Module 8: Functions

8.1 Introduction

- A Functions is a block of code that performs a specific task.
- There are many situations where we might need to write same line of code for more than once in program. This may lead to unnecessary repetition of code, bugs and even becomes boring for programmer.
- Sometimes our program gets bigger in size and it's not possible for a programmer to track which piece of code is doing what.
- So, C language provide a function concept. Function is a way to divide a complex problem into smaller chunks makes our program easy to understand and reuse.
- These functions defined by the user are also known as **User-defined Functions**.

8.1.1 What is Function

- Functions are sets of statements that take inputs, perform some operations, and produce results.
- A function is nothing but a group of code put together and given a name and it can be called anytime without writing the whole code again and again in a program.
- A function is a block of code that performs a specific task.
- Function can be called multiple or several times to provide reusability and modularity to the c program.
- Functions are also called Procedures or subroutines or methods.

8.1.2 why do we need Functions?

- Functions help us in **reducing code redundancy**. If functionality is performed at multiple places in software, then rather than writing the same code, again and again, we create a function and call it everywhere. This also helps in maintenance as we have to change at one place if we make future changes to the functionality.
- Functions make code **modular**. Consider a big file having many lines of code. It becomes really simple to read and use the code if the code is divided into functions.
- Functions provide **abstraction**. For example, we can use library functions without worrying about their internal work.

8.1.3 Important Points

- Execution of a C Program starts from main()
- A C Program can have more than one function.
- Every function gets called directly or indirectly from main()
- To use a function, we need to call a function.
- The function is the block of code that can be reused as many times as we want inside a program.
- Function declaration includes function name, return type, and parameters.
- Function definition includes the body of the function.
- Function is of two types user-defined function and library function.
- In function, we can according to two types call by value and call by reference according to the values passed.

8.1.4 Advantages of Functions in C

Functions in C is a highly useful feature of C with many advantages as mentioned below:

1. The function can reduce the repetition of the same statements in the program.
2. The function makes code readable.
3. There is no fixed number of calling functions it can be called as many times as you want.
4. The function reduces the size of the program.
5. Once the function is declared you can just use it without thinking about the internal working of the function.

8.1.5 Disadvantages of Function in C

Apart from the many advantages of functions, there are certain disadvantages of functions as the execution of the program becoming slower.

8.2 Classification of Function

C functions can be classified into two categories.

1) Standard Library Functions or Pre-defined Functions
2) User-defined functions

8.3 Standard Library Functions:

- Library functions are those functions which are already defined in C Library. Or The Standard library functions are built-in functions in C Programming.
- These functions are defined in header files. For example,
  1) The printf() is a standard library function to send formatted output to the screen (display output on the screen). This function is defined in the stdio.h header file. Hence, to use the printf() function, we need to include the stdio.h header file using #include<stdio.h>

  2) The sqrt() function calculates the square root of a number. The function defined in the math.h header file.

8.3.1 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 function 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.

8.3.2 Example: Square root using sqrt() function

Our task to find the sqare root of a number.

To compute the square root of a number, you can use the sqrt() library function. This function is defined in the math.h header file.

```c
#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;
}
```

When you run the program, the output will be:

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

8.4 User-defined Function

- Function which are defined by the user at the time of writing program are known as User-defined functions or “tailor-made functions”.
- User-defined functions can be improved and modified according to the need of the programmer.
- Whenever we write a function that is case-specific and is not defined in any header file, we need to declare and define our own functions according to the syntax.

8.4.1 Advantages of User-Defined functions:

- Code of these functions in reusable in other programs.
- The program will be easier to understand, maintain and debug.
- A large program can be divided among many programmers.

8.4.2 Example: user-defined function

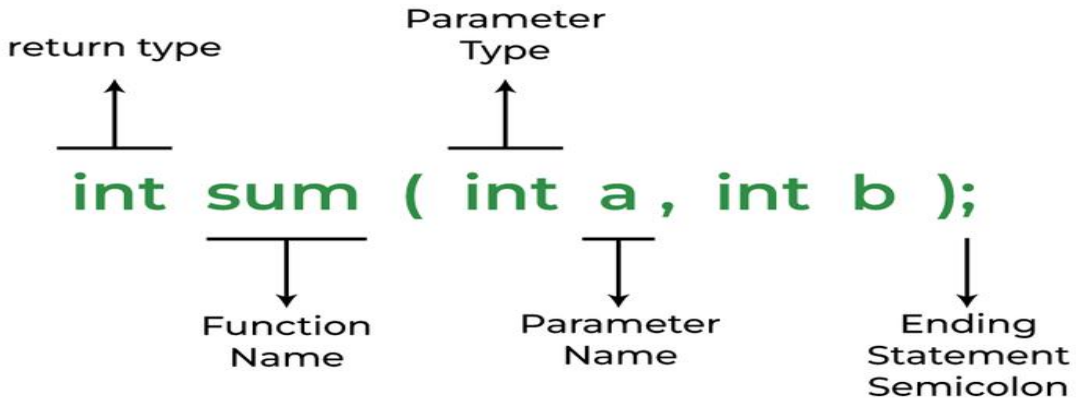
Here is an example of add two integer. To perform this task,we have created an user-defined addNumbers().

```c
#include <stdio.h>
int addNumbers(int a, int b);          // function prototype

int main()
{
    int n1,n2,sum;

    printf("Enters two numbers: ");
    scanf("%d %d",&n1,&n2);

    sum = addNumbers(n1, n2);         // function call
    printf("sum = %d",sum);

    return 0;
}

int addNumbers(int a, int b)          // function definition
{
    int result;
    result = a+b;
    return result;                   // return statement
}
```

8.5. Function Prototype

- A function prototype is simply the declaration of a function that specifies function's name, parameters and return type.
- It doesn't contain function body.

