Computer for Chemists (M.Sc. II Sem)

UNIT III

Structured Programming

Structured programming (sometimes known as *modular programming*) is a programming paradigm that facilitates the creation of programs with readable code and reusable components. All modern programming languages support structured programming, but the mechanisms of support, like the syntax of the programming languages, varies.

Structured programming encourages dividing an application program into a hierarchy of modules or autonomous elements, which may, in turn, contain other such elements. Within each element, code may be further structured using blocks of related logic designed to improve readability and maintainability.

Modular programming, which is today seen as synonymous with structured programming, emerged a decade later as it became clear that reuse of common code could improve developer productivity. In modular programming, a program is divided into semi-independent modules, each of which is called when needed. C is called structured programming language because a program in c language can be divided into small logical functional modules or structures with the help of function procedure.

Advantages of structured programming

The primary advantages of structured programming are:

1. It encourages top-down implementation, which improves both readability and maintainability of code.
2. It promotes code reuse, since even internal modules can be extracted and made independent, residents in libraries, described in directories and referenced by many other applications.
3. It's widely agreed that development time and code quality are improved through structured programming.
4. It is user friendly and easy to understand.
5. Similar to English vocabulary of words and symbols.
6. It is easier to learn.
7. They require less time to write.
8. They are easier to maintain.
9. These are mainly problem oriented rather than machine based.
10. Program written in a higher level language can be translated into many machine languages and therefore can run on any computer for which there exists an appropriate translator.

11. It is independent of machine on which it is used i.e. programs developed in high level languages can be run on any computer.

Disadvantages of structured programming

The following are the disadvantages of structured programming:

1. A high level language has to be translated into the machine language by translator and thus a price in computer time is paid.
2. The object code generated by a translator might be inefficient compared to an equivalent assembly language program.
3. Data type are proceeds in many functions in a structured program. When changes occur in those data types, the corresponding change must be made to every location that acts on those data types within the program. This is really a very time consuming task if the program is very large.
4. Let us consider the case of software development in which several programmers work as a team on an application. In a structured program, each programmer is assigned to build a specific set of functions and data types. Since different programmers handle separate functions that have mutually shared data type. Other programmers in the team must reflect the changes in data types done by the programmer in data type handled. Otherwise, it requires rewriting several functions.

FUNCTION:

A function is a group of statements that together perform a task. Every C program has at least one function, which is 'main' function, and all the most trivial programs can define additional functions. We can divide up your code into separate functions.

A function can be called multiple times to provide reusability and modularity to the C program. In other words, we can say that the collection of functions creates a program. The function is also known as *procedure* or *subroutine* in other programming languages.

Advantages of functions in C/ Why functions?

There are the following advantages of C functions.

- By using functions, we can avoid rewriting same logic/code again and again in a program.
- We can call C functions any number of times in a program and from any place in a program.
- We can track a large C program easily when it is divided into multiple functions.
- Reusability is the main achievement of C functions.
- However, Function calling is always a overhead in a C program.

Function Aspects

There are three aspects of a C function.

- **Function declaration** A function must be declared globally in a c program to tell the compiler about the function name, function parameters, and return type.
- **Function call** Function can be called from anywhere in the program. The parameter list must not differ in function calling and function declaration. We must pass the same number of functions as it is declared in the function declaration.
- **Function definition** It contains the actual statements which are to be executed. It is the most important aspect to which the control comes when the function is called. Here, we must notice that only one value can be returned from the function.

