

Unit - IV: Function and Dynamic Memory Allocation

Functions: Designing structured programs, Declaring a function, Signature of a Function, Parameters and return type of a function, passing parameters to functions, call by value, passing arrays to functions, passing pointers to functions, idea of call by reference, Some C standard functions and libraries

Recursion: Simple programs, such as Finding Factorial, Fibonacci series etc.
Limitations of Recursive functions

Dynamic memory allocation: Allocating and freeing memory, Allocating memory for arrays of different data types

PART-1

FUNCTIONS

Outline

- Designing structured programs
- Declaring a function
- Parameters and return type of a function
- Passing parameters to functions
 1. Call by value,
 2. Call by reference,
- Passing arrays to functions
- Passing pointers to functions
- Recursion
- Dynamic memory allocation

Designing Structured Programs

The programs that we study or read from books are very small, when compared to the programs that we write for a complex problem. If the program is very large,

There are too many disadvantages,

- It is very difficult for the programmer to write a large program.
- Very difficult to identify the logical errors and correct.
- Very difficult to read and understand.
- Large programs are more prone of errors and so on.

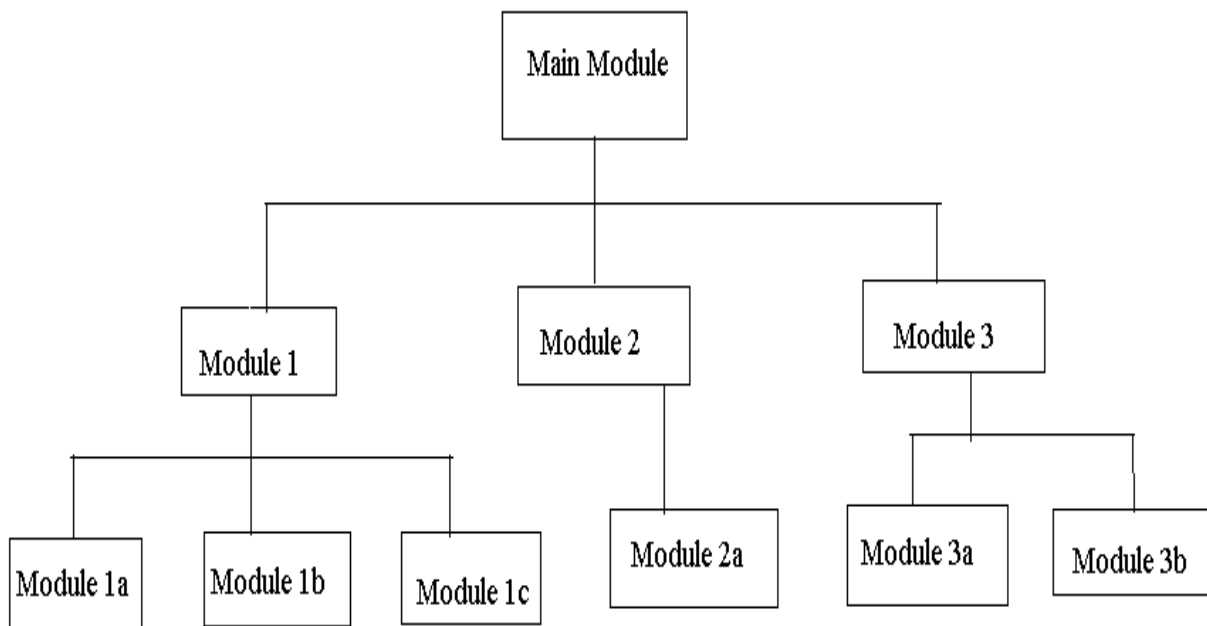
These disadvantages can be overcome using functions.

Breaking a complex problem into smaller parts (or) modules (using functions) is a common practice for larger programs.

The planning for large programs consists of

- First understanding the problem as a whole,
- Second breaking it into simpler,
- Understandable parts.

We call each of these parts of a program a module and the process of subdividing a problem into manageable parts top-down design.



In C language, a large program can be divided into a series of individual related programs called Modules. These modules are called Functions.(It is also called as subprogram.)

Definition of Function

A function is a self contained program segment that carries out some specific and well defined tasks.

- Using Functions, large programs can be reduced to smaller ones. It is easy to debug and find out the errors in it. It is also increases the readability of the program.

