

Unit-IV

Syllabus:

Function types: Library functions (math, string), user defined functions: Function definition, function declaration, arguments, scope rules and lifetime of variables, function calls and return.

Function in C:

A **function** is a block of code that performs a particular task.

There are many situations where we might need to write same line of code for more than once in a program. This may lead to unnecessary repetition of code, bugs and even becomes boring for the programmer. So, C language provides an approach in which you can declare and define a group of statements once in the form of a function and it can be called and used whenever required. These functions defined by the user are also known as **User-defined Functions**

C Library Functions

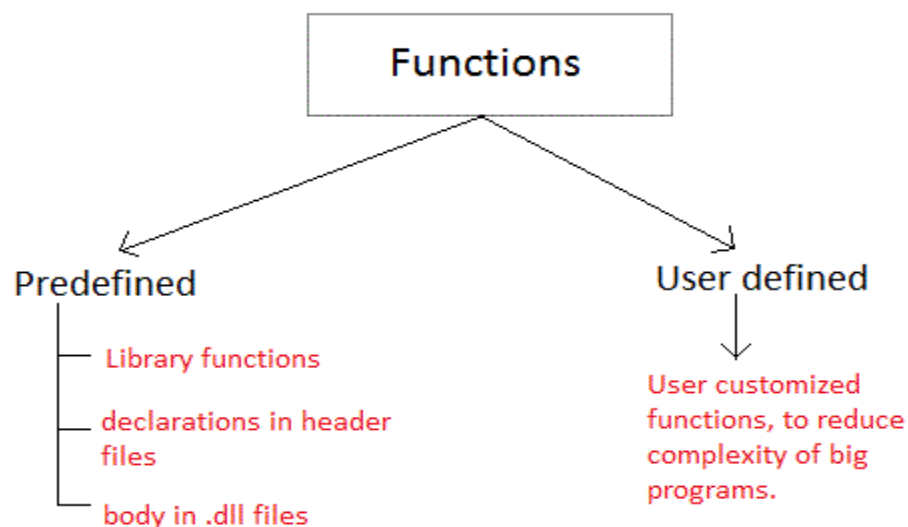
The Standard Function Library in C is a huge library of sub-libraries, each of which contains the code for several functions. In order to make use of these libraries, link each library in the broader library through the use of header files. The definitions of these functions are present in their respective header files. In order to use these functions, we have to include the header file in the program. Below are some header files with descriptions:

S No.	Header Files	Description
1	<assert.h>	It checks the value of an expression that we expect to be true under normal circumstances. If the expression is a nonzero value, the assert macro does nothing.
2	<complex.h>	A set of functions for manipulating complex numbers.
3	<float.h>	Defines macro constants specifying the implementation-specific properties of the floating-point library.

S No.	Header Files	Description
4	<limits.h>	These limits specify that a variable cannot store any value beyond these limits, for example- An unsigned character can store up to a maximum value of 255.
5	<math.h>	The math.h header defines various mathematical functions and one macro. All the Functions in this library take double as an argument and return double as the result.
6	<stdio.h>	The stdio.h header defines three variable types, several macros, and various function for performing input and output.
7	<time.h>	Defines date and time handling functions.
8	<string.h>	Strings are defined as an array of characters. The difference between a character array and a string is that a string is terminated with a special character '\0'.

C functions can be classified into two categories,

1. **Library functions**
2. **User-defined functions**



Library functions are those functions which are already defined in C library, example printf(), scanf(), strcat() etc. You just need to include appropriate header files to use these functions. These are already declared and defined in C libraries.

A **User-defined functions** on the other hand, are those functions which are defined by the user at the time of writing program. These functions are made for code reusability and for saving time and space.

Benefits of Using Functions

1. It provides modularity to your program's structure.
2. It makes your code reusable. You just have to call the function by its name to use it, wherever required.
3. In case of large programs with thousands of code lines, debugging and editing becomes easier if you use functions.
4. It makes the program more readable and easy to understand.

