UNIT III : Function

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

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

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

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

A function can also be referred as a method or a sub-routine or a procedure, etc.

Defining a Function

The general form of a function definition in C programming language is as follows:

return_type function_name(parameter list)

{

body of the function

}

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

- Return Type: A function may return a value. The return_type is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the return_type is the keyword void.
- Function Name: This is the actual name of the function. The function name and the parameter list together constitute the function signature.
- Parameters: A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.
- Function Body: The function body contains a collection of statements that define what the function does.

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 Declarations

A function declaration tells the compiler about a function name and how to call the function. The actual body of the function can be defined separately.

A function declaration has the following parts:

```
return_type function_name( parameter list );
```

For the above defined function max(),the function declaration is as follows:

```
int max(int num1, int num2);
```

Parameter names are not important in function declaration, only their type is required, so the following is also a valid declaration:

```
int max(int, int);
```

Function declaration is required when you define a function in one source file and you call that function in another file. In such case, you should declare the function at the top of the file calling the function.

Calling a Function

While creating a C function, you give a definition of what the function has to do. To use a function, you will have to call that function to perform the defined task.

When a program calls a function, the program control is transferred to the called function. A called function performs a defined task and when its return statement is executed or when its function-ending closing brace is reached, it returns the program control back to the main program.

To call a function, you simply need to pass the required parameters along with the function name, and if the function returns a value, then you can store the returned value. For example:

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

```
/* function declaration */
int max(int num1, int num2);
int main ()
{
/* local variable definition */
int a = 100;
int b = 200;
int ret;
/* calling a function to get max value */ ret = max(a, b);
printf( "Max value is : %d\n", ret );
return 0;
}
/* function returning the max between two numbers */ int max(int num1, int num2) {
/* local variable declaration */
int result;
if (num1 > num2)
result = num1;
else
result = num2;
return result;
}
```

We have kept max()along with main() and compiled the source code. While running the final executable, it would produce the following result:

Max value is : 200

Function Arguments

If a function is to use arguments, it must declare variables that accept the values of the arguments. These variables are called the formal parameters of the function. Formal parameters behave like other local variables inside the function and are created upon entry into the function and destroyed upon exit. While calling a function, there are two ways in which arguments can be passed to a function:

Call Type	Description
Call by value	This method copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.
Call by reference	This method copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument.

Call by Value

The call by value method of passing arguments to a function copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.

By default, C programming uses call by value to pass arguments. In general, it means the code within a function cannot alter the arguments used to call the function. Consider the function swap() definition as follows.

```
/* function definition to swap the values */ void swap(int x, int y) {
int temp;
temp = x; /* save the value of x */
x = y;   /* put y into x */
y = temp; /* put temp into y */
return;
}
```

Now, let us call the function swap() by passing actual values as in the following example:

```
#include <stdio.h>
/* function declaration */
void swap(int x, int y);
int main ()
{
/* local variable definition */
int a = 100;
int b = 200;
printf("Before swap, value of a : %d\n", a );
printf("Before swap, value of b : %d\n", b );
/* calling a function to swap the values */ swap(a, b);
printf("After swap, value of a : %d\n", a );
printf("After swap, value of b : %d\n", b );
return 0;
}
```

Let us put the above code in a single C file, compile and execute it, it will produce the following result:

```
Before swap, value of a :100
Before swap, value of b :200
After swap, value of a :100
After swap, value of b :200
```

It shows that there are no changes in the values, though they had been changed inside the function.

Call by Reference

The call by reference method of passing arguments to a function copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. It means the changes made to the parameter affect the passed argument.

