

**Problem Solving Through  
programming in C  
Course Code: ONL1001**

**Functions: Introduction, storage  
class in C**

**Ms. SHUBHRA DWIVEDI  
School - SCOPE  
VIT-AP Amaravati**

# C Function

- A large C program is divided into basic building blocks called C function.
- C function contains set of instructions enclosed by “{ }” which performs specific operation in a C program.
- Actually, Collection of these functions creates a C program.

# Uses of C Function

- C functions are used to avoid rewriting same logic/code again and again in a program.
- There is no limit in calling C functions to make use of same functionality wherever required.
- We can call functions any number of times in a program and from any place in a program.
- A large C program can easily be tracked when it is divided into functions.
- The core concept of C functions are, **re-usability**, **dividing a big task into small pieces** to achieve the functionality and to **improve understandability** of very large C programs.

# 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.

# **FUNCTIONS**

```
graph TD; FUNCTIONS --> BuiltIn[Built-In Functions]; FUNCTIONS --> UserDefined[User-Defined Functions]; BuiltIn --- List["scanf()  
printf()  
getc()  
putc()"]; UserDefined --- Definition["A series of Instructions  
that are to be executed  
more than once"];
```

## **Built-In Functions**

**scanf()**

**printf()**

**getc()**

**putc()**

## **User-Defined Functions**

**A series of Instructions  
that are to be executed  
more than once**

# NEED FOR USER-DEFINED FUNCTIONS

- ▶ Every program must have a main function
- ▶ It is possible to code any program utilizing only main function, it leads to a number of problems
- ▶ The program may become too large and complex and as a result the task of debugging, testing, and maintaining becomes difficult
- ▶ If a program is divided into functional parts, then each part may be independently coded and later combined into a single unit
- ▶ These subprograms called 'functions' are much easier to understand, debug, and test

## NEED FOR USER-DEFINED FUNCTIONS

- ▶ There are times when some types of operation or calculation is repeated at many points throughout a program
- ▶ In such situations, we may repeat the program statements whenever they are needed
- ▶ Another approach is to design a function that can be called and used whenever required
- ▶ This saves both time and space

## **USER DEFINED FUNCTION :**

### **SYNTAX :**

**return\_type function\_name (argument list) {**

Set of statements – Block of code

**return;  
}**

### **call the function from main() :**

#### **syntax :**

**function\_name (argument list) ;**



**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.

**Block of code:** Set of C statements, which will be executed whenever a call will be made to the function.

# ELEMENTS OF USER-DEFINED FUNCTIONS

- ▶ **Function declaration or prototype** – informs compiler about return value's

the function name, function parameters and data type.

- ▶ **Function call** – This calls the actual function
- ▶ **Function definition** – This contains all the statements to be executed.

Sno	C Function aspects	Syntax
1	Function definition	return_type function_name(arguments list) { Body of function; }
2	function call	function_name ( arguments list );
3	function declaration	return_type function_name ( argument list);

## ELEMENTS OF USER-DEFINED FUNCTIONS

- ▶ Functions are classified as one of the derived data types in C
- ▶ Can define functions and use them like any other variables in C programs.
- ▶ Similarities between functions and variables in C
  - Both function name and variable names are considered identifiers and therefore they must adhere to the rules for identifiers.
  - Like variables, functions have types (such as int) associated with them
  - Like variables, function names and their types must be declared and defined before they are used in a program

For example – Suppose you want to create a function to add two integer variables.

Let's split the problem so that it would be easy to understand –

Function will add the two numbers so it should have some meaningful name like sum, addition, etc. For example let's take the name addition for this function.

**return\_type addition(argument list)**

This function addition adds two integer variables, which means I need two integer variable as input, let's provide two integer parameters in the function signature. The function signature would be –

**return\_type addition(int num1, int num2)**

The result of the sum of two integers would be integer only. Hence function should return an integer value – I got my return type – It would be integer –

**int addition(int num1, int num2);**

So you got your function prototype or signature. Now you can implement the logic in C program like this:

