**Chapter-7**

**Functions**

**Introduction**

A function is a self contained sub program that is meant to be some specific, well defined task. A function is a group of statements that together perform a task. A C-program consists of one or more functions. If a program has only one function then it must be the main( ) function. 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. Functions play a very useful role in modular programming, functions are also known as modules. In Modular Programming, a program can be divided into modules, these modules are known as functions. 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 program execution 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.

**Advantages of using function**

1. Functions increases code reusability by avoiding rewriting of same code over and over.
2. If a program is divided into multiple functions, then each function can be independently developed. So program development will be easier.
3. Program development will be faster.
4. Program debugging will be easier.
5. Function reduces program complexity.
6. Easier to understand logic involved in the program.
7. Recursive call is possible through function.

**C programs have two types of functions-**

i. Library Functions

ii. User defined Functions

1. **Library function**

Library functions are supplied with every C compiler. The source code of the library functions is not given to the user. These functions are precompiled and the user gets only the object code. This object code is linked to the object code of your program by the linker. Different categories of library functions are grouped together in separate library files. When we call a library function in our program, the linker selects the code of that function from the library file and adds it to the program.

To use library function in our program we should know-

i. Name of the function and its purpose

ii. Type and number of arguments it accepts

iii. Type of the value it returns

iv. Name of the header file to be included

We can define any function of our own with the same name as that of any function in the C library. If we do so then the function that we have defined will take precedence over the library function with the same name.

Some examples are

* printf(), scanf() are defined in header file stdio.h
* getch(), clrscr() are defined in header file conio.h
* strlen(), strupr() are defined in header file string.h
* pow(), sqrt() are defined in header file math.h

1. **User defined function**

Users can create their own functions for performing any specific task of the program. These types of functions are called user defined functions. To create and use these functions, we should know about these three things-

1. Function definition

2. Function declaration

3. Function call

Lets take an example of function to add two numbers.

#include<stdio.h>

#include<conio.h>

void sum(int,int); //**function declaration**

void main()

{

int a,b;

clrscr();

printf("Enter two numbers\n");

scanf("%d%d",&a,&b);

sum(a,b); **//function call**

getch();

}

void sum(int x,int y)

{ **//function definition**

int s;

s=x+y;

printf("Sum =%d",s);

}

### Function Definition

The function definition consists of the whole description and code of a function. It tells what the function is doing and that are its inputs and outputs. A function definition consists of two parts - a function header and a function body. The general syntax of a function definition is-

return\_type func\_name(type1 arg1, type2 arg2, ………….)

{

local variables declarations;

statements;

…………..

return(expression);

}

The first line in the function definition is known as the function header and after this the body of the function is written enclosed in curly braces.

The ***return\_type*** denotes the type of the value that will returned by the function. The return\_type is optional and if omitted, it is assumed to be int by default. A function can return either one value or no value. If a function does not return any value then void should be written in place of return\_type.

***func\_name*** specifies the name of the function and it can be any valid C identifier. After function name, the argument declarations are given in parentheses, which mention the type and name of the arguments. These are known as formal arguments and used to accept values. A function can take any number of left empty or void can be written inside the parentheses. 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.

The body of function is a compound statement (or block), which consists of declarations variables, and C statements followed by an optional return statement. The variables declared inside the function are known as local variables, since they are local to that function only, i.e. they have existence only in the function in which they are declared, and they cannot be used anywhere else in the program. There can be any number of valid C statements inside a function body. The ***return*** statement is optional. It may be absent if the function does not return any value.

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 Declaration/Function prototype:

A function declaration tells the compiler about a function's name, return type, and parameters. The calling function needs information about the called function. If definition of the called function is placed before the calling function, then declaration is not needed. The function declaration is also known as the function prototype, and it informs the compiler about the following three things-

1. Name of the function

2. Number and type of arguments received by the function

3. Type of value returned by the function

Function declaration tells the compiler that a function with these features will be defined and used later in the program. The general syntax of a function declaration is-

return\_type func\_name(type1, type2,……..);

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. Any C function by default returns an int value. More specifically, whenever a call is made to a function, the compiler assumes that this function would return a value of the type int. If we desire that a function should return a value other than an int, then it is necessary to explicitly mention so in the calling function as well as in the called function. Suppose we want to find out square of a number using a function. This is how this simple program would look like:

main( )

{

float a, b ;

printf ( "\nEnter any number " ) ;

scanf ( "%f", &a ) ;

b = square ( a ) ;

printf ( "\nSquare of %f is %f", a, b ) ;

}

square ( float x )

{

float y ;

y = x \* x ;

return ( y ) ;

}

And here are three sample runs of this program...