To pass a value by reference, argument pointers are passed to the functions just like any other value. So accordingly, you need to declare the function parameters as pointer types as in the following function swap(), which exchanges the values of the two integer variables pointed to, by their arguments.

```
/* function definition to swap the values */ void swap(int *x, int *y) {
int temp; temp = *x; *x = *y; *y = temp;
/* save the value at address x */ /* put y into x */ /* put temp into y */
return;
}
```

Let us now call the function swap() by passing values by reference as in the following example:

```
#include <stdio.h>
/* function declaration */
void swap(int *x, int *y);
int main ()
{
/* local variable definition */
int a = 100;
int b = 200;
printf("Before swap, value of a : %d\n", a );
printf("Before swap, value of b : %d\n", b );
/* calling a function to swap the values. &a indicates pointer to a i.e. address of variable a and &b indicates pointer to b i.e. address of variable b.
*/
swap(&a, &b);
printf("After swap, value of a : %d\n", a );
printf("After swap, value of b : %d\n", b );
return 0;
}
```

Let us put the above code in a single C file, compile and execute it, to produce the following result:

```
Before swap, value of a :100
Before swap, value of b :200
After swap, value of a :200
After swap, value of b :100
```

It shows that the change has reflected outside the function as well, unlike call by value where the changes do not reflect outside the function.

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

Storage Class

Storage class in C decides the part of storage to allocate memory for a variable, it also determines the scope of a variable. All variables defined in a C program get some physical location in memory where variable's value is stored. Memory and CPU registers are types of memory locations where

a variable's value can be stored. The storage class of a variable in C determines the life time of the variable if this is 'global' or 'local'. Along with the life time of a variable, storage class also determines variable's storage location (memory or registers), the scope (visibility level) of the variable, and the initial value of the variable.

These specifiers tell the compiler how to store the subsequent variable. The general form of a variable declaration that uses a storage class is shown here:
storage_class_specifier data_type variable_name;

External, Auto, Local, Static, Register Variables

Every variable in C programming has two properties: type and storage class.
Type refers to the data type of a variable. And, storage class determines the scope, visibility and lifetime of a variable.
There are 4 types of storage class:

1. automatic
2. external
3. static
4. register

At most one storage class specifier may be given in a declaration. If no storage class specifier is specified then following rules are used:

1. Variables declared inside a function are taken to be auto.
2. Functions declared within a function are taken to be extern.
3. Variables and functions declared outside a function are taken to be static, with external linkage

Automatic Storage Class

A variable defined within a function or block with auto specifier belongs to automatic storage class. All variables defined within a function or block by default belong to automatic storage class if no storage class is mentioned. Variables having automatic storage class are local to the block which they are defined in, and get destroyed on exit from the block.
The following C program demonstrates the visibility level of auto variables.

Example:

```
#include <stdio.h>
int main( )
{
  auto int i = 1;
  {
    auto int i = 2;
    {
      auto int i = 3;
      printf ( "\n%d ", i);
    }
    printf ( "%d ", i);
  }
  printf( "%d\n", i);
}
```

OUTPUT
3 2 1

Static Storage Class
The static specifier gives the declared variable static storage class. Static variables can be used within function or file.Unlike global variables, static variables are not visible outside their function or file, but they maintain their values between calls. The static specifier has different effects upon local and global variables

Register Storage Class

The register specifier declares a variable of register storage class. Variables belonging to register storage class are local to the block which they are defined in, and get destroyed on exit from the block. A register declaration is equivalent to an auto declaration, but hints that the declared variable will be accessed frequently; therefore they are placed in CPU registers, not in memory. Only a few variables are actually placed into registers, and only certain types are eligible; the restrictions are implementation-dependent. However, if a variable is declared register, the unary & (address of) operator may not be applied to it, explicitly or implicitly. Register variables are also given no initial value by the compiler.
Example:

```
#include <stdio.h>
 int main()
{
  register int i = 10;
  int *p = &i; //error: address of register variable requested
   printf("Value of i: %d", *p);
  printf("Address of i: %u", p);

}
```

External Storage Class

The extern specifier gives the declared variable external storage class. The principal use of extern is to specify that a variable is declared with external linkage elsewhere in the program.
When extern specifier is used with a variable declaration then no storage is allocated to that variable and it is assumed that the variable has already been defined elsewhere in the program. When we use extern specifier the variable cannot be initialized because with extern specifier variable is declared, not defined.

```
#include <stdio.h>
extern int x;
int main()
{
  printf("x: %d\n", x);
}
int x = 10;
```

