

Functions in C

A function is a block of statements that performs a specific task. Suppose you are building an application in C language and in one of your program, you need to perform a same task 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 codes in C.

Types of functions

1) Predefined standard library functions –

Functions such as puts(), gets(), printf(), scanf() etc – These are the functions which already have a definition in header files (.h files like stdio.h), so we just call them whenever there is a need to use them.

2) User Defined functions – The functions that we create in a program are known as user defined functions.

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.

Function prototype

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.

Syntax of function prototype

```
returnType functionName(type1 argument1, type2 argument2, ...);
```

In the above example, `int addNumbers(int a, int b);` is the function prototype which provides the following information to the compiler:

- 1. name of the function is addNumbers()**

2. return type of the function is int

3. two arguments of type int are passed to the function

The function prototype is not needed if the user-defined function is defined before the main() function.

Calling a function

Control of the program is transferred to the user-defined function by calling it.

Syntax of function call

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

In the above example, the function call is made using addNumbers(n1, n2); statement inside the main() function.

Function definition

Function definition contains the block of code to perform a specific task. In our example, adding two numbers and returning it.

Syntax of function definition

```
returnType functionName(type1 argument1, type2 argument2, ...)
```

```
{  
  
    //body of the function  
  
}
```

When a function is called, the control of the program is transferred to the function definition. And, the compiler starts executing the codes inside the body of a function.

Passing arguments to a function

In programming, argument refers to the variable passed to the function. In the above example, two variables n1 and n2 are passed during the function call.

The parameters a and b accepts the passed arguments in the function definition. These arguments are called formal parameters of the function.

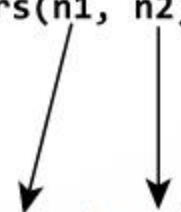
How to pass arguments to a function?

```
#include <stdio.h>

int addNumbers(int a, int b);

int main()
{
    ... ..
    sum = addNumbers(n1, n2);
    ... ..
}

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

A diagram with two arrows. One arrow starts at the parameter 'n1' in the function call 'addNumbers(n1, n2)' inside the 'main' function and points down to the parameter 'a' in the function definition 'int addNumbers(int a, int b)'. The other arrow starts at the parameter 'n2' in the function call and points down to the parameter 'b' in the function definition.

The type of arguments passed to a function and the formal parameters must match, otherwise, the compiler will throw an error.

Example 1: 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("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:

How are you?

Note: for example, if function return type is char, then function should return a value of char type and while calling this function the main() function should have a variable of char data type to store the returned value.

Structure would look like –

```
char abc(char ch1, char ch2)  
{  
    char ch3;  
    ...  
    ...  
    return ch3;  
}
```



```
int main()
{
    ...
    char c1 = abc('a', 'x');
    ...
}
```

Functions call 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.

CALL BY VALUE vs CALL BY REFERENCE

While calling a function, we pass values of variables to it. Such functions are known as “Call By Values”.

In this method, the value of each variable in calling function is copied into corresponding dummy variables of the called function.

With this method, the changes made to the dummy variables in the called function have no effect on the values of actual variables in the calling function.

While calling a function, instead of passing the values of variables, we pass address of variables(location of variables) to the function known as “Call By References.

In this method, the address of actual variables in the calling function are copied into the dummy variables of the called function.

With this method, using addresses we would have an access to the actual variables and hence we would be able to manipulate them.

```
/* C program to illustrate call by value */
```

```
#include <stdio.h>
```

```
void swapx(int x, int y);
```

```
int main()
```

```
{
```

```
    int a = 10, b = 20;
```

```
    /* Pass by Values */
```

```
    swapx(a, b);
```

```
    printf("a=%d b=%d\n", a, b);

    return 0;

}

void swapx(int x, int y)

{

    int t;

    t = x;

    x = y;

    y = t;

    printf("x=%d y=%d\n", x, y);

}
```

Output:

x=20 y=10

a=10 b=20

C program to illustrate Call by Reference

```
#include <stdio.h>
```

```
void swapx(int*, int*);
```

```
int main()

{

    int a = 10, b = 20;

    /*Pass reference */

    swapx(&a, &b);

    printf("a=%d b=%d\n", a, b);

    return 0;

}

void swapx(int* x, int* y)

{

    int t;

    t = *x;

    *x = *y;

    *y = t;

    printf("x=%d y=%d\n", *x, *y);

}
```

Output:

$$x=20 \ y=10$$

$$a=20 \ b=10$$