Andhra Pradesh State Skill Development Corporation

Skill AP
APSSDC

Programming in C

Functions

Functions

Function:

A function is a group of statements that together perform a task. Every C program has at least one function, which is main(), and all the most trivial programs can define additional functions.

You can divide up your code into separate functions. How you divide up your code among different functions is up to you, but logically the division is such that each function performs a specific task.

A function declaration tells the compiler about a function's name, return type, and parameters. A function definition provides the actual body of the function.

The C standard library provides numerous built-in functions that your program can call. For example, strcat() to concatenate two strings, memcpy() to copy one memory location to another location, and many more functions.

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, the 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 sets of statements are replaced by function calls.

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 the 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.

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 a user is known as a 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
}
```

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

A function definition in C programming consists of a function header and a function body. Here are all the parts of a function −

- **Return Type** − A function may return a value. The return_type is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the return_type is the keyword void.

- **Function Name** − This is the actual name of the function. The function name and the parameter list together constitute the function signature.

- **Parameters** − A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.

- **Function Body** − The function body contains a collection of statements that define what the function does.

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?
```

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

Function with arguments and no return valueWe 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;
    }
}
```

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
```

Function Arguments

If a function is to use arguments, it must declare variables that accept the values of the arguments. These variables are called the formal parameters of the function.

Formal parameters behave like other local variables inside the function and are created upon entry into the function and destroyed upon exit.

While calling a function, there are two ways in which arguments can be passed to a function −

S.No.	Call Type & Description
1	**Call by value** This method copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.

2	**Call by reference** This method copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument.

By default, C uses call by value to pass arguments. In general, it means the code within a function cannot alter the arguments used to call the function.

A scope in any programming is a region of the program where a defined variable can have its existence and beyond that variable it cannot be accessed. There are three places where variables can be declared in C programming language −

- Inside a function or a block which is called **local variables**.
- Outside of all functions which are called **global variables.**
- In the definition of function parameters which are called **formal parameters.**

Let us understand what are local and global variables, and formal parameters.

Local Variables

Variables that are declared inside a function or block are called local variables. They can be used only by statements that are inside that function or block of code. Local variables are not known to functions outside their own. The following example shows how local variables are used. Here all the variables a, b, and c are local to main() function.

Example

```
#include <stdio.h>

int main () {

  /* local variable declaration */
  int a, b;
  int c;

  /* actual initialization */
  a = 10;
  b = 20;
  c = a + b;

  printf ("value of a = %d, b = %d and c = %d\n", a, b, c);

  return 0;
}
```

Global Variables

Global variables are defined outside a function, usually on top of the program. Global variables hold their values throughout the lifetime of your program and they can be accessed inside any of the functions defined for the program.

A global variable can be accessed by any function. That is, a global variable is available for use throughout your entire program after its declaration. The following program show how global variables are used in a program.

Example

```
#include <stdio.h>

/* global variable declaration */
int g;

int main () {

  /* local variable declaration */
  int a, b;

  /* actual initialization */
  a = 10;
  b = 20;
  g = a + b;

  printf ("value of a = %d, b = %d and g = %d\n", a, b, g);

  return 0;
}
```

A program can have same name for local and global variables but the value of local variable inside a function will take preference. Here is an example −

```
#include <stdio.h>

/* global variable declaration */
int g = 20;

int main () {

  /* local variable declaration */
  int g = 10;

  printf ("value of g = %d\n",  g);

  return 0;
```

When the above code is compiled and executed, it produces the following result −

```
value of g = 10
```

Formal Parameters

Formal parameters are treated as local variables with-in a function and they take precedence over global variables. Following is an example −

```
#include <stdio.h>

/* global variable declaration */
int a = 20;

int main () {

  /* local variable declaration in main function */
  int a = 10;
  int b = 20;
  int c = 0;

  printf ("value of a in main() = %d\n",  a);
  c = sum( a, b);
  printf ("value of c in main() = %d\n",  c);

  return 0;
}

/* function to add two integers */
int sum(int a, int b) {

   printf ("value of a in sum() = %d\n",  a);
   printf ("value of b in sum() = %d\n",  b);

   return a + b;
}
```

When the above code is compiled and executed, it produces the following result −

```
value of a in main() = 10
value of a in sum() = 10
value of b in sum() = 20
value of c in main() = 30
```

Initializing Local and Global Variables

When a local variable is defined, it is not initialized by the system, you must initialize it yourself. Global variables are initialized automatically by the system when you define them as follows −

Data Type	Initial Default Value
int	0
char	'\0'
float	0
double	0
pointer	NULL

It is a good programming practice to initialize variables properly, otherwise your program may produce unexpected results, because uninitialized variables will take some garbage value already available at their memory location.

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
```

Function call by reference in C Programming

Before we discuss function call by reference, 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: 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
```

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 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
```