- A function prototype gives information to the compiler that the function may later be used in the program.
- Function prototype tell the compiler how many parameters a function takes, what kind of parameters it returns, and what types of data it takes.
- Function declarations do not need to include parameter names, but definitions must.



Function Arguments and Return Values:

A function is all dependent upon what value it will return and what arguments are been passed.

Syntax:

```
return_type name_of_the_function (parameter_1, parameter_2);
```

Function Prototype or function declaration consists of 4 parts.

- Return type
- Function name
- Parameter list
- Terminating semicolon

1. **Function Return Type:**

Function return type tells what type of value is returned after all function is executed.

Example:

```
int func(parameter_1,parameter_2);
```

It will return an integer value after returning statements inside the function.

2. **Function Arguments**
   Function Arguments or Function Parameters are the data that is passed to a function so that it operates over that data inside a function.

Example:

```
int function_name(int var1, int var2);
```

Note:

The parameter name is not mandatory while declaring functions. We can also declare the above function without using the name of the data variables.

Ex.

```
int sum(int, int);
```

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

1. Name of the function is addNumbers()
2. Return type of functions is int
3. Two arguments of type int are passed to the function

Note: the function prototype is not needed if the user-defined function is defined before the main() method.

8.6. Function Definition

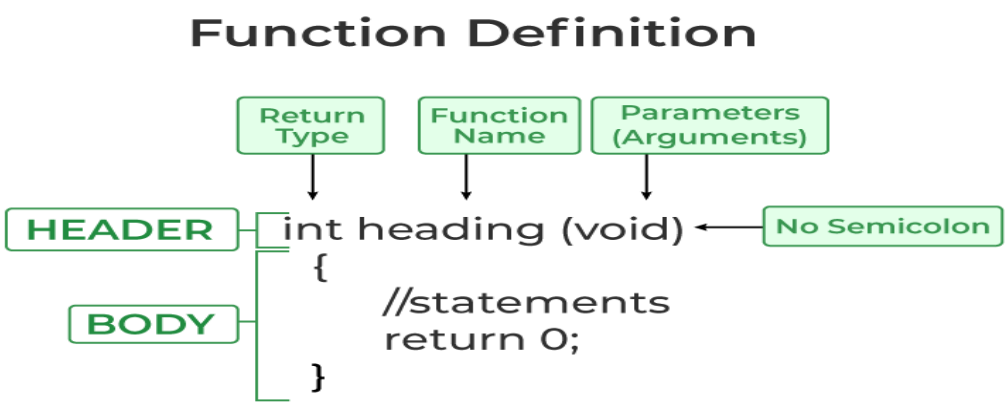
- A Functions definition consists function header and a function body.
- Function definition contains the block of code to perform a specific task.
- When a function is called, the control of the program is transferred to the function definition. And, the compiler starts executing the codes inside the body of a function.

```
return_type function_name (parameters)
{
    //body of the function
}
```

Return Type: The function always starts with a return type of the function. But if there is no return value then the void keyword is used as the return type of the function. The function calculate a value, and return it back to the main() program. This value is said to be the return value. We need to declare what type of variable will be returned from the function. This can be int, double, char, string, and void.

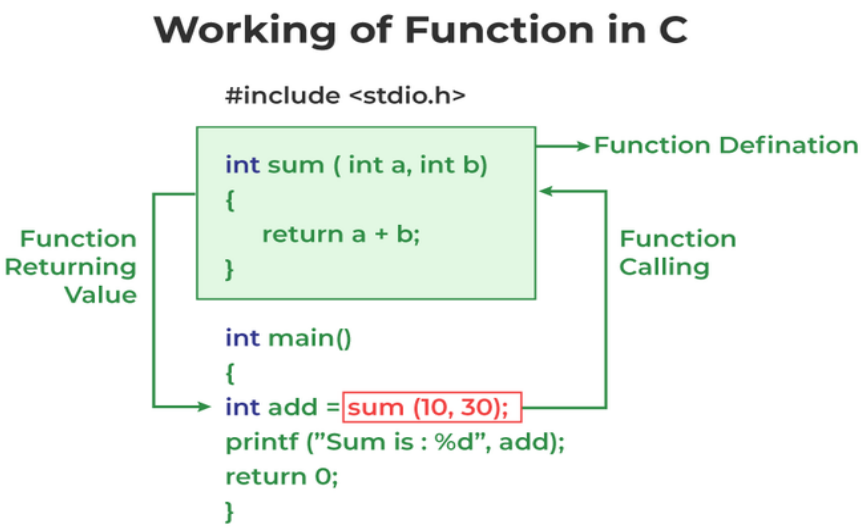
Function Name: Just like variables, every function has Name. Name of the function which should be unique.

Parameters: Values that are passed during the function call. Sometimes the function need specific inputs to execute the code inside. These inputs are passed to the function to be used in the form of function parameters. A function can have more than one parameter.



8.7. Function Call

- When a function is called, control of the program gets transferred to the function.
- To call a function, parameters are passed along the function name.
- In the below example, the first sum sum function is called and 10,30 are passed to the sum function.
- After the function call sum of a and b is returned and control is also returned back to the main function of the program.