SCOPE RULES

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

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

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

Local Variables

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

```
#include <stdio.h>
int main ()
{
/* local variable declaration */
int a, b;
int c;
/* actual initialization */
a = 10;
b = 20;
c = a + b;
printf ("value of a = %d, b = %d and c = %d\n", a, b, c);
return 0;
}
```

Global Variables

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

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

```
#include <stdio.h>
/* global variable declaration */
```

```
int g;
int main ()
{
/* local variable declaration */
int a, b;
/* actual initialization */
a = 10;
b = 20;
g = a + b;
printf ("value of a = %d, b = %d and g = %d\n", a, b, g);
return 0;
}
```

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

```
#include <stdio.h>
/* global variable declaration */
int g = 20;

int main ()
{
/* local variable declaration */
int g = 10;
printf ("value of g = %d\n",  g);
return 0;
}
```

When the above code is compiled and executed, it produces the following result:
value of g = 10

Static Variable

A static variable is declared by using the static keyword. For example;
static int i;
The value of a static variable persists until the end of the program.

```
#include <stdio.h>
void display();
int main()
{
   display();
   display();
}
void display()
{
   static int c = 1;
   c += 5;
```

```
   printf("%d  ",c);
}
```

Output

6 11

Formal Parameters

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

```
#include <stdio.h>
/* global variable declaration */
int a = 20;
int main ()
{
/* local variable declaration in main function */
int a = 10;
int b = 20;
int c = 0;
printf ("value of a in main() = %d\n", a); c = sum( a, b);
printf ("value of c in main() = %d\n",c);
return 0;
}
/* function to add two integers */
int sum(int a, int b)
{
printf ("value of a in sum() = %d\n", a); printf ("value of b in sum() = %d\n", b);
return a + b;
}
```

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

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

Initializing Local and Global Variables

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

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

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

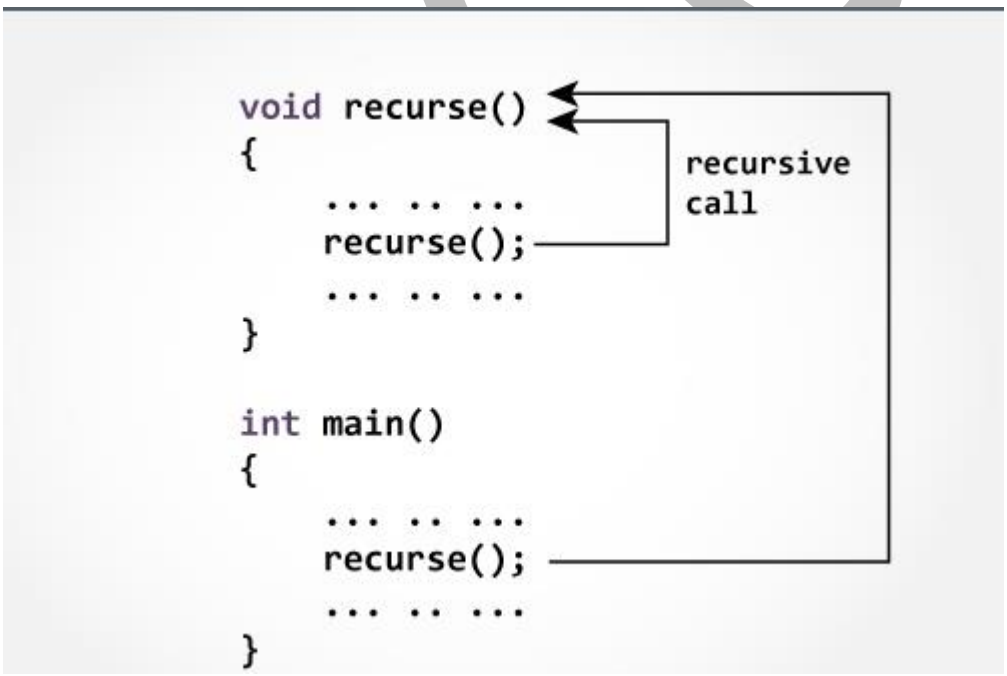
C main return as integer

Recursion

A function that calls itself is known as a recursive function. And, this technique is known as recursion.

```
void recurse()
{
   ... .. ...
   recurse();
   ... .. ...
}

int main()
{
   ... .. ...
   recurse();
   ... .. ...
}
```



The recursion continues until some condition is met to prevent it.