## 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?
```

# Return Statement

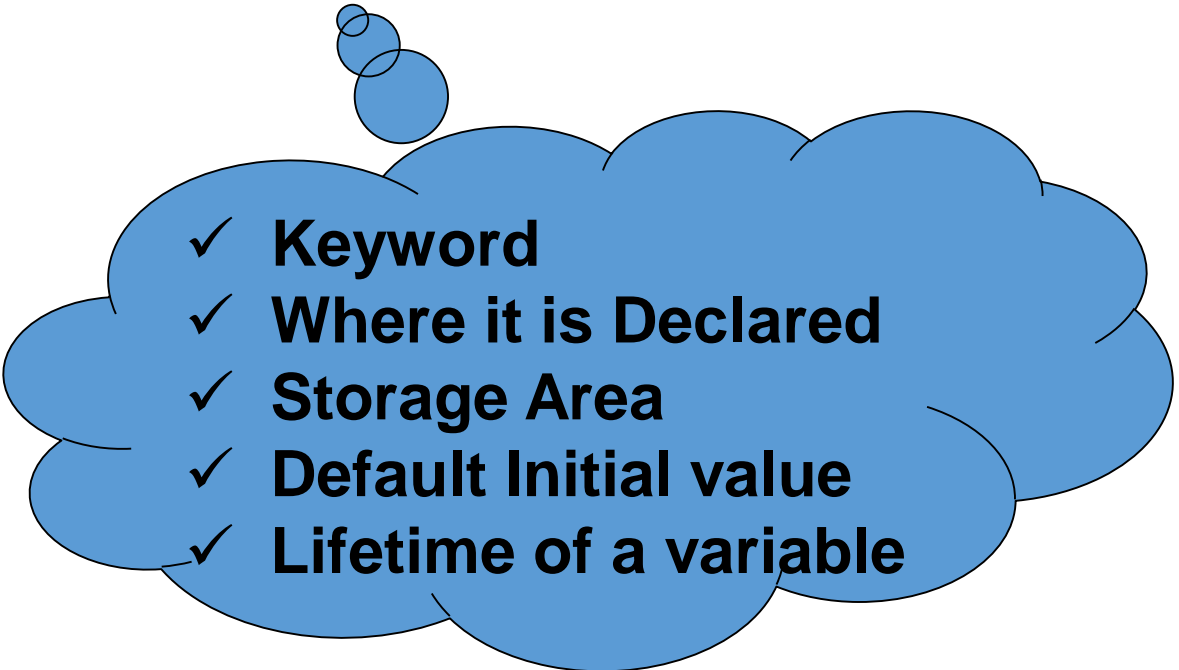
- ⊕ The return statement is used to return from a function.
- ⊕ It causes execution to return to the point at which the call to the function was made.
- ⊕ The return statement can have a value with it, which it returns to the program.

# Storage class in C

Every 'C' variable has a characteristic called its Storage Class.

All variables have datatype and storage classes

**Based On**

- 
- ✓ **Keyword**
  - ✓ **Where it is Declared**
  - ✓ **Storage Area**
  - ✓ **Default Initial value**
  - ✓ **Lifetime of a variable**



# Four different storage classes in "c"

- 1. Local or Auto or Internal variable**
- 2. External or Global variable**
- 3. Static variable**
- 4. Register variable**

Based on the **scope and lifetime** of a variable, variables categorized into three types

- Local variables
- Global variables
- Static variables
- Register variables

"**Scope**," tells about **visibility** (i.e., from where and all places variable is visible), whereas "**lifetime**," tells about durability (till how much time the value in the variable is valid).

## local variable

- The variables declared inside a function is known as a local variable.
- The scope of local variables is throughout the block, i.e., we can't access a local variable from outside the "**block**" in which we declared it.
- Life-time of a local variable is throughout the function, i.e., memory to the local variables allocated when the Execution of a "**function**" is started and will become invalid after finishing the Execution of a function.
- For example, All the variables declared inside the fun() are known as the local variables, and can't be accessed from functions other than fun(). They will be invalid once the Execution of fun() finished.

```
void function_1()
{
    int a, b; // you can use a and b within
braces only
}
```

```
void function_2()
{
    printf("%d\n", a); // ERROR, function_2()
doesn't know any variable a
}
```

```
void function_1()
{
    int a = 1, b = 2;
}
```

```
void function_2()
{
    int a = 10, b = 20;
}
```

**Note:** a and b are called local variables. They are available only inside the function in which they are defined (in this case function\_1()). If you try to use these variables outside the function in which they are defined, you will get an error. Another important point is that variables a and b only exists until function\_1() is executing. As soon as function function\_1() ends variables a and b are destroyed.

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