Ex.

```c
// C program to show function
// call and definition
#include <stdio.h>

// Function that takes two parameters
// a and b as inputs and returns
// their sum
int sum(int a, int b)
{
  return a + b;
}

// Driver code
int main()
{
  // Calling sum function and
  // storing its value in add variable
  int add = sum(10, 30);

  printf("Sum is: %d", add);
  return 0;
}
```

Output

```
Sum is: 40
```

8.8. How the C function Works?

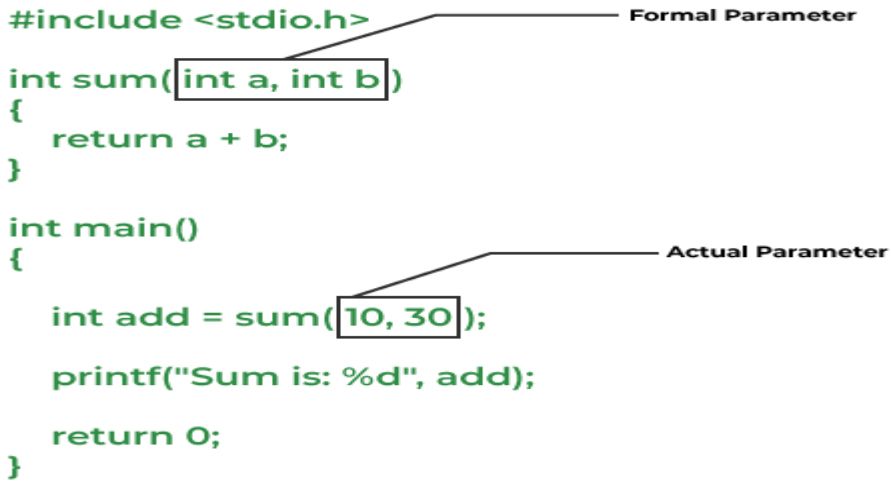
Working of the c function can be broken into the following steps as mentioned below:

1. **Declaring a Function:**
   Declaring a function is a step where we declare a function. Here we define the return types and parameters of the function.
2. **Calling a Function:**
   Calling the function is a step where we call the function by passing the arguments in the function.
3. **Executing a Function**
   Executing the function is a step where we can run all the statements inside the function to get the final result.
4. **Returning a value**
   Returning a value is the step where the calculated value after the execution of the function is returned.
5. **Exiting the function**
   Exiting the function is the final step where all the allocated memory to the variables, functions, etc is destroyed before giving full control to the main function.

8.9. Passing Parameters to Functions

- The value of the function passed when the function is being invoked (function calling) is known as the Actual Parameter. In the below program, 10 and 30 are known as actual Parameters.

- Formal Parameters are the variable and the data type as mentioned in the function declaration. In the below program, a and b are known as formal parameters.




```c
#include <stdio.h>

int sum(int a, int b)
{
    return a + b;
}

int main()
{
    int add = sum(10, 30);

    printf("Sum is: %d", add);

    return 0;
}
```

8.9.1 Pass by value:

Parameter passing in this method copies values from actual parameters into formal function parameters. As a result, any changes made inside the functions do not reflect in the caller's parameters.

Example:

```c
// C program to show use
// of call by value
#include <stdio.h>

void swap(int var1, int var2)
{
  int temp = var1;
  var1 = var2;
  var2 = temp;
}

// Driver code
int main()
{
  int var1 = 3, var2 = 2;
  printf("Before swap Value of var1 and var2 is: %d, %d\n",
         var1, var2);
  swap(var1, var2);
  printf("After swap Value of var1 and var2 is: %d, %d",
         var1, var2);
  return 0;
}
```

Output

```
Before swap Value of var1 and var2 is: 3, 2
After swap Value of var1 and var2 is: 3, 2
```

8.9.2 Pass by Reference

The caller's Actual parameters and the function's actual parameter refers to the same location. So any changes made inside the function are reflected in the caller's actual parameters.

```c
// C program to show use of
// call by Reference
#include <stdio.h>

void swap(int *var1, int *var2)
{
  int temp = *var1;
  *var1 = *var2;
  *var2 = temp;
}

// Driver code
int main()
{
  int var1 = 3, var2 = 2;
  printf("Before swap Value of var1 and var2 is: %d, %d\n",
         var1, var2);
  swap(&var1, &var2);
  printf("After swap Value of var1 and var2 is: %d, %d",
         var1, var2);
  return 0;
}
```

Output

```
Before swap Value of var1 and var2 is: 3, 2
After swap Value of var1 and var2 is: 2, 3
```

8.10. Types of User-defined Function 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.

8.10.1 Function with No Arguments and No return Value or Without Arguments and without return value

In this function we don't have to give any value to the function (argument/parameters) and even don't have to take any value from it in return.

Return_type function_name(void); // Function Prototype

Ex.

```c
void checkPrimeNumber();

int main() {
  checkPrimeNumber();    // argument is not passed
  return 0;
}

// return type is void meaning doesn't return any value
void checkPrimeNumber() {
  int n, i, flag = 0;

  printf("Enter a positive integer: ");
  scanf("%d",&n);

  // 0 and 1 are not prime numbers
  if (n == 0 || n == 1)
    flag = 1;

  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 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(); 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.

8.10.2. Function with No arguments Passed but Returns a Value or Without Arguments and With Return Value

In these types of function, we don't have to give any value or argument to the function but in return, we get something from it i.e., some value.