S.No.	**C function aspects**	**Syntax**
1	Function declaration	return_type function_name (argument list);
2	Function call	function_name (argument_list)
3	Function definition	return_type function_name (argument list) {function body;}

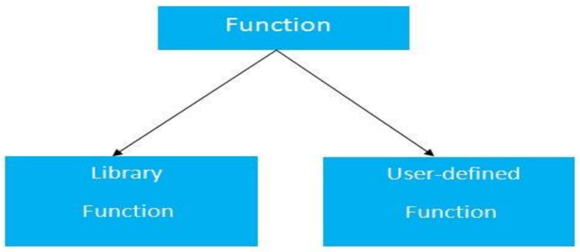
The syntax of creating function in c language is given below:

1. return_type function_name(data_type parameter...){
2. //code to be executed
3. }

Types of Functions

There are two types of functions in C programming:

1. **Library Functions**: are the functions which are declared in the C header files such as scanf(), printf(), gets(), puts(), ceil(), floor() etc.
2. **User-defined functions**: are the functions which are created by the C programmer, so that he/she can use it many times. It reduces the complexity of a big program and optimizes the code.



Example:1

Following is the source code for a function called max. This function takes two parameters num1and num2 and returns the maximum between the two:

```
/* function returning the max between two numbers */

int max(int num1,int num2)

int result;

if(num1 > num2)

result = num1;

else

result = num2;

return result;

}
```

For the above defined function max, following is the function declaration:

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

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

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

Example:2

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

```
#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
}
```

Calling a function

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.

Control of the program is transferred to the user-defined function by calling it.

Syntax of function call

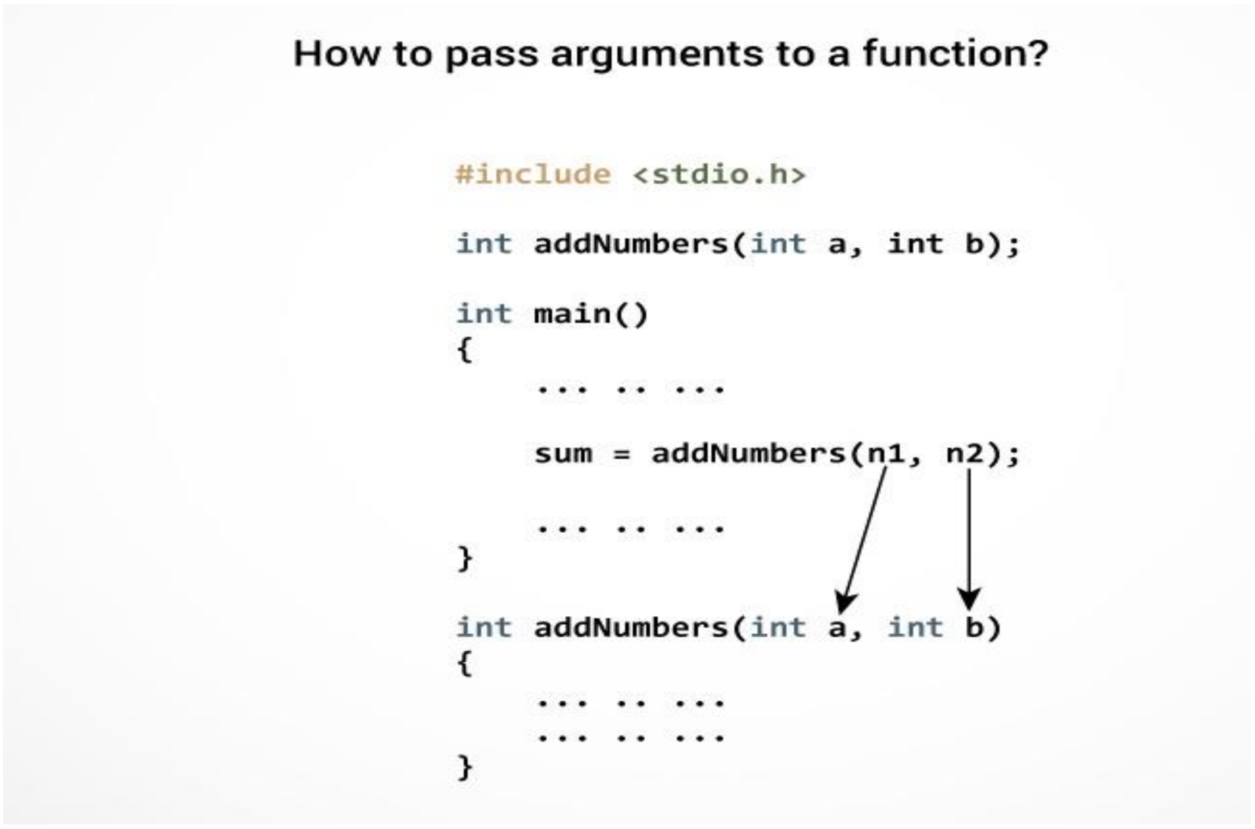
functionName(argument1, argument2, ...);

