Programming in C

(Discuss Function with example)

Prepared By

Alok Haldar

Assistant professor

Department of Computer Science & BCA

Functions in C Programming with examples

A function is a **block of statements** that performs a specific task. Let's say you are writing a C program and you need to perform a same task in that program more than once. In such case you have two options:

a) Use the same set of statements every time you want to perform the task
b) Create a function to perform that task, and just call it every time you need to perform that task.

Using option (b) is a good practice and a **good programmer always uses functions while writing code in C**.

Why we need functions in C

Functions are used because of following reasons –
a) To improve the readability of code.
b) Improves the reusability of the code, same function can be used in any program rather than writing the same code from scratch.
c) Debugging of the code would be easier if you use functions, as errors are easy to be traced.
d) Reduces the size of the code, duplicate set of statements are replaced by function calls.

Types of functions

1) Predefined standard library functions

Standard library functions are also known as **built-in functions**. Functions such as puts(), gets(), printf(), scanf() etc are standard library functions. These functions are already defined in header files (files with .h extensions are called header files such as stdio.h), so we just call them whenever there is a need to use them.

For example, printf() function is defined in <stdio.h> header file so in order to use the printf() function, we need to include the <stdio.h> header file in our program using #include <stdio.h>.

2) User Defined functions

The functions that we create in a program are known as user defined functions or in other words you can say that a function created by user is known as user defined function.

Now we will learn how to create user defined functions and how to use them in C Programming

Syntax of a function

```
return_type function_name (argument list)
{
   Set of statements – Block of code
}
```

return_type: Return type can be of any data type such as int, double, char, void, short etc. Don't worry you will understand these terms better once you go through the examples below.

function_name: It can be anything, however it is advised to have a meaningful name for the functions so that it would be easy to understand the purpose of function just by seeing it's name.

argument list: Argument list contains variables names along with their data types. These arguments are kind of inputs for the function. For example – A function which is used to add two integer variables, will be having two integer argument.

Example

Given below is the source code for a function called **max()**. This function takes two parameters num1 and num2 and returns the maximum value between the two −

```
/* function returning the max between two numbers */
int max(int num1, int num2) {

   /* local variable declaration */
   int result;

   if (num1 > num2)
      result = num1;
   else
      result = num2;

   return result;
}
```

Function Declarations

A function **declaration** tells the compiler about a function name and how to call the function. The actual body of the function can be defined separately.

A **function declaration** has the following parts −

return_type function_name(parameter list);

For the above defined function max(), the function declaration is as follows −

int max(int num1, int num2);

Parameter names are not important in function declaration only their type is required, so the following is also a valid declaration −

int max(int, int);

Function declaration is required when you define a function in one source file and you call that function in another file. In such case, you should declare the function at the top of the file calling the function.

Calling a Function

While creating a C function, you give a definition of what the function has to do. To use a function, you will have to call that function to perform the defined task.

When a program calls a function, the program control is transferred to the called function. A called function performs a defined task and when its return statement is executed or when its function-ending closing brace is reached, it returns the program control back to the main program.

To call a function, you simply need to pass the required parameters along with the function name, and if the function returns a value, then you can store the returned value. For example −

```
#include <stdio.h>

/* function declaration */
int max(int num1, int num2);

int main () {

   /* local variable definition */
   int a = 100;
   int b = 200;
   int ret;

   /* calling a function to get max value */
   ret = max(a, b);

   printf( "Max value is : %d\n", ret );

   return 0;
}

/* function returning the max between two numbers */
int max(int num1, int num2) {

   /* local variable declaration */
   int result;

   if (num1 > num2)
      result = num1;
   else
      result = num2;

   return result;
}
```

We have kept max() along with main() and compiled the source code. While running the final executable, it would produce the following result −

```
Max value is : 200
```

How to call a function in C?