```c
#include <stdio.h>
int getInteger();

int main() {

  int n, i, flag = 0;

  // no argument is passed
  n = getInteger();

  // 0 and 1 are not prime numbers
  if (n == 0 || n == 1)
    flag = 1;

  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;
}
```

```c
// returns integer entered by the user
int getInteger() {
  int n;

  printf("Enter a positive integer: ");
  scanf("%d",&n);

  return n;
}
```

The empty parentheses in the 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.

8.10.3. Function with Arguments and No return value

In these types of functions, we give arguments or parameters to function but we don't get any value from it in return.

```c
#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;
}
```

```
// return type is void meaning doesn't return any value
void checkPrimeAndDisplay(int n) {
  int i, flag = 0;

  // 0 and 1 are not prime numbers
  if (n == 0 || n == 1)
    flag = 1;

  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 the checkPrimeAndDisplay() function.

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

8.10.4. Function with arguments and a return value

In these functions, we give arguments to a function and in return, we also get some value from it.

```
#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 returned value is assigned to the 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;
}
```

```
// int is returned from the function
int checkPrimeNumber(int n) {

  // 0 and 1 are not prime numbers
  if (n == 0 || n == 1)
    return 1;

  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 the 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 the flag variable.

Depending on whether flag is 0 or 1, an appropriate message is printed from the main() function.

8.10.5 Which approach is better?

Well, it depends on the problem you are trying to solve. In this case, passing an argument and returning a value from the function (example 4) 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.

8.11. Function With arrays

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). In C Programming, a single array element or an entire array can be passed to a function. This can be done for both one-dimensional array and a multi-dimensional array.

8.11.1 Passing One-Dimensional Array in function:

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

Ex. 1: Pass Individual array Elements:

```c
#include <stdio.h>
void display(int age1, int age2) {
  printf("%d\n", age1);
  printf("%d\n", age2);
}

int main() {
  int ageArray[] = {2, 8, 4, 12};

  // pass second and third elements to display()
  display(ageArray[1], ageArray[2]);
  return 0;
}
```

Output

```
8
4
```

Here, we have passed array parameters to the display() function in the same way we pass variables to a function.

```
// pass second and third elements to display()
display(ageArray[1], ageArray[2]);
```

We can see this in the function definition, where the function parameters are individual variables:

```
void display(int age1, int age2) {
  // code
}
```

Example 2: Pass Arrays to Functions

First way- The receiving parameter of the array may itself be declared as an array, as shown below

```
Syntax:-

return_type function(type arrayname[SIZE])
```

```
// Program To find the array sum using function

#include<stdio.h>
int add(int array[5]){                  //Declaration with size
   int sum =0;
   for(int i=0;i<5;i++){
       sum += array[i];
   }
   return sum;
}
int main(){
   int arr[5] = {2, 3, 4, 5, 6};
   printf("Array sum is %d\n", add(arr)); // For passing array, only its name is
       passed as argument
   return 0;
}
```

OUTPUT

```
Array sum is 20
```

Here, as you see, in the above example, even though the parameter array is declared as an int array of 5 elements, the compiler automatically converts it to an int pointer like this int *array. This is necessary because no parameter can actually receive an entire array. A pointer to an array gets passed when an array is passed to the function; thus, a pointer parameter can only receive it.

Second Way- The receiving parameters may be declared as an unsized array, as shown below:

```
Syntax:-

return_type function(type arrayname[ ]);
```

```c
// Program to find the minimum element

#include<stdio.h>

int findMin(int arr[] , int size){ // Receiving array base address and size
   int min = arr[0];
   for(int i =1; i<size;i++){
       if(min > arr[i]){
           min = arr[i];
       }
   }
   return min;
}
int main(){
   int arr[5] = {76 , 89 , 67 , 23 , 24};
   printf("The minimum element is %d\n ",findMin(arr , 5)); // Passing array with
       size
   return 0;
}
```

OUTPUT

```
The minimum element is  23
```

Since the compiler converts an array declaration(informal parameters of a function) into an array pointer, the actual size of the array is irrelevant to the parameter.

Third Way- The receiving parameters can be declared as a pointer, as shown below:

```
Syntax:-

return_type function(type *arrayname) {}
```

```c
//Program to reverse the array using function

#include <stdio.h>
void reverseArray(int *arr, int start, int end) ;
void printArray(int *arr, int size);

int main()
{
   int arr[] = {1, 2, 3, 4, 5, 6};

   int n = sizeof(arr) / sizeof(arr[0]); // calculating size of the array

   printArray(arr, n); // To print original array

   reverseArray(arr, 0, n-1); // Calling the function with array name, starting
       point and ending point

   printf("Reversed array is\n");

   printArray(arr, n);      // To print the Reversed array

   return 0;
}
```

```c
void reverseArray(int *arr, int start, int end) //Receiving parameter declared as
    pointer
{
   while (start < end)
   {
       int temp = arr[start];
       arr[start] = arr[end];
       arr[end] = temp;
       start++;
       end--;
   }
}
void printArray(int *arr, int size)
{
  for (int i = 0; i < size; i++)
      printf("%d ",arr[i]);

  printf("\n");

}
```

OUTPUT

```
1 2 3 4 5 6
Reversed array is
6 5 4 3 2 1
```

This is allowed because a pointer receiving an array can be used as an array. The critical point is that arrays and pointers are very closely linked. However, A pointer to an array gets passed

when they are two different things and are generally not equivalent. The only exception is for function arguments, but this is only because function arguments can never be arrays- they are always converted to pointers.

8.11.2 Pass Multidimensional Arrays to a function

- For Passing multidimensional arrays to a function we need to pass the name of the array similar to a one-dimensional array.
- When a function argument is an array of more than one dimension, we must declare the size of the dimensions. However, the size of the first dimension is optional.
- For passing two-dimensional arrays, it is not mandatory to specify the number of rows in the array. However number of columns should always be specified.
- One important thing for passing multidimensional arrays is, first array dimension does not have to be specified. The second (and any subsequent) dimensions must be given.

1) When both dimensions are available globally (either as a macro or as a global constant).