In the above example, the function call is made using addNumbers(n1, n2); statement inside the main() function.

Passing arguments to a function

In programming, argument refers to the variable passed to the function. In the above example, two variables *n1* and *n2* are passed during the function call.

The parameters *a* and *b* accepts the passed arguments in the function definition. These arguments are called formal parameters of the function.


How to pass arguments to a function?

```
#include <stdio.h>

int addNumbers(int a, int b);

int main()
{
    ... .. ...

    sum = addNumbers(n1, n2);

    ... .. ...
}

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

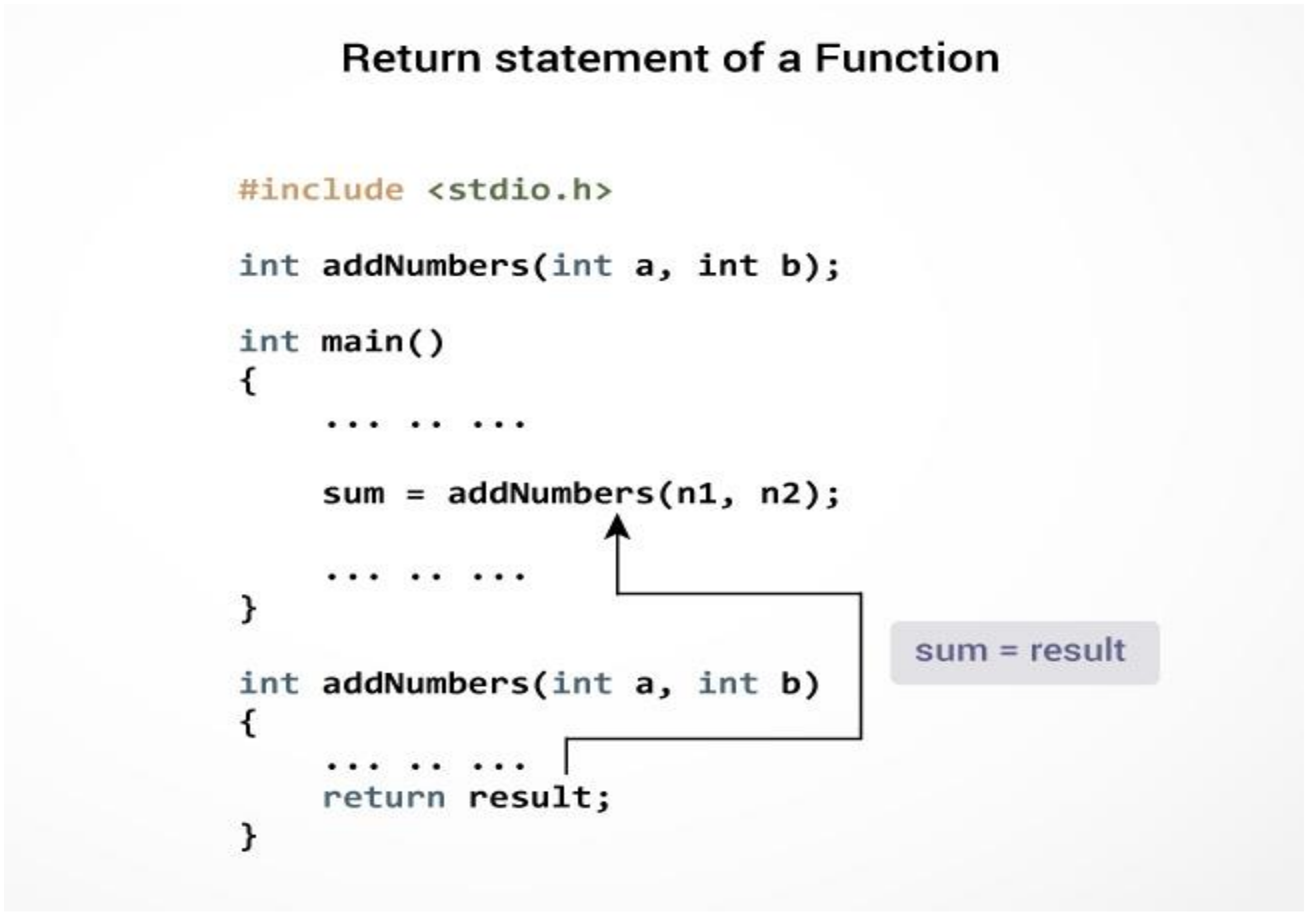
The type of arguments passed to a function and the formal parameters must match, otherwise, the compiler will throw an error.

A function can also be called without passing an argument.

<u>Return Statement</u>

The return statement terminates the execution of a function and returns a value to the calling function. The program control is transferred to the calling function after the return statement.

In the above example, the value of the *result* variable is returned to the main function. The *sum* variable in the `main()` function is assigned this value.


Return statement of a Function