Enter any number 3

Square of 3 is 9.000000

Enter any number 1.5

Square of 1.5 is 2.000000

Enter any number 2.5

Square of 2.5 is 6.000000

The first of these answers is correct. But square of 1.5 is definitely not 2. Neither is 6 a square of 2.5. This happened because any C function, by default, always returns an integer value. Therefore, even though the function square( ) calculates the square of 1.5 as 2.25, the problem crops up when this 2.25 is to be returned to main( ). square( ) is not capable of returning a float value. How do we overcome this? The following program segment illustrates how to make square( ) capable of returning a float value.

main( )

{

float square ( float ) ;

float a, b ;

printf ( "\nEnter any number " ) ;

scanf ( "%f", &a ) ;

b = square ( a ) ;

printf ( "\nSquare of %f is %f", a, b ) ;

}

float square ( float x )

{

float y ;

y = x \* x ;

return ( y ) ;

}

And here is the output...

Enter any number 1.5

Square of 1.5 is 2.250000

Enter any number 2.5

Square of 2.5 is 6.250000

Now the expected answers i.e. 2.25 and 6.25 are obtained. Note that the function square( ) must be declared in main( ) as

float square ( float ) ;

This statement is often called the prototype declaration of the square( ) function. What it means is square( ) is a function that receives a float and returns a float. We have done the prototype declaration in main( ) because we have called it from main( ). There is a possibility that we may call square( ) from several other functions other than main( ). Does this mean that we would need prototype declaration of square( ) in all these functions. No, in such a case we would make only one declaration outside all the functions at the beginning of the program.

1. **Function call**

A function is called by simply writing its name followed by the argument list inside the parentheses.

func\_name(arg1, arg2, arg3,….);

These arguments arg1,arg2,arg3,…are called actual arguments. Here ***func\_name*** is known as the called function while the function in which this function call is placed is known as the calling function. When a function is called, the control passes to the called function, which is executed and after this control is transferred to the statement following the function call in the calling function. The code of a function is executed only when it is called by some other function. If the function is defined and not called even once then it’s code will never be executed. A function can be called more than once, so the code is executed each time it is called. The execution of a function finishes either when the closing braces of the function body are reached or if return statement is encountered.

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

**Return statement**

The return statement is used in a function to return a value to the calling function. It may also be used for immediate exit from the called function to the calling function without returning a value. This statement can appear anywhere inside the body of the function. There are two ways in which it can be used-

return;

return (expression);

Here *return* is a keyword. The first form of return statement is used to terminate the function without returning any value. The second form of return statement is used to terminate a function and return a value to the calling function.

**Function arguments**

The calling function sends some values to the called function for communication; these values are called arguments or parameters. Aparameter is a variable in a method definition. When a method is called, the arguments are the data you pass into the method's parameters. Parameter is variable in the declaration of function. Argument is the actual value of this variable that gets passed to function.

* **Actual argument**

The arguments which are mentioned in the function call are known as actual arguments, since these are the values which are actually sent to the called function. Actual arguments can be written in the form of variables, constants or expressions or any function call that returns a value. For example

func1(12, 23);

here 12 and 23 are actual arguments.

Actual arguments can be constant, variables, expressions etc.

func1(a, b); // here actual arguments are variable

func1(a + b, b + a); // here actual arguments are expression

* **Formal argument**

The name of the arguments, which are mentioned in the function definition are called formal or dummy arguments since they are used just to hold the values that are sent by the calling function. These formal arguments are simply like other local variables of the function which are created when the function call starts and are destroyed when the function ends.

int factorial(int n)

{

// write logic here

}

Here n is the formal argument.

Things to remember about actual and formal arguments.