```c
#include <stdio.h>
const int M = 3;
const int N = 3;

void print(int arr[M][N])
{
    int i, j;
    for (i = 0; i < M; i++)
      for (j = 0; j < N; j++)
        printf("%d ", arr[i][j]);
}

int main()
{
    int arr[][3] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
    print(arr);
    return 0;
}
```

```
Output

  1 2 3 4 5 6 7 8 9
```

2) When only second dimension is available globally (either as a macro or as a global constant).

```c
#include <stdio.h>
const int N = 3;

void print(int arr[][N], int m)
{
    int i, j;
    for (i = 0; i < m; i++)
      for (j = 0; j < N; j++)
        printf("%d ", arr[i][j]);
}

int main()
{
    int arr[][3] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
    print(arr, 3);
    return 0;
}
```

```
OUTPUT
 1 2 3 4 5 6 7 8 9
```

The above method is fine if second dimension is fixed and is not user specified. The following methods handle cases when second dimension can also change.

3) If compiler is C99 compatible

From C99, C language supports variable sized arrays to be passed simply by specifying the variable dimensions.

```c
// The following program works only if your compiler is C99 compatible.
#include <stdio.h>

// n must be passed before the 2D array
void print(int m, int n, int arr[][n])
{
    int i, j;
    for (i = 0; i < m; i++)
      for (j = 0; j < n; j++)
        printf("%d ", arr[i][j]);
}

int main()
{
    int arr[][3] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
    int m = 3, n = 3;
    print(m, n, arr);
    return 0;
}
```

```
Output

  1 2 3 4 5 6 7 8 9
```

If compiler is not C99 compatible, then we can use one of the following methods to pass a variable sized 2D array.

4) Using a single pointer

In this method, we must typecast the 2d array when passing to function

```c
#include <stdio.h>
void print(int *arr, int m, int n)
{
    int i, j;
    for (i = 0; i < m; i++)
      for (j = 0; j < n; j++)
        printf("%d ", *((arr+i*n) + j));
}

int main()
{
    int arr[][3] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
    int m = 3, n = 3;

    // We can also use "print(&arr[0][0], m, n);"
    print((int *)arr, m, n);
    return 0;
}
```

```
Output

  1 2 3 4 5 6 7 8 9
```

5) Using the concept of pointer to an array

```c
#include <stdio.h>
const int M = 3;


void print(int (*arr)[M])
{
    int i, j;
    for (i = 0; i < M; i++)
    for (j = 0; j < M; j++)
        printf("%d ", arr[i][j]);
}

int main()
{
    int arr[][3] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
    print(arr);
    return 0;
}
```

```
Output

1 2 3 4 5 6 7 8 9
```

8.11.3 Points to Remember:

- Passing arrays to functions in C/C++ are passed by reference. Even though we do not create a reference variable, the compiler passes the pointer to the array, making the original array available for the called function's use. Thus, if the function modifies the array, it will be reflected back to the original array.
- The equivalence between arrays and pointers to an array is valid only and only for the function arguments.
- If an argument is a multidimensional array, its size must be specified. However, the size of the first dimension is optional.

8.12 Function with String

We Know that a String is a sequence of characters enclosed in double quotes.
For example, "Hello World" is a string and it consists of a sequence of English letters in both uppercase and lowercase and the two words are separated by a white space. So, there are total 11 characters.
We know that a string in C programming language ends with a NULL \0 character. So, in order to save the above string we will need an array of size 12. The first 11 places will be used to store the words and the space and the 12th place will be used to hold the NULL character to mark the end.

8.12.1 Passing one Dimensional String to a function

Function declaration to accept one dimensional string
We know that strings are saved in arrays so, to pass an one dimensional array to a function we will have the following declaration.

```
Syntax:
returnType functionName(char str[]);

Example:
void displayString(char str[]);
```

Ex.

```
#include <stdio.h>

void displayString(char []);

int main(void) {
  // variables
  char message[] = "Hello World";

  // print the string message
  displayString(message);

  return 0;
}

void displayString(char str[]) {
  printf("String: %s\n", str);
}
```

```
String: Hello World
```

Another way we can print the string is by using a loop like for or while and print characters till we hit the NULL character.

In the following example we have redefined the displayString function logic.

```
#include <stdio.h>
void displayString(char []);

int main(void) {
  // char array
  char message[] = "Hello World";
  // print the string message
  displayString(message);

  return 0;
}

void displayString(char str[]) {
  int i = 0;
  printf("String: ");
  while (str[i] != '\0') {
    printf("%c", str[i]);
    i++;
  }
  printf("\n");
}
```

```
String: Hello World
```

8.12.2 Passing Two-Dimensional String to a function

In order to accept two-dimensional string array, the function declaration will look like the following.

