

Functions in C

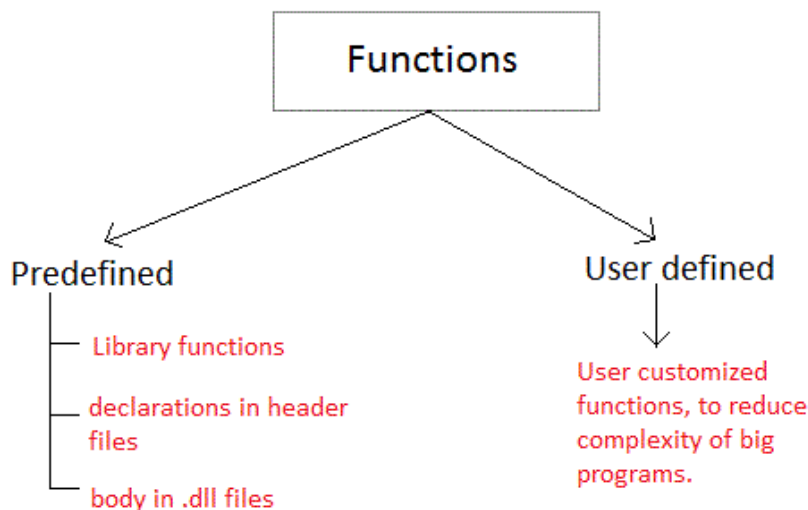
A **function** is a block of code that performs a particular task.

There are many situations where we might need to write same line of code for more than once in a program. This may lead to unnecessary repetition of code, bugs and even becomes boring for the programmer. So, C language provides an approach in which you can declare and define a group of statements once in the form of a function and it can be called and used whenever required.

These functions defined by the user are also known as **User-defined Functions**

C functions can be classified into two categories,

1. **Library functions**
2. **User-defined functions**



Library functions are those functions which are already defined in C library, example `printf()`, `scanf()`, `strcat()` etc. You just need to include appropriate header files to use these functions. These are already declared and defined in C libraries.

A **User-defined functions** on the other hand, are those functions which are defined by the user at the time of writing program. These functions are made for code reusability and for saving time and space.

Benefits of Using Functions

1. It provides modularity to your program's structure.
2. It makes your code reusable. You just have to call the function by its name to use it, wherever required.
3. In case of large programs with thousands of code lines, debugging and editing becomes easier if you use functions.
4. It makes the program more readable and easy to understand.

Function Declaration

General syntax for function declaration is,

```
returntype functionName(type1 parameter1, type2 parameter2,...);
```

Like any variable or an array, a function must also be declared before its used. Function declaration informs the compiler about the function name, parameters is accept, and its return type. The actual body of the function can be defined separately. It's also called as **Function Prototyping**. Function declaration consists of 4 parts.

- returntype
- function name
- parameter list
- terminating semicolon

returntype

When a function is declared to perform some sort of calculation or any operation and is expected to provide with some result at the end, in such cases, a `return` statement is added at the end of function body. Return type specifies the type of value(`int`, `float`, `char`, `double`) that function is expected to return to the program which called the function.

Note: In case your function doesn't return any value, the return type would be `void`.

functionName

Function name is an `identifier` and it specifies the name of the function. The function name is any valid C identifier and therefore must follow the same naming rules like other variables in C language.

parameter list

The parameter list declares the type and number of arguments that the function expects when it is called. Also, the parameters in the parameter list receives the argument values when the function is called. They are often referred as **formal parameters**.

Example

Let's write a simple program with a `main()` function, and a user defined function to multiply two numbers, which will be called from the `main()` function.

```
#include<stdio.h>
int multiply(int a, int b);    // function declaration
main()
{
    int i, j, result;
    printf("Please enter 2 numbers you want to multiply...");
    scanf("%d%d", &i, &j);
    result = multiply(i, j);    // function call
    printf("The result of muliplication is: %d", result);
}
int multiply(int a, int b)
{
    return (a*b);              // function defintion, this can be done in
                                one line
}
```

Function definition Syntax

Just like in the example above, the general syntax of function definition is,

```
returntype functionName(type1 parameter1, type2 parameter2,...)
{
    // function body goes here
}
```

The first line `returntype functionName(type1 parameter1, type2 parameter2,...)` is known as **function header** and the statement(s) within curly braces is called **function body**.

Note: While defining a function, there is no semicolon(;) after the parenthesis in the function header, unlike while declaring the function or calling the function.

functionbody

The function body contains the declarations and the statements(algorithm) necessary for performing the required task. The body is enclosed within curly braces { ... } and consists of three parts.

- **local** variable declaration(if required).
- **function statements** to perform the task inside the function.
- a **return** statement to return the result evaluated by the function(if return type is `void`, then no return statement is required).