1. Order, number, and type of the actual arguments in the function call must match with formal arguments of the function.
2. If there is type mismatch between actual and formal arguments then the compiler will try to convert the type of actual arguments to formal arguments if it is legal, Otherwise, a garbage value will be passed to the formal argument.
3. Changes made in the formal argument do not affect the actual arguments.

The following program demonstrates this behavior.

#include<stdio.h>

void func\_1(int);

int main()

{

int x = 10;

printf("Before function call\n");

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

func\_1(x);

printf("After function call\n");

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

// signal to operating system program ran fine

return 0;

}

void func\_1(int a)

{

a += 1;

a++;

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

}

Here the value of variable x is 10 before the function func\_1() is called, after func\_1() is called, the value of x inside main() is still 10. The changes made inside the function func\_1() doesn't affect the value of x. This happens because when we pass values to the functions, a copy of the value is made and that copy is passed to the formal arguments. Hence Formal arguments work on a copy of the original value, not the original value itself, that's why changes made inside func\_1() is not reflected inside main(). This process is known as passing arguments using **Call by Value**,

*The order, number and type of actual arguments in the function call should match with the order, number and type of formal arguments in the function definition.*

### Types of user defined Functions

The user defined functions can be classified into four categories on the basis of the arguments and return value;

1. Function with no arguments and no return value.

2. Function with no argument and return value.

3. Function with argument and no return value.

4. Function with argument and return value.

1. **Function with no arguments and no return value**

When a function has no arguments, it does not receive any data from the calling function. Similarly when it does not return a value, the calling function does not receive any data from the called function.  
Syntax :

Function declaration : void function();

Function call : function();

Function definition :

void function()

{

statements;

}

Example 1.

Program to add two numbers.

#include<stdio.h>

#include<conio.h>

void sum();

void main()

{

clrscr();

sum();

getch();

}

void sum()

{

int x,y,s;

printf("Enter two numbers\n");

scanf("%d%d",&x,&y);

s=x+y;

printf("Sum =%d",s);

}

## Example 2

#include <stdio.h>

void checkPrimeNumber();

int main()

{

checkPrimeNumber(); // no argument is passed to prime()

return 0;

}

// return type of the function is void becuase no value is returned from the function

void checkPrimeNumber()

{

int n, i, flag=0;

printf("Enter a positive integer: ");

scanf("%d",&n);

for(i=2; i <= n/2; ++i)

{

if(n%i == 0)

{

flag = 1;

}

}

if (flag == 1)

printf("%d is not a prime number.", n);

else

printf("%d is a prime number.", n);

}

The checkPrimeNumber(); function takes input from the user, checks whether it is a prime number or not and displays it on the screen.

The empty parentheses in checkPrimeNumber(); statement inside the main() function indicates that no argument is passed to the function.

The return type of the function is void. Hence, no value is returned from the function.

1. **Function with no argument and a return value**

There could be occasions where we may need to design functions that may not take any arguments but returns a value to the calling function. A example for this is getchar function it has no parameters but it returns an integer an integer type data that represents a character.  
Syntax :

Function declaration : int function();

Function call : function();

Function definition :

int function()

{

statements;

return x;

}

Example 1

Program to add two numbers.

#include<stdio.h>

#include<conio.h>

int sum();

void main()

{

int c;

clrscr();

c=sum();

printf("sum=%d",c);

getch();

}

int sum()

{

int x,y,s;

printf("Enter two numbers\n");

scanf("%d%d",&x,&y);

s=x+y;

return(s);

}

## Example 2.

#include <stdio.h>

int getInteger();

int main()

{

int n, i, flag = 0;

// no argument is passed to the function

// the value returned from the function is assigned to n

n = getInteger();

for(i=2; i<=n/2; ++i)

{

if(n%i==0){

flag = 1;

break;

}

}

if (flag == 1)

printf("%d is not a prime number.", n);

else

printf("%d is a prime number.", n);

return 0;

}

// getInteger() function returns integer entered by the user

int getInteger()

{

int n;

printf("Enter a positive integer: ");

scanf("%d",&n);

return n;

}

The empty parentheses in n = getInteger(); statement indicates that no argument is passed to the function. And, the value returned from the function is assigned to n.

Here, the getInteger() function takes input from the user and returns it. The code to check whether a number is prime or not is inside the main() function.

1. **Function with argument and no return value**

 When a function has arguments, it receive any data from the calling function but it returns no values.