Types of Functions

There are two types of function in C

1. Library Functions(Build –in- functions or Standard library functions)
2. User Defined Functions(UDF)

1. Standard library functions(Library Functions)

- The standard library functions are built-in functions in C programming.
- These functions are defined in header files.
- **These functions which are part of the C Compiler that have been written for general purpose are called library functions. They are also called build-in-functions.**

Example:

- The function sqrt() is used to find square root of a given number
- The function scanf() is used to read data from the keyboard

Some of the Header files are:

<ctype.h>	character testing and conversion functions. Ex: toupper() ,tolower()
<math.h>	Mathematical functions Sqrt() :returns the square root of x
<stdio.h>	standard I/O library functions Printf(): Formatted printing to stdout Scanf(): Formatted input from stdin
<stdlib.h>	Utility functions such as string conversion routines memory allocation routines , random number generator , etc. Free(): Release allocated memory Malloc(): Allocate memory
<string.h>	string Manipulation functions Strcpy(): Copies a string Strlen(): Calculates the length of a string

Advantages

1. The programmer's job is made easier because the functions are already available.
2. The library functions can be used whenever required.
3. The functions are portable
4. It saves considerable development time

Disadvantages

1. Since the standard library functions are limited, programmer cannot completely rely on these library functions.
2. The programmer cannot write his/her own function in programs.

This disadvantage of standard library functions can be overcome using user defined functions.

Example:

Square root using sqrt() function

```
#include <stdio.h>
#include <math.h> // Build -in- functions
Void main()
{
    float n, root;
    printf("\n Enter a number: ");
    scanf("%f", &n);
    // Computes the square root of num and stores in root.
    root = sqrt(n);

    printf("\n Square root of %.2f = %.2f", n, root);

}
```

Output:

Enter a number: 9

Square root of 9 = 3.00

1. User Defined Functions (UDFs)

- In most cases, the programmers need other than library functions to achieve some specific tasks.
- Since the standard library functions are limited, programmer cannot completely rely on these library functions. So, user defined functions are necessary.

These functions which are written by the programmer/user to do some specific tasks are called User Defined Functions (UDFs).

Advantage of user defined functions in C

1. **Reusability and reduction of code size** : The existing functions can be reused as building blocks to create new programs. This results in reduced program size. These functions can be used any number of times.
2. **Readability of the program can be increased**: Programs can be written easily and we can keep track of what each function is doing.
3. **Modular Programming Approach**: A large program is divided into smaller sub programs so that each sub program performs a specific task. This approach makes the program development more manageable.
4. **Easier Debugging**: Using modular approach, localizing, locating and isolating a faulty function is much easier.
5. **Function Sharing**: A Function can be shared by many programmers.

The user defined functions have three elements.


1. Function declaration
2. Function call
3. Function definition

User-defined function

Syntax:

```
#include <stdio.h>
void functionName(); // Function declaration
void main()
{
    ... ..
    ... ..
    functionName(); // Function call
    ... ..
    ... ..
}

void functionName() // Function definition
{
    ... ..
    ... ..
}
```



Write c program addition of two numbers. (Without Functions)

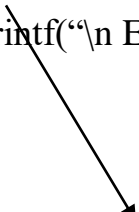
```
#include<stdio.h>
void main()
{
    int a,b,sum=0;
    printf("\n Enter any two integers:");
    scanf("%d%d",&a,&b);
    sum=a+b;
    printf("\n The sum = %d\n ",sum);
}
```

Output:

```
Enter any two integers:
2
4
The sum = 6
```

Write c program addition of two number using function

```
#include<stdio.h>
void add( ); // Function declaration
void main()
{
printf(“\n main function”);
add(); // Function call
printf(“\n End of Program\n”);
}
void add( ) // Function definition
{
int a,b,sum=0;
printf(“Enter any two integers:”);
scanf(“%d%d”,&a,&b);
sum=a+b;
printf(“The sum = %d”,sum);
}
```



Output:

Main function

Enter any two integers:

2

4

The sum = 6

End of Program

Function Declaration or prototype:

- A function must be declared globally in a c program
- To tell the compiler about the function name, function parameters, and return type.
- Before using a function, declare it first

Syntax:

return_type function_name();

Where:

- return type: any data type
- Function name: user define function

Ex: 1. void add();

2. int sum();

Note: function declaration must be before the main ()

Syntax: 2

return type function name(parameter list/ argument list);

Where: parameter list is data type with variable

EX: 1. int add(int a,int b);

2. float avg(float a, float b, float c...so on)

Function call:

- Function can be called from anywhere in the main 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.

Syntax:

1. Function_name();

Ex: add(); sub(); printline();

2. Function_name(variables list);

3. Ex: add(a,b); sub(a,b,c);

Function definition:

Function definition should have two parts

1. Function header
 - Type
 - Name
 - Parameters(arguments)

2. Function body
 - Declaration statements
 - Executable statements
 - Return statements

Syntax of function definition

```
return_value_type function _name (parameter _list)
{
    // body of the function
}
```

Example:

```
Void add() // function header
{
int a=10;b=20,sum=0; // function body
Sum=a+b;
}
```

Note: functions have three elements; among the elements **function declaration** should be optional when the **function definition** write before the main function.