String:

String is a sequence of characters that are treated as a single data item and terminated by a null character '\0'. Remember that the C language does not support strings as a data type. A **string** is actually a one-dimensional array of characters in C language. These are often used to create meaningful and readable programs.

For example: The string "home" contains 5 characters including the '\0' character which is automatically added by the compiler at the end of the string.

char str[] = " HOME "					
Index ---->	0	1	2	3	4
str ---->	H	O	M	E	\0
Address ---->	0x23451	0x23452	0x23453	0x23454	0x23455

Declaring and Initializing a string variables:

// valid declaration

```
char name[10] = "StudyHard";
```

```
char name[10] = {'c','o','d','e','\0'};
```

// Illegal declaration

```
char ch[3] = "hello";
```

```
char str[4];
```

```
str = "hello";
```

String Input and Output:

- `%s` format specifier to read a string input from the terminal.
- But `scanf()` function, terminates its input on the first white space it encounters.
- **edit set conversion code** `%[..]` that can be used to read a line containing a variety of characters, including white spaces.
- The `gets()` function can also be used to read character string with white spaces

Program 1:

```
char str[20];
printf("Enter a string");
scanf("%[^\n]", &str);
printf("%s", str);
```

Program 2:

```
char text[20];
gets(text);
printf("%s", text);
```

String Handling Functions:

C language supports a large number of string handling functions that can be used to carry out many of the string manipulations. These functions are packaged in the **string.h** library. Hence, you must include **string.h** header file in your programs to use these functions.

Functions	Description
strcat()	It is used to concatenate(combine) two strings
strlen()	It is used to show the length of a string
strrev()	It is used to show the reverse of a string
strcpy()	Copies one string into another
strcmp()	It is used to compare two string

strcat() function in C:

Before

str1 -->

H	E	L	L	O
---	---	---	---	---

str2 -->

W	O	R	L	D
---	---	---	---	---

After strcat()

H	E	L	L	O	W	O	R	L	D
---	---	---	---	---	---	---	---	---	---

Syntax:

```
strcat("hello", "world");
```

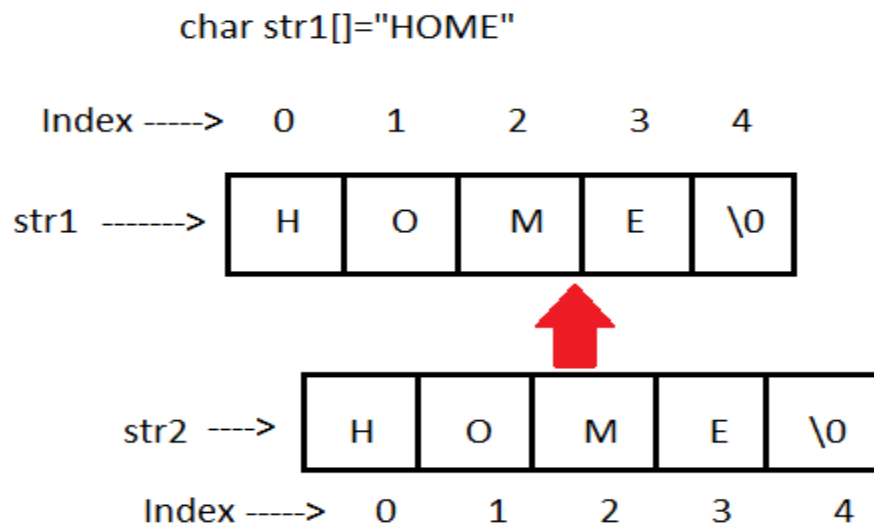
strlen() and strcmp() function:

strlen() will return the length of the string passed to it and **strcmp()** will return the ASCII difference between first unmatched character of two strings.

```
int j = strlen("studyhard");
int i = strcmp("study ", "hard");
printf("%d %d", j, i);
```

strcpy() function:

It copies the second string argument to the first string argument.