Syntax :

Function declaration : void function ( int );

Function call : function( x );

Function definition**:**

void function( int x )

{

statements;

}

Example

Program to add two numbers

#include<stdio.h>

#include<conio.h>

void sum(int,int);

void main()

{

int a,b;

clrscr();

printf("Enter two numbers\n");

scanf("%d%d",&a,&b);

sum(a,b);

getch();

}

void sum(int x,int y)

{

int s;

s=x+y;

printf("Sum =%d",s);

}

## Example 2

#include <stdio.h>

void checkPrimeAndDisplay(int n);

int main()

{

int n;

printf("Enter a positive integer: ");

scanf("%d",&n);

// n is passed to the function

checkPrimeAndDisplay(n);

return 0;

}

// void indicates that no value is returned from the function

void checkPrimeAndDisplay(int n)

{

int i, flag = 0;

for(i=2; i <= n/2; ++i)

{

if(n%i == 0){

flag = 1;

break;

}

}

if(flag == 1)

printf("%d is not a prime number.",n);

else

printf("%d is a prime number.", n);

}

The integer value entered by the user is passed to checkPrimeAndDisplay() function.

Here, the checkPrimeAndDisplay() function checks whether the argument passed is a prime number or not and displays the appropriate message.

1. **Function with argument and return value**

Syntax :

Function declaration : int function ( int );

Function call : function( x );

Function definition**:**

int function( int x )

{

statements;

return x;

}

Example 1

Program to add two numbers.

#include<stdio.h>

#include<conio.h>

int sum(int,int);

void main()

{

int a,b,c;

clrscr();

printf("Enter two numbers\n");

scanf("%d%d",&a,&b);

c=sum(a,b);

printf("sum=%d",c);

getch();

}

int sum(int x,int y)

{

int s;

s=x+y;

return(s);

}

## Example 2

#include <stdio.h>

int checkPrimeNumber(int n);

int main()

{

int n, flag;

printf("Enter a positive integer: ");

scanf("%d",&n);

// n is passed to the checkPrimeNumber() function

// the value returned from the function is assigned to flag variable

flag = checkPrimeNumber(n);

if(flag==1)

printf("%d is not a prime number",n);

else

printf("%d is a prime number",n);

return 0;

}

// integer is returned from the function

int checkPrimeNumber(int n)

{

/\* Integer value is returned from function checkPrimeNumber() \*/

int i;

for(i=2; i <= n/2; ++i)

{

if(n%i == 0)

return 1;

}

return 0;

}

The input from the user is passed to checkPrimeNumber() function. The checkPrimeNumber() function checks whether the passed argument is prime or not. If the passed argument is a prime number, the function returns 0. If the passed argument is a non-prime number, the function returns 1. The return value is assigned to flag variable. Then, the appropriate message is displayed from the main() function.

## Which approach is better?

Well, it depends on the problem you are trying to solve. In case of this problem, the last approach is better.

A function should perform a specific task. The checkPrimeNumber() function doesn't take input from the user nor it displays the appropriate message. It only checks whether a number is prime or not, which makes code modular, easy to understand and debug.

**Difference between library function and user defined function**

|  |  |
| --- | --- |
| **Library function** | **User defined function** |
| 1. Library function is a predefined function in a header file or preprocessor directive | 1. User defined function is not a predefined function, it is defined by the programmer according to the need. |
| 1. Programmer can simply use this function by including respective header file. | 1. Programmer has to declare, define and use this function by themself. |
| 1. The program using library function will be usually short as the programmer doesn’t have to define the function. | 1. The program using user defined function will be usually lengthy as the programmer has to define the function. |
| 1. Program development time will be faster. | 1. Program development time will be usually slower. |
| 1. Program will be simple. | 1. Program will be complex. |
| 1. This function requires header file to use it. | 1. This function requires function prototype to use it. |

### main ( )

Execution of every C program always begin with the function main( ). Each function is called directly or indirectly in main( ) and after all functions have done their operations, control returns back to main( ). There can be only one main( ) function in a program.