```
int main()
```

```
{
```

```
    int a = 100;
```

```
    {
```

```
        /*
```

```
        variable a declared in this block is  
        completely different from variable  
        declared outside.
```

```
        */
```

```
        int a = 10;
```

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

```
    }
```

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

```
    // signal to operating system everything works fine
```

```
    return 0;
```

```
}
```

Expected Output:

Inner a = 10

Outer a = 100

The variable a created inside the compound statement or block i.e inside braces ({} ) is completely different from variable a declared outside the block. As soon as the block ends the variable a declared inside the block is destroyed.

# Global variable:

- The variables declared outside the function is known as global variables.
- The scope of global variables is throughout the file, i.e. all the functions in a file can access it.
- The Life-time of a global variable is throughout the program, i.e. memory to the global variables will be allocated when the Execution of the program is started and will become invalid after finishing the Execution of the program.
- **For example**, the below global variable (a = 10) we are accessing the variable from inside the "main" function.

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

```
// global variable
```

```
int a = 10;
```

```
void main()
```

```
{
```

```
    printf("Priniting in main %d", a);
```

```
}
```

```
void fun()
```

```
{
```

```
    printf("printing in fun %d", a);
```

```
}
```

```
#include<stdio.h>
void func_1();
void func_2();
int a, b = 10; // declaring and initializing global variables
```

```
int main()
{
    printf("Global a = %d\n", a);
    printf("Global b = %d\n\n", b);

    func_1();
    func_2();

    // signal to operating system program ran fine
    return 0;
}
```

```
void func_1()
{
    printf("From func_1() Global a = %d\n", a);
    printf("From func_1() Global b = %d\n\n", b);
}
```

```
void func_2()
{
    int a = 5;
    printf("Inside func_2() a = %d\n", a);
}
```

Expected Output:

Global a = 0  
Global b = 10

From func\_1() Global a = 0  
From func\_1() Global b = 10

Inside func\_2() a = 5

**Note:** In line 4, a and b are declared as two global variables of type int. The variable a will be automatically initialized to 0. You can use variables a and b inside any function. Notice that inside function func\_2() there is a local variable with the same name as a global variable. When there is a conflict between the global variable and local variable, the local variable gets the precedence, that's why inside the func\_2() value of local variable a is printed.

Unlike local variables, global variables are not destroyed as soon as the function ends. They are available to any function until the program is executing.

# Static variables

A Static variable is able to retain its value between different function calls. The static variable is only initialized once, if it is not initialized, then it is automatically initialized to 0. Here is how to declare a static variable.

## Syntax: static type var\_name;

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

```
void func_1();
```

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

```
int main()
```

```
{
```

```
    func_1();
```

```
    func_1();
```

```
    func_1();
```

```
    // signal to operating system everything works fine
```

```
    return 0;
```

```
}
```

```
void func_1()
```

```
{
```

```
    int a = 1;
```

```
    static int b = 100;
```

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

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

```
    a++;
```

```
    b++;
```

```
}
```



**Expected Output:**

```
a = 1  
b = 100
```

```
a = 1  
b = 101
```

```
a = 1  
b = 102
```

In `func_1()`, the variable `b` is declared as a static. When `func_1()` is called for the first time `b` is initialized to 100, in line 22, the value of `b` is incremented. This new value of `b` will be retained the next time the `func_1()` is called. When `func_1()` is called the second time, the variable `b` has retained its value which was 101, line 20, proves it by printing the value of `b` and once again the value of `b` is incremented by 1. similarly, when the third time `func_1()` is called, the value of `b` is 102. Note that only variable `b` is able to retain its value because variable `b` is declared as static, However, this is not the case with variable `a`, which is initialized every time when `func_1()` is called. Also, note that static variable `b` is initialized only once when `func_1()` is called for the first time.

**Register variables:** It tell the compiler to store the variable in CPU register instead of memory. Frequently used variables are kept in registers and they have faster accessibility. We can never get the addresses of these variables. “register” keyword is used to declare the register variables.

**Scope** – They are local to the function.

**Default value** – Default initialized value is the garbage value.

**Lifetime** – Till the end of the execution of the block in which it is defined.

### Example

```
#include <stdio.h>
int main() {
    register char x = 'S';
    register int a = 10;
    auto int b = 8;
    printf("The value of register variable b : %c\n",x);
    printf("The sum of auto and register variable : %d",(a+b));
    return 0;
}
```

### Output

The value of register variable b : S

The sum of auto and register variable : 18