Calling a function

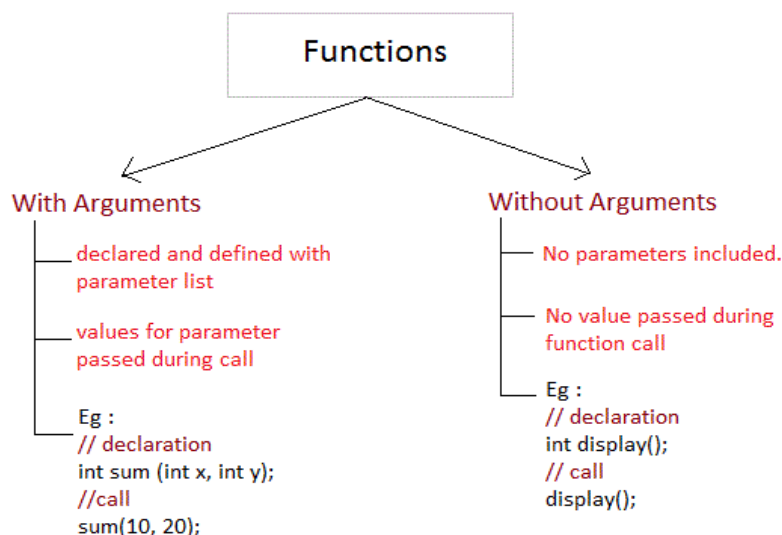
When a function is called, control of the program gets transferred to the function.

```
functionName(argument1, argument2,...);
```

In the example above, the statement `multiply(i, j);` inside the `main()` function is function call.

Passing Arguments to a function

Arguments are the values specified during the function call, for which the formal parameters are declared while defining the function.



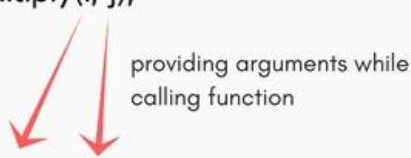
It is possible to have a function with parameters but no return type. It is not necessary, that if a function accepts parameter(s), it must return a result too.

```
#include<stdio.h>

int multiply(int a, int b);

int main()
{
    ... ..
    result = multiply(i, j);
    ... ..
}

int multiply(int a, int b)
{
    ... ..
}
```



providing arguments while calling function

While declaring the function, we have declared two parameters `a` and `b` of type `int`. Therefore, while calling that function, we need to pass two arguments, else we will get compilation error. And the two arguments passed should be received in the function definition, which means that the function header in the function definition should have the two parameters to hold the argument values. These received arguments are also known as **formal parameters**. The name of the variables while declaring, calling and defining a function can be different.

Returning a value from function


A function may or may not return a result. But if it does, we must use the `return` statement to output the result. `return` statement also ends the function execution, hence it must be the last statement of any function. If you write any statement after the `return` statement, it won't be executed.

```
#include<stdio.h>

int multiply(int a, int b);

int main()
{
    ... ..
    result = multiply(i, j);
    ... ..
}

int multiply(int a, int b)
{
    ... ..
    return a*b;
}
```



The value returned by the function must be stored in a variable.

The datatype of the value returned using the `return` statement should be same as the return type mentioned at function declaration and definition. If any of it mismatches, you will get compilation error.

In the next tutorial, we will learn about the different types of user defined functions in C language and the concept of Nesting of functions which is used in recursion

Type of User-defined Functions in C

There can be 4 different types of user-defined functions, they are:

1. Function with no arguments and no return value
2. Function with no arguments and a return value
3. Function with arguments and no return value
4. Function with arguments and a return value

Below, we will discuss about all these types, along with program examples.

Function with no arguments and no return value

Such functions can either be used to display information or they are completely dependent on user inputs.

Below is an example of a function, which takes 2 numbers as input from user, and display which is the greater number.

```
#include<stdio.h>
void greatNum();           // function declaration
int main()
{
    greatNum();           // function call
    return 0;
}
void greatNum()           // function definition
{
    int i, j;
    printf("Enter 2 numbers that you want to compare...");
    scanf("%d%d", &i, &j);
    if(i > j)
        printf("The greater number is: %d", i);
    else
        printf("The greater number is: %d", j);
}
```

Function with no arguments and a return value

We have modified the above example to make the function `greatNum()` return the number which is greater amongst the 2 input numbers.

```
#include<stdio.h>
int greatNum();           // function declaration
int main()
{
    int result;
    result = greatNum();   // function call
    printf("The greater number is: %d", result);
    return 0;
}
int greatNum()           // function definition
{
```

```

int i, j, greaterNum;
printf("Enter 2 numbers that you want to compare...");
scanf("%d%d", &i, &j);
if(i > j) {
    greaterNum = i;
}
else {
    greaterNum = j;
}
// returning the result
return greaterNum;
}

```

Function with arguments and no return value

We are using the same function as example again and again, to demonstrate that to solve a problem there can be many different ways.