The main( ) function is a user defined function but the name, number and type of arguments are predefined in the language. The operating system calls the main function and main( ) returns a value of integer type to the operating system. If the value returned in zero, it implies that the function has terminated successively and any non zero returns value indicates an error. If no return value is specified in main( ) then any garbage value will be returned automatically. Calling the function exit( ) with an integer value is equivalent to returning that value from main( ). The function main( ) can also take arguments, which will be discussed in further chapters.

The definition, declaration and call of main( ) function-

function declaration - by the C compiler

function definition - by the programmer

function call - by the operating system

**Recursion**

Recursion is a powerful technique of writing a complicated algorithm in an easy way. According to this technique a problem is defined in terms of itself. The problem is solved by dividing it into smaller problems, which are similar in nature to the original problem. These smaller problems are solved and their solutions are applied to get the final solution of our original problem. In C, it is possible for the functions to call themselves. A function is called ‘recursive’ if a statement within the body of a function calls the same function. Sometimes called ‘circular definition’, recursion is thus the process of defining something in terms of itself. The recursion continues until some condition is met to prevent it. To prevent infinite recursion, [if...else statement](https://www.programiz.com/c-programming/c-if-else-statement) (or similar approach) can be used where one branch makes the recursive call and other doesn't. A function will be recursive, if it contain following features:

1. Function should call itself.
2. Function should have a stopping condition (base criteria) and every time the function calls itself it must be closer to base criteria.

Example.

1. Write a program to calculate factorial of any given number using recursive function.

#include<stdio.h>

#include<conio.h>

long int factorial(int);

void main()

{

int num,fact;

clrscr();

printf("Enter a number\n");

scanf("%d",&num);

fact=factorial(num);

printf("Factorial of %d = %d",num,fact);

getch();

}

long int factorial(int n)

{

if(n==0)

return 1;

else

return (n\*factorial(n-1));

}

1. Write a program to calculate the sum of N natural numbers using recursive function.

#include<stdio.h>

#include<conio.h>

int sum(int);

void main()

{

int n,s;

clrscr();

printf("Input a number\n");

scanf("%d",&n);

s=sum(n);

printf("Sum of natural numbers=%d",s);

getch();

}

int sum(int n)

{

if(n<=0)

return 0;

else

return(n+sum(n-1));

}

1. Write a recursive program to calculate xy (x to the power y).

#include<stdio.h>

#include<conio.h>

int power(int,int);

void main()

{

int a,p,n;

clrscr();

printf("Enter value of a and n :");

scanf("%d%d",&a,&n);

p=power(a,n);

printf("%d raised to power %d is %d",a,n,p);

getch();

}

int power(int a,int n)

{

if(n==0)

return 1;

else

return(a\*power(a,n-1));

}

### 4. Sum of Natural Numbers Using Recursion

#include <stdio.h>

int sum(int n);

int main()

{

int number, result;

printf("Enter a positive integer: ");

scanf("%d", &number);

result = sum(number);

printf("sum=%d", result);

}

int sum(int num)

{

if (num!=0)

return num + sum(num-1); // sum() function calls itself

else

return num;

}

**Output**

Enter a positive integer:

3

6

Initially, the sum() is called from the main() function with number passed as an argument. Suppose, the value of num is 3 initially. During next function call, 2 is passed to the sum()function. This process continues until num is equal to 0. When num is equal to 0, the if condition fails and the else part is executed returning the sum of integers to the main() function.

## Example 5. GCD of Two Numbers using Recursion

#include <stdio.h>

int hcf(int n1, int n2);

int main()

{

int n1, n2;

printf("Enter two positive integers: ");

scanf("%d %d", &n1, &n2);

printf("G.C.D of %d and %d is %d.", n1, n2, hcf(n1,n2));

return 0;

}

int hcf(int n1, int n2)

{

if (n2 != 0)

return hcf(n2, n1%n2);

else

return n1;

}

**Output**

Enter two positive integers: 366

60

G.C.D of 366 and 60 is 6.

Advantages of recursion:

1. A complex problem seems logically convincible when we solve it using recursion.
2. Recursion uses fairly lesser programming constructs to solve the problem than its iterative counterpart.

Disadvantages of recursion:

1. Fairly slower than its iterative solution.
2. For each step we make a recursive call to a function. For which it occupies significant amount of stack memory with each step.
3. May cause stack-overflow if the recursion goes too deep to solve the problem.
4. Difficult to debug and trace the values with each step of recursion.