```
Syntax:
returnType functionName(char [][C], type rows);

Example:
void displayCities(char str[][50], int rows);
```

In the above example we have a function by the name displayCities and it takes a two dimensional string array of type char.

The str is a two dimensional array as because we are using two [][] sqaure brackets. It is important to specify the second dimension of the array and in this example the second dimension i.e., total number of columns is 50.

The second parameter rows tell us about the total number of rows in the give two dimensional string array str.

The return type for this function is set to void that means it will return no value.

Ex.

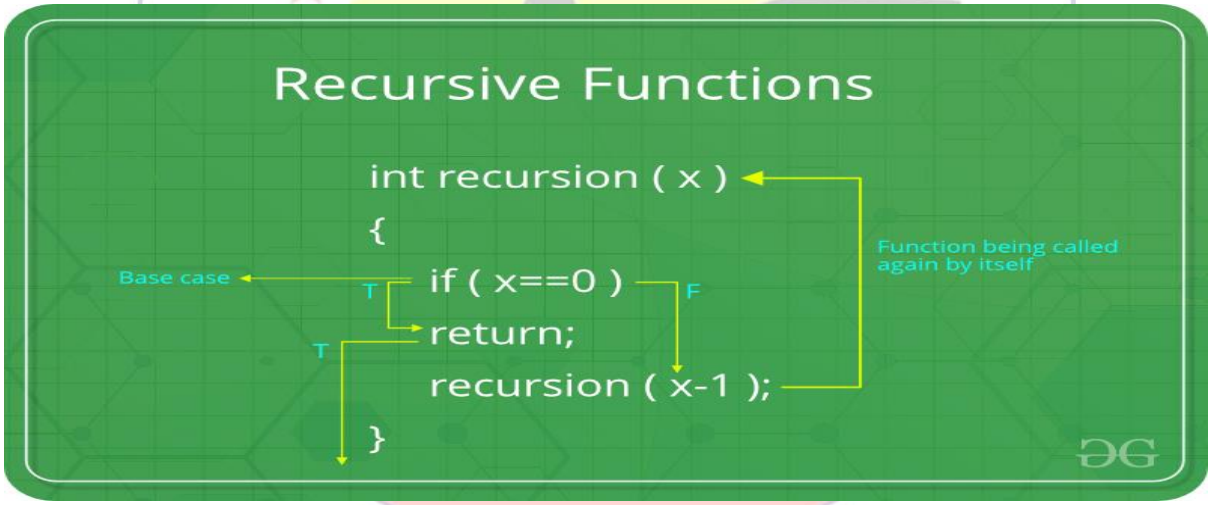
```c
#include <stdio.h>
void displayCities(char [][50], int rows);

int main(void) {
  // char array
  char cities[][50] = {"Bangalore","Chennai","Kolkata",
      "Mumbai","New Delhi" };

  int rows = 5;
  // print the name of the cities
  displayCities(cities, rows);
  return 0;
}

void displayCities(char str[][50], int rows) {
  // variables
  int r, i;
  printf("Cities:\n");
  for (r = 0; r < rows; r++) {
    i = 0;
    while(str[r][i] != '\0') {
      printf("%c", str[r][i]);
      i++;
    }
    printf("\n");
  }
}
```

Output:

```
Cities:
Bangalore
Chennai
Kolkata
Mumbai
New Delhi
```

8.13 Recursive Functions:

- A function that calls itself is known as a recursive function. And, this technique is known as Recursion.
- Any function which calls Itself is called a recursive function.
- This make the life of programmer easy by dividing a complex problem into simple or easier problems.
- Any function which can be solved recursively can also be solved iteratively.
- We must have certain condition in the function to break out of the recursion, otherwise recursion will occur infinite times.
- Recursion are used to solve tougher problems like tower of Hanoi, Fibonacci Series, Factorial finding, etc., and many more, where solving by intuition become tough.



8.13.1 What is the base Condition?

The case in which the function doesn't recur is called the base case.

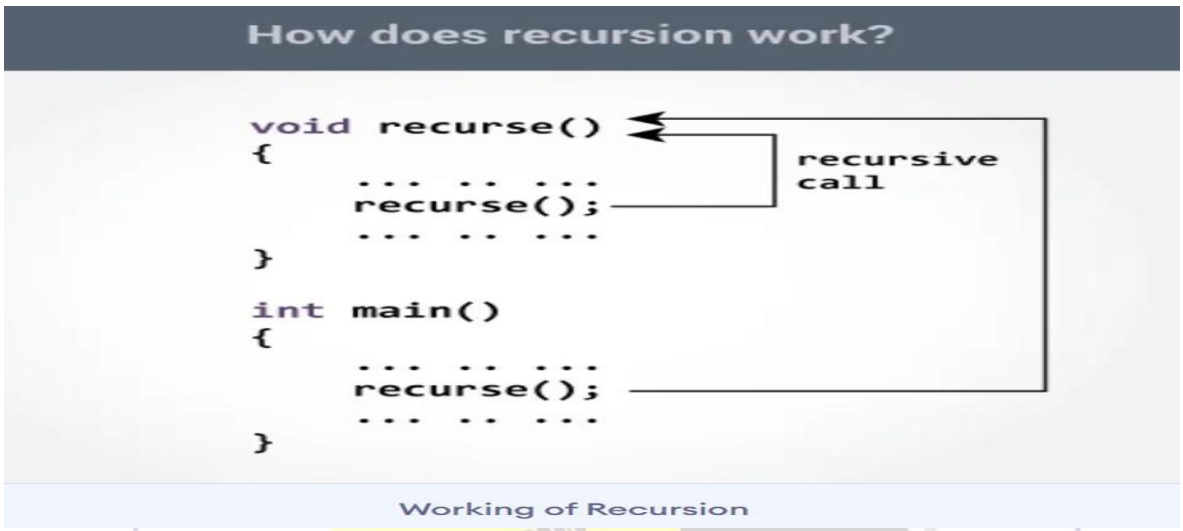
For ex., when be try to find the factorial of a number using recursion, the case when our number becomes smaller than 1 is base case.

8.13.2 Recursive Case:

The instance where the function keep calling itself to perform subtask which is generally the same problem with the problem size reduced to many smaller parts, is called the recursive case.

8.13.3 How Recursion Works?

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

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



Working of Recursion

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

To prevent infinite recursion, if...else statement (or similar approach) can be used where one branch makes the recursive call, and other doesn't.

8.13.4 Example

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 n) {
     if (n != 0)
          // sum() function calls itself
          return n + sum(n-1);
     else
          return n;
}
```