```
#include <stdio.h>

int addNumbers(int a, int b);

int main()
{
    ... .. ...

    sum = addNumbers(n1, n2);

    ... .. ...
}

int addNumbers(int a, int b)
{
    ... .. ...
    return result;
}
```

Syntax of return statement

```
return (expression);
```

For example,

```
return a;
return (a+b);
```

Call by Value and Call by Reference

Functions can be invoked in two ways: **Call by Value** or **Call by Reference**. These two ways are generally differentiated by the type of values passed to them as parameters.

The parameters passed to function are called *actual parameters* whereas the parameters received by function are called *formal parameters*.

Call By Value in C: In this parameter passing method, values of actual parameters are copied to function's formal parameters and the two types of parameters are stored in different memory locations. So any changes made inside functions are not reflected in actual parameters of caller.

In other words, in this parameter passing method, values of actual parameters are copied to function's formal parameters, and the parameters are stored in different memory locations. So any changes made inside functions are not reflected in actual parameters of the caller.

Call by Value Example: Swapping the values of the two variables

```
#include <stdio.h>
void swap(int , int); //prototype of the function
int main()
{
int a = 10;
int b = 20;
printf("Before swapping the values in main a = %d, b = %d\n",a,b);
 swap(a,b);
printf("After swapping values in main a = %d, b = %d\n",a,b);  }
void swap (int a, int b)
{
int temp;
temp = a;
a=b;
b=temp;
printf("After swapping values in function a = %d, b = %d\n",a,b);
 }
```

Output

```
Before swapping the values in main a = 10, b = 20
After swapping values in function a = 20, b = 10
After swapping values in main a = 10, b = 20
```

Call by reference in C

Call by reference method copies the address of an argument into the formal parameter. In this method, the address is used to access the actual argument used in the function call. It means that changes made in the parameter alter the passing argument.

In this method, the memory allocation is the same as the actual parameters. All the operations in the function are performed on the value stored at the address of the actual parameter, and the modified value will be stored at the same address. Means, both the actual and formal parameters refer to same locations, so any changes made inside the function are actually reflected in actual parameters of caller.

Call by reference Example: Swapping the values of the two variables

```
#include <stdio.h>
void swap(int *, int *); //prototype of the function
int main()
{
   int a = 10;
   int b = 20;
   printf("Before swapping the values in main a = %d, b = %d\n",a,b);
   swap(&a,&b);
   printf("After swapping values in main a = %d, b = %d\n",a,b);
 }
void swap (int *a, int *b)
{
   int temp;
   temp = *a;
   *a=*b;
   *b=temp;
   printf("After swapping values in function a = %d, b = %d\n",*a,*b);
 }
```

Output