**Nested functions in C**

Some programmer thinks that defining a function inside an another function is known as “nested function”. But the reality is that it is not a nested function, it is treated as lexical scoping. Lexical scoping is not valid in C because the compiler cant reach/find the correct memory location of the inner function.

Nested function **is not supported** by C because we cannot define a function within another function in C. We can declare a function inside a function, but it’s not a nested function.  
Because nested functions definitions can not [access local variables](https://www.geeksforgeeks.org/g-fact-24/) of the surrounding blocks, they can access only global variables of the containing module. This is done so that lookup of global variables doesn’t have to go through the directory. As in C, there are two nested scopes: local and global (and beyond this, built-ins). Therefore, nested functions have only a limited use. If we try to approach nested function in C, then we will get compile time error.

|  |
| --- |
| // C program to illustrate the  // concept of Nested function.  #include <stdio.h>  int main(void)  {      printf("Main");      int fun()      {          printf("fun");    // defining view() function inside fun() function.          int view()          {              printf("view");         }          return 1;      }      view();  } |

Run on IDE

Output:

Compile time error: undefined reference to `view'

**Visibility Scope and Lifetime of Variables**

There are two basic types of scope:  [local](https://docs.mql4.com/basis/variables/local) scope and [global](https://docs.mql4.com/basis/variables/global) scope.

A variable declared outside all functions is located into the global scope. Access to such variables can be done from anywhere in the program. These variables are located in the global pool of memory, so their lifetime coincides with the lifetime of the program.

A variable declared inside a block (part of code enclosed in curly brackets) belongs to the local scope. Such a variable is not visible (and therefore not available) outside the block, in which it is declared. The most common case of local declaration is a variable declared within a function. A variable declared locally, is located on the stack, and the lifetime of such a variable is equal to the lifetime of the function.

Since the scope of a local variable is the block in which it is declared, it is possible to declare variables with the same name, as those of variables declared in other blocks; as well as of those declared at upper levels, up to the global level.

**Local variables**

The variables that are defined within the body of a function or a block, are local variables. For example.

func()

{

int a,b;

…….

……

}

Here a and b are local variables which are defined within the body of the function ***func()***. Local variables can be used only in those functions or blocks, in which they are declared. The same variable name may be used in different functions. For example-

func1()

{

int a=2,b=4;

……….

……….

}

func2()

{

int a=15,b=20;

……….

……….

}

Here values of a=2 , b=4 are local to the function ***func1()*** and a=15, b=20 are local to the function ***func2().***

**Global variables**

The variables that are defined outside any function are called global variables. All functions in the program can access and modify global variables. It is useful to declare a variable global if it is to be used by many functions in the program. Global variables are automatically initialized to 0 at the time of declaration.

Example-

#include<stdio.h>

#include<conio.h>

int a,b=5; **//global variables**

void func1();

void func2();

void main()

{

clrscr();

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

func1();

func2();

getch();

}

void func1()

{

printf("Inside func1() a=%d and b=%d\n",a,b);

a=a+1;

b=b-1;

}

void func2()

{

printf("Inside func2() a=%d and b=%d",a,b);

}

**Passing by value (call by value)**

In pass by value, values of variables are passed to the function from the calling function. This method copies the value of actual parameters into formal parameters. In other words, when the function is called, a separate copy of the variables is created in the memory and the value of original variables is given to these variables. So if any changes are made in the value (of the called function) is not reflected to the original variable (of calling function). It can return only one value.

Example:-

#include<stdio.h>

#include<conio.h>

void change(int);

void main()

{

int a=15;

clrscr();

printf("Before calling function, a=%d\n",a);

change(a);

printf("After calling function, a=%d",a);

getch();

}

void change(int x)

{

x=x+5;

}

Output:

Before calling function, a=15

After calling function, a=15

**Passing by reference (call by reference)**

In passing by reference we pass the address or location of a variable to the function during function call. Pointers are used to call a function by reference. When a function is called by reference, then the formal argument becomes reference to the actual argument. This means that the called function doesn’t create its own copy of values rather, it refers to the original values only by reference name. Thus function works with original data and the changes are made in the original data itself. It can return more than one value at a time.