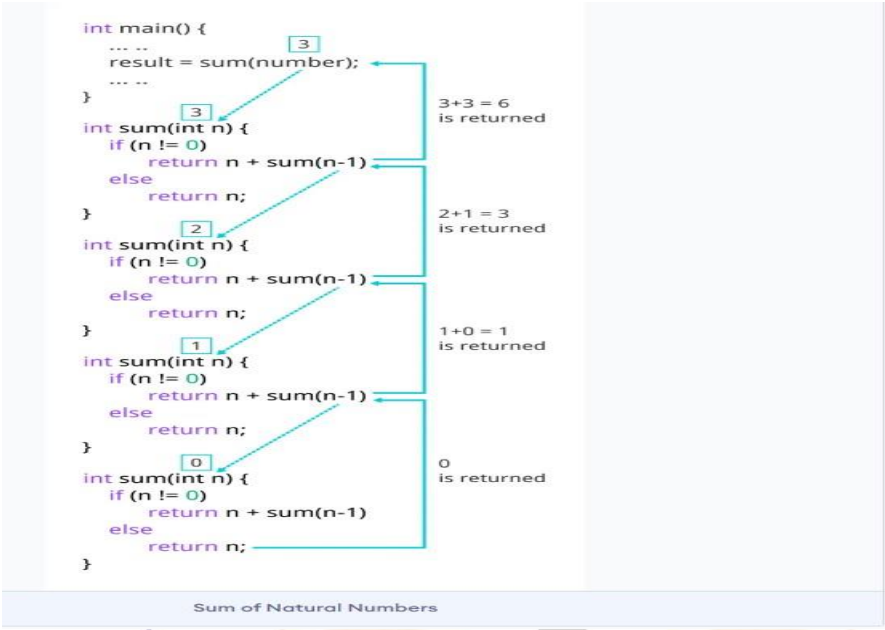
Output

```
Enter a positive integer:3
sum = 6
```

Initially, the sum() is called from the main() function with number passed as an argument.

Suppose, the value of n inside sum() is 3 initially. During the next function call, 2 is passed to the sum() function. This process continues until n is equal to 0.

When n is equal to 0, the if condition fails and the else part is executed returning the sum of integers ultimately to the main() function.



Sum of Natural Numbers

Ex. 2 Recursion is used to calculate the factorial of a number.

```c
#include <stdio.h>

int factorial(int num)
{
    if (num > 0)
    {
        return num * factorial(num - 1);
    }
    else
    {
        return 0;
    }
}

int main()
{
    int ans = factorial(5);
    printf("%d", ans);
    return 0;
}
```

We can understand the above program of the recursive method call by the figure given below:

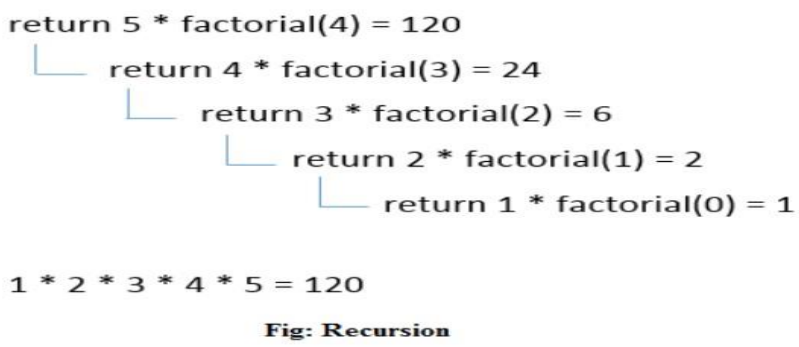


Fig: Recursion

Output: 120

8.13.5 Memory Allocation Of Recursive Method:

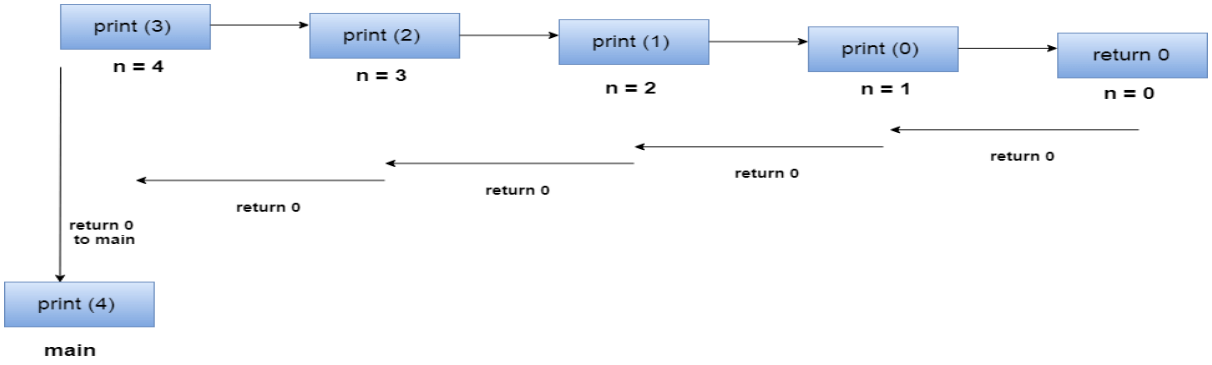
- Each recursive call creates a new copy of that method in the memory.
- Once some data is returned by the method, the copy is removed from the memory. Since all the variables and other stuff declared inside function get stored in the stack, therefore a separate stack is maintained at each recursive call.
- Once the value is returned from the corresponding function, the stack gets destroyed.
- Recursion involves so much complexity in resolving and tracking the values at each recursive call. Therefore we need to maintain the stack and track the values of the variables defined in the stack.

Let us consider the following example to understand the memory allocation of the recursive functions.

```
int display (int n)
{
    if(n == 0)
        return 0; // terminating condition
    else
    {
        printf("%d",n);
        return display(n-1); // recursive call
    }
}
```

Explanation:

- Let us examine this recursive function for n = 4.
- First, all the stacks are maintained which prints the corresponding value of n until n becomes 0, Once the termination condition is reached, the stacks get destroyed one by one by returning 0 to its calling stack.
- Consider the following image for more information regarding the stack trace for the recursive functions.



Stack tracing for recursive function call

8.13.6 Advantages and Disadvantages of Recursion

- Recursion makes program elegant. However, if performance is vital, use loops instead as recursion is usually much slower.
- That being said, recursion is an important concept. It is frequently used in data structure and algorithms. For example, it is common to use recursion in problems such as tree traversal.

8.14 Mostly asked Questions:

1. **Define functions:**

Functions are the block of code that is executed every time called during an execution of a program.

2.**What is the difference between function argument and parameters?**

Function Parameters are the types of values declared in a function declaration. Whereas, Function Arguments are the values that are passed in the function during the function calling.

Example:

```
int func(int x,int y);
func(10,20);
```

Here, int x and int y are Parameters in C while, 10 and 20 are the arguments passed to the function.

3. **Can We return multiple values from a C Function?**
   No, we can't return multiple values from a function in C.
4. **What is the actual and formal parameter?**

   Formal parameter: The variables declared in the function prototype is known as Formal arguments or parameters.

   Actual parameter: The values that are passed in the function are known as actual arguments or arguments.

5. **How to declare a function in C?**
   A function in C can be declared using function_name, parameters, and return type. We can use syntax to declare it as follows

```
return_type func_name(parameter_1,parameter_2....)
```

6. **What is the difference between function declaration and definition?**
   The data like function name, return type, and the parameter is included in the function declaration whereas the definition is the body of the function. All these data are shared with the compiler according to their corresponding steps.

7. **What is the difference between arrays and pointers?**
   An array is a collection of variables belonging to the corresponding data type. It carries the same size. In contrast, a Pointer is a single variable that stores the address of another variable.
8. **Why do you need to send the size of an array to a function?**
   Usually, the name of the array "decays" to a pointer to its first element. That means you no longer know the size of that array, as you can only pass around a pointer to an element in that array. So you have to pass its size, so the function receiving the "array" knows how long it is.

8.15 Bonus (write a function and use in another program)

1) Step 1- make a function which you want to use it in another program.

2) Step 2- Save it in .h extension e.g., mathCalculation.h

```
mathCalculation.h
```

3)Step 3- include this function where you want to use.

```
#include "mathsOperations.h"
```

4)Step 4- Call it.

Ex. **add.h**

add.h | mathsOperations.h | main.c

```
1  //add() function defined in add.h file
2  int add(int a,int b)
3  {
4      return a+b;
5  }
```

mathOperation.h

add.h mathsOperations.h main.c

```c
//here some basic calculations are :

int addTwoNumber(int a,int b)
{
    return a+b;
}

int substractThreeNumber(int a,int b,int c)
{
    return a-b-c;
}

int multiplicationOfThreeNumbers(int a,int b,int c)
{
    return a*b*c;
}
```

Main.c

add.h mathsOperations.h main.c

```c
//now i want to use add and multiply function which is defined in another files
//for this we need to include in our main program which is given below
#include "add.h"
#include "multiply.h"
#include "mathsOperations.h"

int add(int ,int );
int multiply(int ,int );
int divisionOfTwoNumbers(int ,int );
int substractThreeNumber(int ,int ,int );

int main()
{
    printf("sum is %d ",add(9,12));
    printf("\nmultiplication of 9 and 12 is %d",multiply(9,12));
    printf("\n division of 108 and 12 is %d",divisionOfTwoNumbers(108,12));
    printf("\nsubstraction of 19 ,1 and 12 is %d",substractThreeNumber(19,1,12));
}
```

*****

Designed By-

Rahul Gupta (MCA From BIT Mesra)

All India Rank -79 (NIMCET 2019 MCA Entrance EXAM)