```
Before swapping the values in main a = 10, b = 20
After swapping values in function a = 20, b = 10
After swapping values in main a = 20, b = 10
```

Call By Value	**Call By Reference**
While calling a function, we pass values of variables to it. Such functions are known as "Call By Values".	While calling a function, instead of passing the values of variables, we pass address of variables(location of variables) to the function known as "Call By References.
In this method, the value of each variable in calling function is copied into corresponding dummy variables of the called function.	In this method, the address of actual variables in the calling function are copied into the dummy variables of the called function.
With this method, the changes made to the dummy variables in the called function have no effect on the values of actual variables in the calling function.	With this method, using addresses we would have an access to the actual variables and hence we would be able to manipulate them.
A copy of the value is passed into the function	An address of value is passed into the function
Actual and formal arguments are created at the different memory location	Actual and formal arguments are created at the same memory location

<u>**Recursion and Iteration**</u>

Recursion and Iteration are both used for executing a set of statements repeatedly, until the condition becomes false. A program is called recursive when an entity calls itself. A program is call iterative when there is a loop (or repetition).

Recursion

Recursion is applied to functions, where the function calls itself to repeat the lines of code/ certain statements.

Iteration

Iteration, on the other hand, uses looping in order to execute a set of statements multiple times. A given problem can be solved both by recursion as well as iteration, however, they have certain important differences as well.

Difference between recursion and iteration

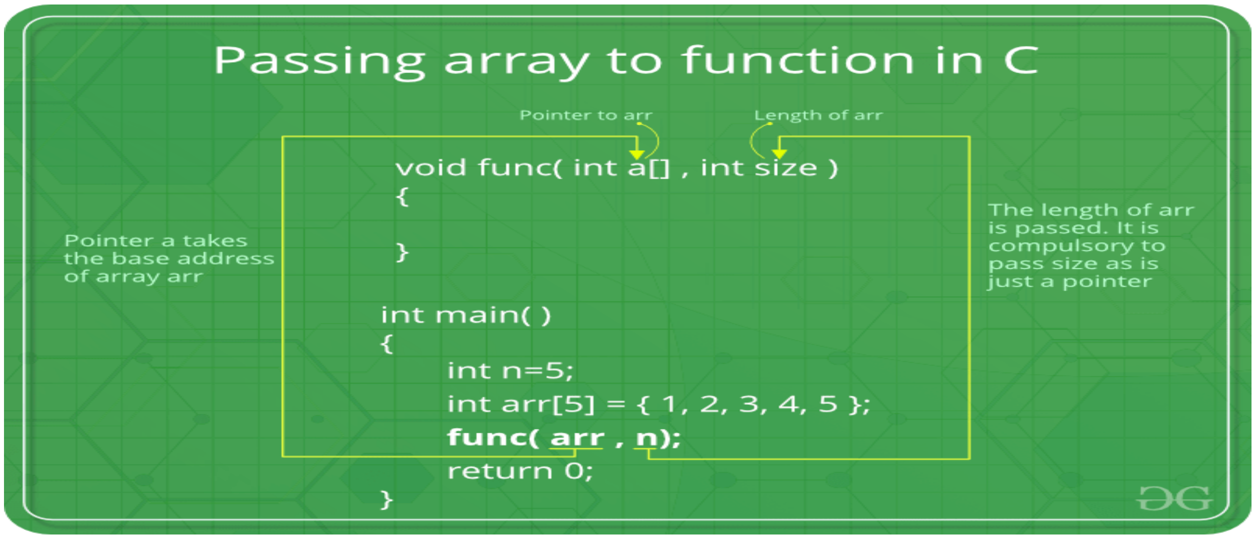
Parameter	**Recursion**	**Iteration**
Definition	Recursion involves a recursive function which calls itself repeatedly until a base condition is not reached.	Iteration involves the usage of loops through which a set of statements are executed repeatedly until the condition is not false.
Termination condition	Here termination condition is a base case defined within the recursive function.	Termination condition is the condition specified in the definition of the loop.
Infinite Case	If base case is never reached it leads to infinite recursion leading to memory crash.	If condition is never false, it leads to infinite iteration with computers CPU cycle being used repeatedly.
Memory Usage	Recursion uses stack area to store the current state of the function,due to which memory usage is high.	Iteration uses the permanent storage area only for the variables involved in its code block, hence memory usage is less.
Code Size	Code size is comparitively smaller.	Code size is comparitively larger.
Performance	Since stack are is used to store and restore the state of recursive function after every function call , performance is comparitively slow.	Since iteration does not have to keep re-initializing its component variables and neither has to store function states, the performance is fast.
Memory Runout	There is a possibility of running out of memory, since for each function call stack	There is no possibility of running out of

	area gets used.	memory as stack area is not used.
Overhead	Recursive functions involve extensive overhead, as for each function call the current state, parameters etc have to be pushed and popped out from stack.	There is no overhead in Iteration.
Applications	Factorial , Fibonacci Series etc.	Finding average of a data series, creating multiplication table etc.
Example	#include <stdio.h> int fact(int n){ if(n == 0) return 1; else return n * factorial(n-1); } int main() { printf("Factorial for 5 is %d", fact(5)); return 0; } Output: Factorial for 5 is 120	#include <stdio.h> int main() { int i, n = 5, fact = 1; for(i = 1; i <= n; ++i) fact = fact * i;printf("Factorial for 5 is %d", fact); return 0; } Output: Factorial for 5 is 120

Passing Arrays to Functions:

In C programming, you can pass en entire array to functions. Before we learn that, let's see how you can pass individual elements of an array to functions. Passing array elements to a function is similar to passing variables to a function.

Following is a simple example to show how arrays are typically passed in C:


Passing array to function in C
Pointer to arr
Length of arr
Pointer a takes the base address of array arr
The length of arr is passed. It is compulsory to pass size as is just a pointer