Example:-

#include<stdio.h>

#include<conio.h>

void change(int\*);

void main()

{

int a=15;

clrscr();

printf("Before calling function, a=%d\n",a);

change(&a);

printf("After calling function, a=%d",a);

getch();

}

void change(int \*x)

{

\*x=\*x+5;

}

Output:

Before calling function, a=15

After calling function, a=20

**Use of Array in function**

In C programming, a single array element or an entire [array](https://www.programiz.com/c-programming/c-arrays) can be passed to a [function](https://www.programiz.com/c-programming/c-functions). This can be done for both one-dimensional array or a multi-dimensional array.

**Passing One-dimensional Array in Function**

Single element of an array can be passed in similar manner as passing variable to a function.

**Example to pass a single element of an array to function**

#include <stdio.h>

void display(int age)

{

printf("%d", age);

}

int main()

{

int ageArray[] = { 2, 3, 4 };

display(ageArray[2]); //Passing array element ageArray[2] only.

return 0;

}

**Output**

4

**Passing an entire one-dimensional array to a function**

While passing arrays as arguments to the function, only the name of the array is passed (,i.e, starting address of memory area is passed as argument).

**Example to pass an array containing age of person to a function. This function should find average age and display the average age in main function.**

#include <stdio.h>

float average(float age[]);

int main()​

{

float avg, age[] = { 23.4, 55, 22.6, 3, 40.5, 18 };

avg = average(age); /\* Only name of array is passed as argument. \*/

printf("Average age=%.2f", avg);

return 0;

}

float average(float age[])

{

int i;

float avg, sum = 0.0;

for (i = 0; i < 6; ++i) {

sum += age[i];

}

avg = (sum / 6);

return avg;

}

**Output**

Average age=27.08

Example:-

#include<stdio.h>

#include<conio.h>

void func(int[]);

void main()

{

int arr[5]={5,8,12,45,9};

clrscr();

func(arr);

getch();

}

void func(int a[])

{

int i;

for(i=0;i<5;i++)

{

printf("%d\n",a[i]);

}

}

**Passing Multi-dimensional Arrays to Function**

To pass two-dimensional array to a function as an argument, starting address of memory area reserved is passed as in one dimensional array

#Example: Pass two-dimensional arrays to a function

#include <stdio.h>

void displayNumbers(int num[2][2]);

int main()

{

int num[2][2], i, j;

printf("Enter 4 numbers:\n");

for (i = 0; i < 2; ++i)

for (j = 0; j < 2; ++j)

scanf("%d", &num[i][j]);

// passing multi-dimensional array to displayNumbers function

displayNumbers(num);

return 0;

}

void displayNumbers(int num[2][2])

{

// Instead of the above line,

// void displayNumbers(int num[][2]) is also valid

int i, j;

printf("Displaying:\n");

for (i = 0; i < 2; ++i)

for (j = 0; j < 2; ++j)

printf("%d\n", num[i][j]);

}

**Output**

Enter 4 numbers:

2

3

4

5

Displaying:

2

3

4

5

Example:-

#include<stdio.h>

#include<conio.h>

int sum(int[][2],int[][2]);

void main()

{

int mat1[2][2]={3,7,9,1};

int mat2[2][2]={4,2,6,8};

clrscr();

sum(mat1,mat2);

getch();

}

int sum(int a[][2],int b[][2])

{

int i,j,c[2][2];

for(i=0;i<2;i++)

{

for(j=0;j<2;j++)

{

c[i][j]=a[i][j]+b[i][j];

}

}

for(i=0;i<2;i++)

{

for(j=0;j<2;j++)

{

printf("%d\t",c[i][j]);

}

printf("\n");

}

}

**Passing Strings to Functions**

Strings are just char arrays. So, they can be passed to a function in a similar manner as arrays.

Learn more about [passing array to a function](https://www.programiz.com/c-programming/c-arrays-functions).

#include <stdio.h>

void displayString(char str[]);

int main()

{

char str[50];

printf("Enter string: ");

gets(str);

displayString(str); // Passing string c to function.

return 0;

}

void displayString(char str[]){

printf("String Output: ");

puts(str);

}

Here, string c is passed from main() function to user-defined function displayString(). In function declaration, str[] is the formal argument.