Example:

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

int main()
{
    char s1[50], s2[50];

    strcpy(s1, "StudyHard");
    strcpy(s2, s1);

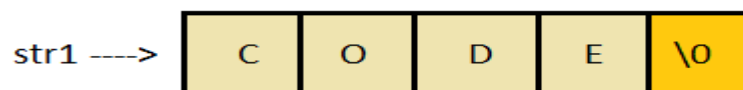
    printf("%s\n", s2);

    return(0);
}
```

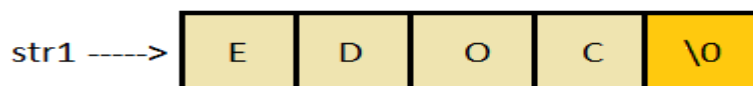
strrev() function:

It is used to reverse the given string expression.

Before



After strrev(str1)



Example:

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

int main()
{
    char s1[50];

    printf("Enter your string: ");
    gets(s1);
    printf("\nYour reverse string is: %s",strrev(s1));
    return(0);
}
```

Function Declaration

General syntax for function declaration is,

returntype functionName(type1 parameter1, type2 parameter2,...);

Like any variable or an array, a function must also be declared before its used. Function declaration informs the compiler about the function name, parameters is accept, and its return type. The actual body of the function can be defined separately. It's also called as **Function Prototyping**.

Function declaration consists of 4 parts.

- returntype
- function name
- parameter list
- terminating semicolon

returntype

When a function is declared to perform some sort of calculation or any operation and is expected to provide with some result at the end, in such cases, a return statement is added at the end of function body. Return type specifies the type of value(int, float, char, double) that function is expected to return to the program which called the function.

Note: In case your function doesn't return any value, the return type would be void.

functionName

Function name is an [identifier](#) and it specifies the name of the function. The function name is any valid C identifier and therefore must follow the same naming rules like other variables in C language.

parameter list

The parameter list declares the type and number of arguments that the function expects when it is called. Also, the parameters in the parameter list receives the argument values when the function is called. They are often referred as **formal parameters**.

Example

Let's write a simple program with a main() function, and a user defined function to multiply two numbers, which will be called from the main() function.

```
#include<stdio.h>

int multiply(int a, int b);    // function declaration

int main()
{
    int i, j, result;
    printf("Please enter 2 numbers you want to multiply...");
    scanf("%d%d", &i, &j);

    result = multiply(i, j);    // function call
    printf("The result of multiplication is: %d", result);

    return 0;
}

int multiply(int a, int b)
{
    return (a*b);    // function definition, this can be done in one line
}
```

Function definition Syntax

Just like in the example above, the general syntax of function definition is,

```
returntype functionName(type1 parameter1, type2 parameter2,...)
{
    // function body goes here
}
```

The first line *returntype* **functionName**(type1 parameter1, type2 parameter2,...) is known as **function header** and the statement(s) within curly braces is called **function body**.

Note: While defining a function, there is no semicolon(;) after the parenthesis in the function header, unlike while declaring the function or calling the function.

functionbody

The function body contains the declarations and the statements(algorithm) necessary for performing the required task. The body is enclosed within curly braces { ... } and consists of three parts.

- **local** variable declaration(if required).
- **function statements** to perform the task inside the function.
- a **return** statement to return the result evaluated by the function(if return type is void, then no return statement is required).