```
void func( int a[] , int size )
{

}

int main( )
{
        int n=5;
        int arr[5] = { 1, 2, 3, 4, 5 };
        func( arr , n);
        return 0;
}
```

Example 1: Passing an array

```
#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};

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

Output

```
8
4
```

Example 2: Passing arrays to functions

```
// Program to calculate the sum of array elements by passing to a function

#include <stdio.h>
float calculateSum(float age[]);

int main() {
    float result, age[] = {23.4, 55, 22.6, 3, 40.5, 18};

    // age array is passed to calculateSum()
    result = calculateSum(age);
    printf("Result = %.2f", result);
    return 0;
}
```

Output

```
Result = 162.50

float calculateSum(float age[]) {

  float sum = 0.0;

  for (int i = 0; i < 6; ++i) {
               sum += age[i];
  }

  return sum;
}
```

How to pass function pointer as parameter

In C programming you can only pass variables as parameter to function. You cannot pass function to another function as parameter. But, you can pass function reference to another function using function pointers. Using function pointer you can store reference of a function and can pass it to another function as normal pointer variable. And finally in the function you can call function pointer as normal functions.

```
 /*C program to pass function pointer as parameter to another function
 */

#include <stdio.h>


/* Function declarations */
void greetMorning();
void greeEvening();
void greetNight();

void greet(void (*greeter)());


int main()
{

    greet(greetMorning);

        greet(greeEvening);

        greet(greetNight);

    return 0;
}
```

```
void greet(void (*greeter)())
{

    greeter();
}



void greetMorning()
{
    printf("Good, morning!\n");
}


void greeEvening()
{
    printf("Good, evening!\n");
}


void greetNight()
{
    printf("Good, night!\n");
}
```

Output:

```
Good, morning!
Good, evening!
Good, night!
```