Consider the following C program

Example1: Creating a user defined function addition()

```
#include <stdio.h>
int addition(int num1, int num2)
{
    int sum;
    /* Arguments are used here*/
    sum = num1+num2;

    /* Function return type is integer so we are returning
     * an integer value, the sum of the passed numbers.
     */
    return sum;
}

int main()
{
    int var1, var2;
    printf("Enter number 1: ");
    scanf("%d",&var1);
    printf("Enter number 2: ");
    scanf("%d",&var2);

    /* Calling the function here, the function return type
     * is integer so we need an integer variable to hold the
     * returned value of this function.
     */
    int res = addition(var1, var2);
    printf ("Output: %d", res);

    return 0;
}
```

Output:

```
Enter number 1: 100
Enter number 2: 120
Output: 220
```

Example2: Creating a void user defined function that doesn’t return anything

```
#include <stdio.h>
/* function return type is void and it doesn't have parameters*/
void introduction()
{
   printf("Hi\n");
   printf("My name is Chaitanya\n");
   printf("How are you?");
```

```
    /* There is no return statement inside this function, since its
     * return type is void
     */
}

int main()
{
    /*calling function*/
    introduction();
    return 0;
}
```

Output:

```
Hi
My name is Chaitanya
How are you?
```

Few Points to Note regarding functions in C:

1) main() in C program is also a function.

2) Each C program must have at least one function, which is main().

3) There is no limit on number of functions; A C program can have any number of functions.

4) A function can call itself and it is known as "**Recursion**". I have written a separate guide for it.

C Functions Terminologies that you must remember

return type: Data type of returned value. It can be void also, in such case function doesn't return any value.

More Topics on Functions in C

1) **Function – Call by value method** – In the call by value method the actual arguments are copied to the formal arguments, hence any operation performed by function on arguments doesn't affect actual parameters.

2) **Function – Call by reference method** – Unlike call by value, in this method, address of actual arguments (or parameters) is passed to the formal parameters, which means any operation performed on formal parameters affects the value of actual parameters.

Function call by value in C programming

Function call by value is the default way of calling a function in C programming. Before we discuss function call by value, lets understand the terminologies that we will use while explaining this:

Actual parameters: The parameters that appear in function calls.
Formal parameters: The parameters that appear in function declarations.

For example:

```
#include <stdio.h>
int sum(int a, int b)
{
    int c=a+b;
    return c;
}

int main(
{
   int var1 =10;
   int var2 = 20;
   int var3 = sum(var1, var2);
   printf("%d", var3);

   return 0;
}
```

In the above example variable a and b are the formal parameters (or formal arguments). Variable var1 and var2 are the actual arguments (or actual parameters). The actual parameters can also be the values. Like sum(10, 20), here 10 and 20 are actual parameters.

In this guide, we will discuss **function call by value**. If you want to read call by reference method then refer this guide: function call by reference.

What is Function Call By value?

When we pass the actual parameters while calling a function then this is known as function call by value. In this case the values of actual parameters are copied to the formal parameters. Thus operations performed on the formal parameters don’t reflect in the actual parameters.

Example of Function call by Value

As mentioned above, in the call by value the actual arguments are copied to the formal arguments, hence any operation performed by function on arguments doesn’t affect actual parameters. Lets take an example to understand this:

```
#include <stdio.h>
int increment(int var)
{
   var = var+1;
   return var;
}

int main()
{
  int num1=20;
  int num2 = increment(num1);
  printf("num1 value is: %d", num1);
  printf("\nnum2 value is: %d", num2);
```

```
   return 0;
}
```

Output:

```
num1 value is: 20
num2 value is: 21
```

Explanation
We passed the variable num1 while calling the method, but since we are calling the function using call by value method, only the value of num1 is copied to the formal parameter var. Thus change made to the var doesn't reflect in the num1.

Example 2: Swapping numbers using Function Call by Value

```
#include <stdio.h>
void swapnum( int var1, int var2 )
{
   int tempnum ;
   /*Copying var1 value into temporary variable */
   tempnum = var1 ;

   /* Copying var2 value into var1*/
   var1 = var2 ;

   /*Copying temporary variable value into var2 */
   var2 = tempnum ;

}
int main( )
{
    int num1 = 35, num2 = 45 ;
    printf("Before swapping: %d, %d", num1, num2);

    /*calling swap function*/
    swapnum(num1, num2);
    printf("\nAfter swapping: %d, %d", num1, num2);
}
```

Output:

```
Before swapping: 35, 45
After swapping: 35, 45
```

Why variables remain unchanged even after the swap?
The reason is same – function is called by value for num1 & num2. So actually var1 and var2 gets swapped (not num1 & num2). As in call by value actual parameters are just copied into the formal parameters.

Function call by reference in C Programming

Actual parameters: The parameters that appear in function calls.
Formal parameters: The parameters that appear in function declarations.
For example: We have a function declaration like this:

```
int sum(int a, int b);
```

The a and b parameters are formal parameters.