Calling a function

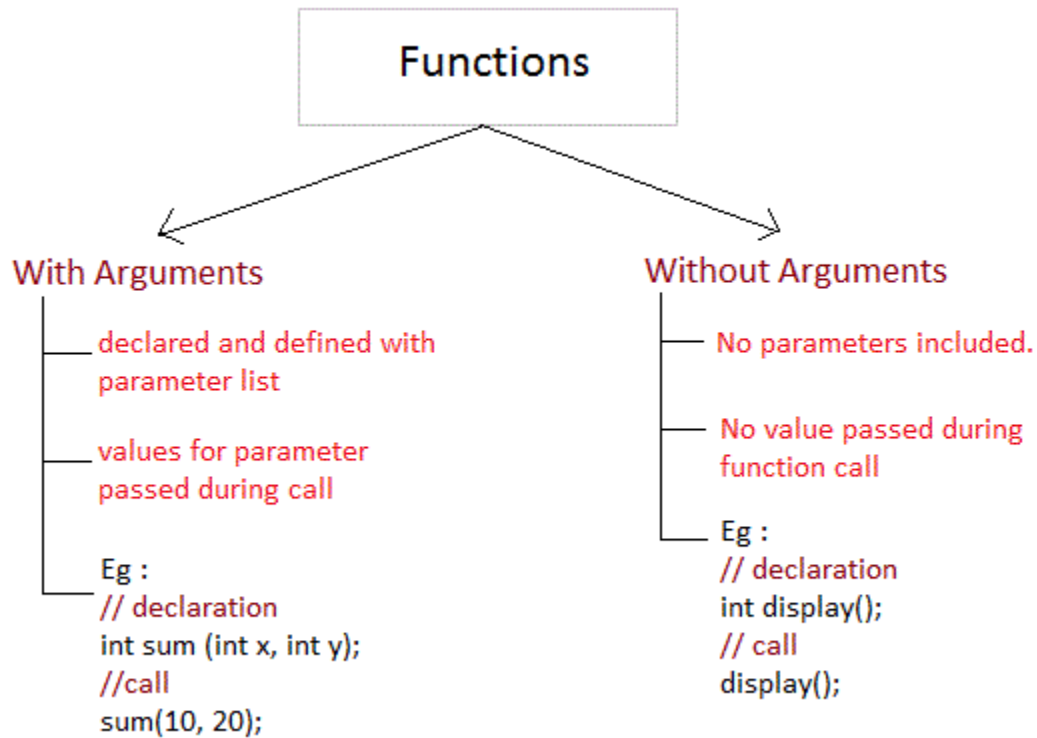
When a function is called, control of the program gets transferred to the function.

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

In the example above, the statement multiply(i, j); inside the main() function is function call.

Passing Arguments to a function

Arguments are the values specified during the function call, for which the formal parameters are declared while defining the function.




It is possible to have a function with parameters but no return type. It is not necessary, that if a function accepts parameter(s), it must return a result too.

```
#include<stdio.h>

int multiply(int a, int b);

int main()
{
    ... ..
    result = multiply(i, j);
    ... ..
}

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



providing arguments while calling function

While declaring the function, we have declared two parameters a and b of type int. Therefore, while calling that function, we need to pass two arguments, else we will get compilation error. And the two arguments passed should be received in the function definition, which means that the function header in the function definition should have the two parameters to hold the argument values. These received arguments are also known as **formal parameters**. The name of the variables while declaring, calling and defining a function can be different.

Returning a value from function


A function may or may not return a result. But if it does, we must use the return statement to output the result. return statement also ends the function execution, hence it must be the last statement of any function. If you write any statement after the return statement, it won't be executed.

```
#include<stdio.h>

int multiply(int a, int b);

int main()
{
    ... ..
    result = multiply(i, j);
    ... ..
}

int multiply(int a, int b)
{
    ... ..
    return a*b;
}
```



The value returned by the function must be stored in a variable.

The datatype of the value returned using the return statement should be same as the return type mentioned at function declaration and definition. If any of it mismatches, you will get compilation error.

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

Below, we will discuss about all these types, along with program examples.

1. 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);
    }
}
```

2. 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;
}
```


3. Function with arguments and no return value

We 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);
    }
}
```

4. 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;
    }
}
```

Nesting of Functions

C language also allows nesting of functions i.e to use/call one function inside another function's body. We must be careful while using nested functions, because it may lead to infinite nesting.

