is: bcc7a00
```

Back to Function Calls

Arguments can generally be passed to functions in one of the two ways:

(a) sending the values of the arguments
(b) sending the addresses of the arguments



- In call by value method, the value of the actual parameters is copied into the formal parameters. In other words, we can say that the value of the variable is used in the function call in the call by value method.
- In call by value method, we can not modify the value of the actual parameter by the formal parameter.
- In call by value, different memory is allocated for actual and formal parameters since the value of the actual parameter is copied into the formal parameter.
- The actual parameter is the argument which is used in the function call whereas formal parameter is the argument which is used in the function definition.

```
#include<stdio.h>
void change(int num)
{
   printf("Before adding value inside function num=%d \n",num);
   num=num+100;
   printf("After adding value inside function num=%d \n", num);
}
int main()
{
   int x=100;
   printf("Before function call x=%d \n", x);
   change(x);//passing value in function
   printf("After function call x=%d \n", x);
return 0;
}
```

Output

```
Before function call x=100
Before adding value inside function num=100
After adding value inside function num=200
After function call x=100
```

Call by reference in C

- In call by reference, the address of the variable is passed into the function call as the actual parameter.
- The value of the actual parameters can be modified by changing the formal parameters since the address of the actual parameters is passed.
- In call by reference, the memory allocation is similar for both formal parameters and actual parameters. All the operations in the function are performed on the value stored at the address of the actual parameters, and the modified value gets stored at the same address.

```
#include<stdio.h>
void change(int *num)
{
   printf("Before adding value inside function num=%d \n",*num);
   (*num) += 100;
   printf("After adding value inside function num=%d \n", *num);
}
int main()
 {
   int x=100;
   printf("Before function call x=%d \n", x);
   change(&x);//passing reference in function
   printf("After function call x=%d \n", x);
return 0;
}
```

Output

```
Before function call x=100
Before adding value inside function num=100
After adding value inside function num=200
After function call x=200
```

Recursion

Recursion is **the technique of making a function call itself**. This technique provides a way to break complicated problems down into simple problems which are easier to solve. Recursion may be a bit difficult to understand. The best way to figure out how it works is to experiment with it.

Ex:

```
void recursion()
{
  recursion(); /* function calls itself */
}

int main() {
  recursion();
}
```

Recursion and Stack

Recursive functions use something called “the call stack.” **When a program calls a function, that function goes on top of the call stack**. This is similar to a stack of books. You add things one at a time. Then, when you are ready to take something off, you always take off the top item.

I will show you the call stack in action with the `factorial` function. `factorial(5)` is written as 5! and it is defined like this: 5! = 5 * 4 * 3 * 2 * 1. Here is a recursive function to calculate the factorial of a number:

```
function fact(x) {
    if (x == 1) {
        return 1;
    } else {
        return x * fact(x-1);
    }
}
```