This time, we have modified the above example to make the function `greatNum()` take two `int` values as arguments, but it will not be returning anything.

```

#include<stdio.h>
void greatNum(int a, int b);          // function declaration
int main()
{
    int i, j;
    printf("Enter 2 numbers that you want to compare...");
    scanf("%d%d", &i, &j);
    greatNum(i, j);                  // function call
    return 0;
}
void greatNum(int x, int y)           // function definition
{
    if(x > y) {
        printf("The greater number is: %d", x);
    }
    else {
        printf("The greater number is: %d", y);
    }
}

```

Function with arguments and a return value

This is the best type, as this makes the function completely independent of inputs and outputs, and only the logic is defined inside the function body.

```

#include<stdio.h>
int greatNum(int a, int b);          // function declaration
int main()
{
    int i, j, result;
    printf("Enter 2 numbers that you want to compare...");
    scanf("%d%d", &i, &j);
    result = greatNum(i, j); // function call
    printf("The greater number is: %d", result);
    return 0;
}

```

```

}
int greatNum(int x, int y)           // function definition
{
    if(x > y) {
        return x;
    }
    else {
        return y;
    }
}

```

Nesting of Functions

C language also allows nesting of functions i.e to use/call one function inside another function's body. We must be careful while using nested functions, because it may lead to infinite nesting.

```

function1()
{
    // function1 body here

    function2();

    // function1 body here
}

```

If function2() also has a call for function1() inside it, then in that case, it will lead to an infinite nesting. They will keep calling each other and the program will never terminate.

Not able to understand? Lets consider that inside the `main()` function, function1() is called and its execution starts, then inside function1(), we have a call for function2(), so the control of program will go to the function2(). But as function2() also has a call to function1() in its body, it will call function1(), which will again call function2(), and this will go on for infinite times, until you forcefully exit from program execution.

What is Recursion?

Recursion is a special way of nesting functions, where a function calls itself inside it. We must have certain conditions in the function to break out of the recursion, otherwise recursion will occur infinite times.

```

function1()
{
    // function1 body
    function1();
    // function1 body
}

```

Example: Factorial of a number using Recursion

```

#include<stdio.h>
int factorial(int x);           //declaring the function
void main()
{
    int a, b;

```

```

    printf("Enter a number...");
    scanf("%d", &a);
    b = factorial(a);           //calling the function named
factorial
    printf("%d", b);
}

int factorial(int x) //defining the function
{
    int r = 1;
    if(x == 1)
        return 1;
    else
        r = x*factorial(x-1);   //recursion, since the
function calls itself

    return r;
}

```

Similarly, there are many more applications of recursion in C language. Go to the programs section, to find out more programs using recursion.

Types of Function calls in C

Functions are called by their names, we all know that, then what is this tutorial for? Well if the function does not have any arguments, then to call a function you can directly use its name. But for functions with arguments, we can call a function in two different ways, based on how we specify the arguments, and these two ways are:

1. Call by Value
2. Call by Reference

Call by Value

Calling a function by value means, we pass the values of the arguments which are stored or copied into the formal parameters of the function. Hence, the original values are unchanged only the parameters inside the function changes.

```

#include<stdio.h>

void calc(int x);

int main()
{
    int x = 10;
    calc(x);
    // this will print the value of 'x'
    printf("\nvalue of x in main is %d", x);
    return 0;
}

void calc(int x)
{
    // changing the value of 'x'
    x = x + 10 ;
}

```



```
printf("value of x in calc function is %d ", x);  
}
```

value of x in calc function is 20

value of x in main is 10

In this case, the actual variable `x` is not changed. This is because we are passing the argument by value, hence a copy of `x` is passed to the function, which is updated during function execution, and that copied value in the function is destroyed when the function ends(goes out of scope). So the variable `x` inside the `main()` function is never changed and hence, still holds a value of `10`.

But we can change this program to let the function modify the original `x` variable, by making the function `calc()` return a value, and storing that value in `x`.

```
#include<stdio.h>  
int calc(int x);  
int main()  
{  
    int x = 10;  
    x = calc(x);  
    printf("value of x is %d", x);  
    return 0;  
}  
int calc(int x)  
{  
    x = x + 10 ;  
    return x;  
}
```

value of x is 20

Call by Reference

In call by reference we pass the address(reference) of a variable as argument to any function. When we pass the address of any variable as argument, then the function will have access to our variable, as it now knows where it is stored and hence can easily update its value.

In this case the formal parameter can be taken as a **reference** or a **pointer**(don't worry about pointers, we will soon learn about them), in both the cases they will change the values of the original variable.

```
#include<stdio.h>  
void calc(int *p);          // functin taking pointer as argument  
int main()  
{  
    int x = 10;  
    calc(&x);               // passing address of 'x' as argument  
    printf("value of x is %d", x);  
    return(0);  
}  
void calc(int *p)           //receiving the address in a reference  
pointer variable
```

```
{  
    /*  
        changing the value directly that is  
        stored at the address passed  
    */  
    *p = *p + 10;  
}
```

value of x is 20

NOTE: If you do not have any prior knowledge of pointers, do study [Pointers](#) first. Or just go over this topic and come back again to revise this, once you have learned about pointers.