Example: Sum of Natural Numbers Using Recursion

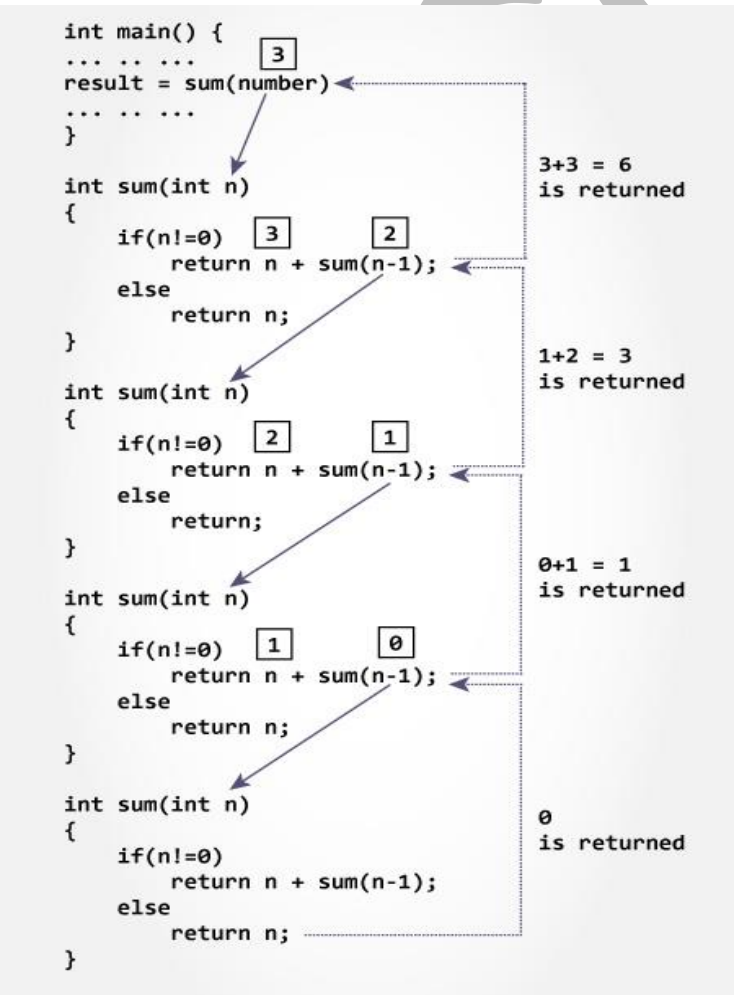
```
#include <stdio.h>
int sum(int n);
```

```
int main()
{
   int number, result;
   printf("Enter a positive integer: ");
   scanf("%d", &number);
   result = sum(number);
   printf("sum = %d", result);
   return 0;
}
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

sum = 6



Advantages

1. Reduce unnecessary calling of function.
2. Through Recursion one can Solve problems in easy way while its iterative solution is very big and complex.

Disadvantages

1. Recursive solution is always logical and it is very difficult to trace.(debug and understand).
2. In recursive we must have an if statement somewhere to force the function to return without the recursive call being executed, otherwise the function will never return.
3. Recursion takes a lot of stack space, usually not considerable when the program is small and running on a PC.
4. Recursion uses more processor time.

Preprocessor

The C preprocessor is a macro processor that is used automatically by the C compiler to transform your program before actual compilation (Proprocessor direcives are executed before compilation.). It is called a macro processor because it allows you to define macros, which are brief abbreviations for longer constructs. A macro is a segment of code which is replaced by the value of macro. Macro is defined by #define directive. Preprocessing directives are lines in your program that start with #. The # is followed by an identifier that is the directive name. For example, #define is the directive that defines a macro. Whitespace is also allowed before and after the #.

Sr.No.	Directive	Description
1	**#define**	Substitutes a preprocessor macro.
2	**#include**	Inserts a particular header from another file.
3	**#undef**	Undefines a preprocessor macro.
4	**#ifdef**	Returns true if this macro is defined.
5	**#ifndef**	Returns true if this macro is not defined
6	**#if**	Tests if a compile time condition is true.
7	**#else**	The alternative for #if.
8	**#elif**	#else and #if in one statement.
9	**#endif**	Ends preprocessor conditional.
10	**#error**	Prints error message on stderr
11	**#pragma**	Issues special commands to the compiler, using a standardized method.

Standard Library Functions

C Standard library functions or simply C Library functions are inbuilt functions in C programming. The prototype and data definitions of these functions are present in their respective header files. To use these functions we need to include the header file in our program.

Advantages of Using C library functions

1. They work

One of the most important reasons you should use library functions is simply because they work. These functions have gone through multiple rigorous testing and are easy to use.

2. The functions are optimized for performance

Since, the functions are "standard library" functions, a dedicated group of developers constantly make them better. In the process, they are able to create the most efficient code optimized for maximum performance.

3. It saves considerable development time

Since the general functions like printing to a screen, calculating the square root, and many more are already written. You shouldn't worry about creating them once again.

4. The functions are portable

With ever-changing real-world needs, your application is expected to work every time, everywhere. And, these library functions help you in that they do the same thing on every computer.

Example: Square root using sqrt() function