```
function1()
{
    // function1 body here

    function2();

    // function1 body here
}
```

If function2() also has a call for function1() inside it, then in that case, it will lead to an infinite nesting. They will keep calling each other and the program will never terminate. Not able to understand? Lets consider that inside the main() function, function1() is called and its execution starts, then inside function1(), we have a call for function2(), so the control of program will go to the function2(). But as function2() also has a call to function1() in its body, it will call function1(), which will again call function2(), and this will go on for infinite times, until you forcefully exit from program execution.

What is Recursion?

Recursion is a special way of nesting functions, where a function calls itself inside it. We must have certain conditions in the function to break out of the recursion, otherwise recursion will occur infinite times.

```
function1()
{
    // function1 body
    function1();
    // function1 body
}
```

Example: Factorial of a number using Recursion

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

```
int factorial(int x);    //declaring the function
```

```
void main()
```

```
{
```

```
    int a, b;
```

```
    printf("Enter a number...");
```

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

```
    b = factorial(a);    //calling the function named factorial
```

```
    printf("%d", b);
```

```
}
```

```
int factorial(int x) //defining the function
```

```
{
```

```
    int r = 1;
```

```
    if(x == 1)
```

```
        return 1;
```

```
    else
```

```
        r = x*factorial(x-1);    //recursion, since the function calls itself
```

```
    return r;
```

```
}
```

Types of Function calls in C

Functions are called by their names, we all know that, then what is this tutorial for? Well if the function does not have any arguments, then to call a function you can directly use its name. But for functions with arguments, we can call a function in two different ways, based on how we specify the arguments, and these two ways are:

1. Call by Value
 2. Call by Reference
-

Call by Value

Calling a function by value means, we pass the values of the arguments which are stored or copied into the formal parameters of the function. Hence, the original values are unchanged only the parameters inside the function changes.

```
#include<stdio.h>
void calc(int x);
int main()
{
    int x = 10;
    calc(x);
    // this will print the value of 'x'
    printf("\nvalue of x in main is %d", x);
    return 0;
}

void calc(int x)
{
    // changing the value of 'x'
    x = x + 10 ;
    printf("value of x in calc function is %d ", x);
}
```

Output: value of x in calc function is 20
 value of x in main is 10

In this case, the actual variable x is not changed. This is because we are passing the argument by value, hence a copy of x is passed to the function, which is updated during function execution, and that copied value in the function is destroyed when the function ends(goes out of scope). So the variable x inside the main() function is never changed and hence, still holds a value of 10.

But we can change this program to let the function modify the original x variable, by making the function calc() return a value, and storing that value in x.

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

```
int calc(int x);
```

```
int main()
{
    int x = 10;
    x = calc(x);
    printf("value of x is %d", x);
    return 0;
}
```

```
int calc(int x)
{
    x = x + 10 ;
    return x;
}
```

Output: value of x is 20

Call by Reference

In call by reference we pass the address(reference) of a variable as argument to any function. When we pass the address of any variable as argument, then the function will have access to our variable, as it now knows where it is stored and hence can easily update its value.

In this case the formal parameter can be taken as a **reference** or a **pointers** (don't worry about pointers, we will soon learn about them), in both the cases they will change the values of the original variable.

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

```
void calc(int *p);    // functin taking pointer as argument
```

```
int main()
```

```
{
```

```
    int x = 10;
```

```
    calc(&x);    // passing address of 'x' as argument
```

```
    printf("value of x is %d", x);
```

```
    return(0);
```

```
}
```

```
void calc(int *p)    //receiving the address in a reference pointer variable
```

```
{
```

```
    /*
```

```
        changing the value directly that is  
        stored at the address passed
```