We are calling the function like this:

```
int s = sum(10, 20); //Here 10 and 20 are actual parameters
or
int s = sum(n1, n2); //Here n1 and n2 are actual parameters
```

In this guide, we will discuss function call by reference method. If you want to read call by value method then refer this guide: function call by value.

Lets get back to the point.

What is Function Call By Reference?
When we call a function by passing the addresses of actual parameters then this way of calling the function is known as call by reference. In call by reference, the operation performed on formal parameters, affects the value of actual parameters because all the operations performed on the value stored in the address of actual parameters. It may sound confusing first but the following example would clear your doubts.

Example of Function call by Reference

Lets take a simple example. Read the comments in the following program.

```
#include <stdio.h>
void increment(int  *var)
{
   /* Although we are performing the increment on variable
    * var, however the var is a pointer that holds the address
    * of variable num, which means the increment is actually done
    * on the address where value of num is stored.
    */
   *var = *var+1;
}
int main()
{
    int num=20;
    /* This way of calling the function is known as call by
     * reference. Instead of passing the variable num, we are
     * passing the address of variable num
     */
    increment(&num);
    printf("Value of num is: %d", num);
  return 0;
```

```
}
```

Output:

```
Value of num is: 21
```

Example 2: Function Call by Reference – Swapping numbers

Here we are swapping the numbers using call by reference. As you can see the values of the variables have been changed after calling the swapnum() function because the swap happened on the addresses of the variables num1 and num2.

```
#include
void swapnum ( int *var1, int *var2 )
{
  int tempnum ;
  tempnum = *var1 ;
  *var1 = *var2 ;
  *var2 = tempnum ;
}
int main( )
{
  int num1 = 35, num2 = 45 ;
  printf("Before swapping:");
  printf("\nnum1 value is %d", num1);
  printf("\nnum2 value is %d", num2);

  /*calling swap function*/
  swapnum( &num1, &num2 );

  printf("\nAfter swapping:");
  printf("\nnum1 value is %d", num1);
  printf("\nnum2 value is %d", num2);
  return 0;
}
```

Output:

```
Before swapping:
num1 value is 35
num2 value is 45
After swapping:
num1 value is 45
num2 value is 35
```

Passing array to function in C programming with example

To understand this guide, you should have the knowledge of following C Programming topics:

1. C – Array
2. Function call by value in C

3. Function call by reference in C

Passing array to function using call by value method

As we already know in this type of function call, the actual parameter is copied to the formal parameters.

```
#include <stdio.h>
void disp( char ch)
{
  printf("%c ", ch);
}
int main()
{
  char arr[] = {'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j'};
  for (int x=0; x<10; x++)
  {
     /* I'm passing each element one by one using subscript*/
     disp (arr[x]);
  }

  return 0;
}
```

Output:

a b c d e f g h i j

Passing array to function using call by reference

When we pass the address of an array while calling a function then this is called function call by reference. When we pass an address as an argument, the function declaration should have a pointer as a parameter to receive the passed address.

```
#include <stdio.h>
void disp( int *num)
{
   printf("%d ", *num);
}

int main()
{
    int arr[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 0};
    for (int i=0; i<10; i++)
    {
       /* Passing addresses of array elements*/
       disp (&arr[i]);
    }

    return 0;
}
```

Output:

1 2 3 4 5 6 7 8 9 0

How to pass an entire array to a function as an argument?

In the above example, we have passed the address of each array element one by one using a However you can also pass an entire array to a function like this:

> Note: The array name itself is the address of first element of that array. For example if array name is arr then you can say that **arr** is equivalent to the **&arr[0]**.

```
#include <stdio.h>
void myfuncn( int *var1, int var2)
{
      /* The pointer var1 is pointing to the first element of
       * the array and the var2 is the size of the array. In the
       * loop we are incrementing pointer so that it points to
       * the next element of the array on each increment.
       *
       */
   for(int x=0; x<var2; x++)
   {
      printf("Value of var_arr[%d] is: %d \n", x, *var1);
      /*increment pointer for next element fetch*/
      var1++;
   }
}

int main()
{
    int var_arr[] = {11, 22, 33, 44, 55, 66, 77};
    myfuncn(var_arr, 7);
    return 0;
}
```

Output:

```
Value of var_arr[0] is: 11
Value of var_arr[1] is: 22
Value of var_arr[2] is: 33
Value of var_arr[3] is: 44
Value of var_arr[4] is: 55
Value of var_arr[5] is: 66
Value of var_arr[6] is: 77
```

-