```
#include <stdio.h>
#include <math.h>
int main()
{
  float num, root;
  printf("Enter a number: ");
  scanf("%f", &num);
  // Computes the square root of num and stores in root.
  root = sqrt(num);
  printf("Square root of %.2f = %.2f", num, root);
  return 0;
}
```

Output:

```
Enter a number: 12
Square root of 12.00 = 3.46
```

Question Bank

3. Explain the use of %i format specifier w.r.t scanf().
4. what is modular programming.
5. Does a built-in header file contains built-in function definition?

6. What is the difference between actual and formal parameters?
7. Can a program be compiled without main() function?
8. What are static functions? What is their use?
9. In header files whether functions are declared or defined?
10. Out of fgets() and gets() which function is safe to use and why?
11. Difference between strdup and strcpy?
12. What is recursion?
13. How can you print a (backslash) using any of the printf() family of functions.
14. What are the different ways of passing parameters to the functions? Which to use when?
15. What is a static function?
16. What is the difference between Call by Value and Call by Reference?
17. How do you generate random numbers in C?
18. What could possibly be the problem if a valid function name such as tolower() is being reported by the C compiler as undefined?
19. What are actual arguments?
20. What does the function toupper() do?
21. Is it possible to have a function as a parameter in another function?
22. What is the difference between functions getch() and getche()?
23. Can you pass an entire structure to functions?
24. What is gets() function?
25. How do you search data in a data file using random access method?
26. Is there a built-in function in C that can be used for sorting data?
27. What is function prototype?
28. Where should type cast function not be used in C?
29. How many arguments can be passed to a function in C?
30. What is static function in C? If you want to execute C program even after main function is terminated, which function can be used?
31. Is it possible to call atexit() function more than once in a C program?
32. What is exit() function in C?
33. What is the difference between exit() and return() in C?
34. Is there any inbuilt library function in C to remove leading and trailing spaces from a string? How will you remove them in C?
35. What is the use of the function in C?
36. Example of call by value in C Program
37. Explain the needs of User-Defined Function.
38. What is Modular Programming? Explain its characteristics.
39. Define following terms: Function, Recursion
40. What is the purpose of external/global variable? What is its scope?
41. Explain scope and lifetime of variables.
42. Explain User-Defined Function in detail.
43. Explain different categories of function with example.
44. Explain Recursion in detail with example.
45. Explain Call by value and call by reference with example.
46. Write down following Differences : 1. Local variable Vs. global variables.
    1. Call by value Vs. Call by reference.
    2. Actual Arguments Vs. Formal Arguments.