```
    */
```

```
    *p = *p + 10;
```

```
}
```

Output: value of x is 20

Scope Rules and Lifetime of a Variable in C

Scope, Visibility and Lifetime of variables in C Language are very much related to each other, but still, there are some different properties associated with them that make them distinct. **Scope** determines the region in a C program where a variable is *available* to use, **Visibility** of a variable is related to the *accessibility* of a variable in a particular scope of the program and **Lifetime** of a variable is for how much *time* a variable remains in the system's memory.

Scope

- Scope and types of scope of a variable in C Language.
- Visibility of a variable in C program.
- Lifetime of a variable in C program.

Introduction

Scope, Visibility and Lifetime can be understood with a simple real-life example of Netflix, Hotstar or Amazon Prime. There are Movies / TV Series present on these platforms that have local/global scope, visibility and a lifetime. Movies based on local languages like Hindi, Japanese, and Korean have a limited scope. They can be watched (accessible) in a limited area, while movies with global languages like English have a global scope and are available throughout the world to watch. These movies also have a lifetime, and once that lifetime is over these movies are removed from the streaming platforms.

Coming to the appropriate definitions for Scope, Visibility and Lifetime of a variable :

- **Scope** is defined as the availability of a variable inside a program, scope is basically the region of code in which a variable is *available* to use. There are four types of scope:
 - *file scope*,
 - *block scope*,
 - *function scope* and
 - *prototype scope*.
- **Visibility** of a variable is defined as if a variable is *accessible* or not inside a particular region of code or the whole program.

- **Lifetime of a variable** is the *time* for which the variable is taking up a *valid space* in the system's memory, it is of three types:
 - *static lifetime*,
 - *automatic lifetime* and
 - *dynamic lifetime*.

What is the Scope of Variables in C?

Let's say you reside in an apartment complex, and you only have one key to use to get access to your room. The apartment's owner/manager may also have a master key that grants access to all the rooms.

The scope of variables in C have a similar idea. The availability of a variable in a program or function is referred to as the scope of variable in C language.

A variable, for example, may be accessed only within a single function/block of code (your apartment key) or throughout the whole C program (the shared access key). We can associate the room keys with the *local variables* in C language because they only work in that single room. The term *global variables* refers to variables (keys) that are accessible to the whole program (apartment complex).

Four types of scope of a variable in C:

1. File Scope

File scope of variables in C is defined as having the *availability* of the variable throughout the file/program. It means that the variable has a global scope, and it is available all around for every function and every block in the program.

Example :

```
#include <stdio.h>

// variable with file scope
int x = 10;

void func() {
    // x is available in func() function,
    // x now equals 10 + 10 = 20
    x += 10;
    printf("Value of x is %d\n", x);
}

int main() {

    func();
    // x is also available in main() function
    x += 30; // x now equals 20 + 30 = 50
    printf("Value of x is %d", x);
    return 0;
}
```

Output :

Value of x is 20

Value of x is 50

Explanation : A global variable x is declared having file scope. main() function and func() function are able to access the variable x because it has *file scope*. In the code, first we have increased the value of x by 10 (x = 20 now) and then by 30 (x =

50 now), and we can see from the output that x preserves its value 20 after the changes made in the function func() because it has file scope.

2. Block Scope

Block scope of variables in C is defined as when the variable has a limited scope, and the memory occupied by the variable will be deleted once the execution of the block ends. The variable is not accessible or available outside the block. A block of code can be defined in curly braces {code_block}.

Example :

```
#include <stdio.h>

int main() {
    int a = 5;
    int b = 10;

    // inner block of code having block scope
    {
        int sum = a + b;
        printf("Sum of a and b: %d", sum);
    }

    // the below statement will throw an error because,
    // sum variable is not available outside the scope of above block,
    // printf("Sum of a and b: %d", sum);

    return 0;
}
```

Output :

Sum of a and b: 15

Explanation : In the above program, the sum variable has block scope, we can't access the sum variable outside this block where sum is declared. If we uncomment the printf("Sum of a and b: %d", sum); statement then the compiler will throw an error of undeclared sum variable in the current scope.

3. Function Scope

Function scope of variables in C begins with the left curly brace { and ends with a closing right curly brace }. A variable declared inside a function has a function scope. It has been allocated memory when the function is called, and once the function returns something, the function execution ends and with it, the variable goes out of scope, i.e. it gets deleted from the memory.

Example :

```
#include <stdio.h>

void findAge() {
    // the age variable is not accessible outside the function findAge()
    // as it is having local scope to the function i.e. function scope
    int age = 18;
}

int main() {
    printf("Age is %d", age);
    return 0;
}
```

You can run and check code it will display error;

Error :

```
prog.c: In function 'main':
prog.c:11:25: error: 'age' undeclared (first use in this function)
    printf("Age is %d", age);
                        ^
prog.c:11:25: note: each undeclared identifier is reported only once for each
function it
```

Explanation : We can see in the output that a compilation error is thrown to us because we are using an age variable outside the function from where it is declared. Compiler has thrown an error because age variable is not available outside the function as age variable only has function scope.

4. Function Prototype Scope

Function prototype scope of variables in C are declared in some function as its parameters. These variables are similar to the function scope variables where a variable's memory gets deleted once the function execution terminates.

Example :

```
#include <stdio.h>
// variables a and b are available only inside the function and
// both variables have function prototype scope
int findSum(int a, int b) {
    return a + b;
}

int main() {
    int sum = findSum(3, 5);
    printf("Sum of 3 and 5 is %d", sum);
    return 0;
}
```

Output :

```
Sum of 3 and 5 is 8
```

Explanation : When the findSum() function is invoked, space for variables a and b is allocated in the system's memory. Variables a and b are findSum() function parameters and these variables have function prototype scope, we can access these variables only in the function definition and not outside of it.

Scope Rules

Scope of variables in C also have some rules demonstrated below. Variables can be declared on three places in a C Program :

1. Variables that are declared inside the function or a block of code are known as *local variables*.
2. Variables that are declared outside of any function are known as *global variables* (usually at the start of a program).
3. Variables that are declared in the definition of a function parameters as its *formal parameters*.

1. Local Variables

When we declare variables inside a function or an inner block of code, then these variables are known as local variables. They can only be used inside the *function scope* or the *block scope*.

Example :

```
#include <stdio.h>

int main() {
    int sum = 0;

    // inner block of code having block scope
    {
        int a = 5;
        int b = 10;
    }
    // this statement will throw an error because a and b are not available outside the
    // above block of code
    sum = a + b;

    return 0;
}
```

You can run and check code it will give error:

Output :

```
prog.c: In function 'main':
prog.c:12:8: error: 'a' undeclared (first use in this function)
    sum = a + b;
         ^
prog.c:12:8: note: each undeclared identifier is reported only once for each
function it appears in
prog.c:12:12: error: 'b' undeclared (first use in this function)
    sum = a + b;
```

Explanation:

We can see in the output that a compilation error is showing to us because we are

using a and b variables outside the block where they are declared. It has thrown an error because a and b are not available outside the above block of code.

2. Global Variables

When we declare variables outside of all the functions then these variables are known as global variables. These variables always have *file scope* and can be *accessed* anywhere in the program, they also remain in the memory until the execution of our program finishes.

Example :

```
#include <stdio.h>

// global variable
int side = 10;

int main() {
    int squareArea = side * side;

    printf("Area of Square : %d", squareArea);

    return 0;
}
```

Output :

```
Area of Square : 100
```

Explanation : As we have declared a side variable outside of the main() function, it has *file scope* or it is known as a *global variable*. We are using the side variable to calculate the area of the square in the main() function.

3. Formal Parameters

When we define a function and declare some variables in the function parameters, then these parameters are known as formal parameters/variables. They have a function prototype scope.

Example:

```
#include <stdio.h>

// variable n is a formal parameter
int square(int n) {
    return n * n;
}

int main() {
    int number = 5;
    printf("Square of %d is %d.", number, square(number));
    return 0;
}
```

Output :

```
Square of 5 is 25.
```

Explanation : We have defined a function `square()` to calculate the square of a number. The `int n` variable in the `square()` function parameters is a *formal parameter* having *function prototype scope*.

Rules of use

1. Global variables have a file scope, i.e. they are available for the whole program file.
2. The scope of a local variable in C starts at the declaration point and concludes at the conclusion of the block or a function/method where it is defined.

3. The scope of formal parameters is known as function prototype scope and it is the same as its function scope, we can access the variables only in the function definition and not outside it.
4. Although the scope of a static local variable is confined to its function, its lifetime extends to the end of program execution.

What is the Lifetime of a variable in C?

Lifetime of a variable is defined as for how much time period a variable occupies a valid space in the system's memory or lifetime is the period between when memory is allocated to hold the variable and when it is freed. Once the variable is out of scope its lifetime ends. Lifetime is also known as the range/extent of a variable.

A variable in the C programming language can have a

- *static*,
- *automatic*, or
- *dynamic lifetime*.

a. Static Lifetime

Objects/Variables having static lifetime will remain in the memory until the execution of the program finishes. These types of variables can be declared using the static keyword, global variables also have a static lifetime: they survive as long as the program runs.

Example :

```
static int count = 0;
```

The count variable will stay in the memory until the execution of the program finishes.

b. Automatic Lifetime

Objects/Variables declared inside a block have automatic lifetime. Local variables (those defined within a function) have an automatic lifetime by default: they arise when the function is invoked and are deleted (together with their values) once the

function execution finishes.

Example :

```
{ // block of code
    int auto_lifetime_var = 0;
}
```

auto_lifetime_var will be deleted from the memory once the execution comes out of the block.

c. Dynamic Lifetime

Objects/Variables which are made during the run-time of a C program using the [Dynamic Memory Allocation](#) concept using the malloc(), calloc() functions in C or new operator in C++ are stored in the memory until they are explicitly removed from the memory using the free() function in C or delete operator in C++ and these variables are said to have a dynamic lifetime. These variables are stored in the *heap* section of our system's memory, which is also known as the *dynamic memory*.

Example :

```
int *ptr = (int *)malloc(sizeof(int));
```

Memory block (variable) pointed by ptr will remain in the memory until it is explicitly freed/removed from the memory or the program execution ends.

Example Program with Functions and Parameters

C Program :

```
#include<stdio.h>

int num1, num2;
void func2(int x) {
    int j, k;
    num2 = 2 * x;
    x += 1;
    printf("num2: %d, x: %d", num2, x);
}

int func1(int a, int b) {
    int num2, j;
    func2(num1);
}

int main() {
    int var1, var2;
    num1 = 5, var1 = 3, var2 = 2;
    func1(var1, var2);
    return 0;
}
```

You can run and check your code [here](#). (IDE by InterviewBit)

Output :

```
num2: 10, x: 6
```

Explanation :

- Variables num2 and j as well as function parameters a and b are local to the function func1() and have function scope and prototype scope respectively.
- Variables declared in function func2() and main() function are not visible in func1().
- The variables num1 and num2 are global variables having file scope.
- num1 is visible in func1() but num2 is hidden/invisible by the local declaration of num2 i.e. int num2.

What is the Difference Between Scope and Lifetime of a Variable in C?

Scope Of A Variable in C	Lifetime Of A Variable in C
Scope of a variable determines the area or a region of code where a variable is available to use.	Lifetime of a variable is defined by the time for which a variable occupies some valid space in the system's memory.
Scope determines the life of a variable.	Life of a variable depends on the scope.
Scope of a static variable in a function or a block is limited to that function.	A static variable stays in the memory and retains its value until the program execution ends irrespective of its scope.
Scope is of four types: file, block, function and prototype scope.	Lifetime is of three types: static, auto and dynamic lifetime.