Write c program addition of two number using function

```
#include<stdio.h>
void add( ) // Function definition
{
int a,b,sum=0;
printf(“Enter any two integers:”);
scanf(“%d%d”,&a,&b);
sum=a+b;
```

```
printf("The sum = %d",sum);
}
void main()
{
add(); // Function call
}
```

Output:

Enter any two integers: 2 4

The sum = 6

Write c program addition of two number using function

```
#include<stdio.h>
void add( ); // Function declaration
```

```
void main()
{
add(); // Function call
}
```

```
void add( ) // Function definition
{
int a,b,sum=0;
printf("Enter any two integers:");
scanf("%d%d",&a,&b);
sum=a+b;
printf("The sum = %d",sum);
}
```

Output:

Enter any two integers: 2 4

The sum is 6

Parameters (argument)

Two type of parameters

1. Actual Parameters (argument)
2. Formal Parameters (argument)

Formal and Actual Parameters (argument)

Actual parameter

- The actual parameter are those variables at main function
- Actual parameters are passing the values

EX: add(a,b)

Here a and b are actual parameters

Formal parameter

- The Formal parameter are those variables at function definition header / function declaration
- Formal parameters are catch the vale from main function

Ex: void add (int x, int y)

Here x and y are formal parameters

Note: actual parameters and formal parameters need not be same but return type must be same

- **Main function is called as calling function**
- **Sub function is called as called function**

Difference Formal and Actual Parameters (argument)

Actual Parameters	Formal Parameters
<p>1. Actual parameters are used in calling function when a function is invoked.</p> <p>E.g. <code>c=sum(a,b);</code> here, a and b are actual parameters</p>	<p>1. Formal parameters are used in the function header of a called function.</p> <p>E.g. <code>int sum(int m, int n)</code> here, m and n are formal parameters</p>
<p>2. Actual parameters can be constant, variables or expression.</p> <p>E.g. <code>c=sum(a+4,b);</code></p>	<p>2. Formal parameters should be only variables. Expressions and constants are not allowed.</p> <p>e.g. <code>int sum(int m, int n) // correct</code> <code>int sum(int m+n, int n) //Wrong</code> <code>int sum(int m, 10) //wrong</code></p>
<p>3. Actual parameters send values to the formal parameters.</p> <p>E.g. <code>c=sum(4,5);</code></p>	<p>3. Formal parameters receive values from the actual parameters.</p> <p>E.g. <code>int sum(int m, int n)</code> Here, M will have the value 4 and n will have the value 5.</p>
<p>4. Addresses of actual parameters can be sent to formal parameters.</p>	<p>4. If formal parameters contains addresses, they should be declared as pointers.</p>

How to pass arguments to a function?

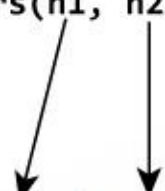
```
#include <stdio.h>

int addNumbers(int a, int b);

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

    sum = addNumbers(n1, n2);
    ... .. ...
}

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

The diagram consists of two arrows. The first arrow starts at the variable 'n1' in the function call 'sum = addNumbers(n1, n2);' and points down to the parameter 'a' in the function definition 'int addNumbers(int a, int b)'. The second arrow starts at the variable 'n2' in the function call and points down to the parameter 'b' in the function definition.

Where: n1, n2 are actual parameters

a,b are formal parameters

Return statement of a Function

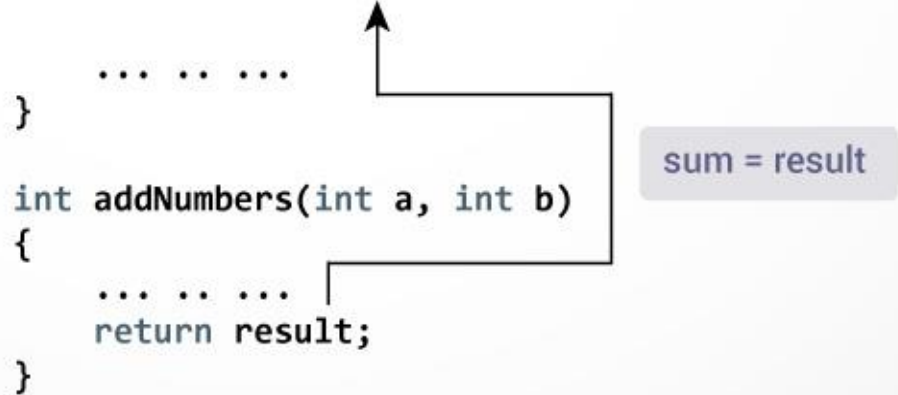
```
#include <stdio.h>

int addNumbers(int a, int b);

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

    sum = addNumbers(n1, n2);
    ... .. ...
}

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



sum = result

Categories of Functions

Function in C may define with or without parameters, and a function may or may not return a value.

1. **Functions with no parameters and no return types. Ex: void abc();**
2. **Functions with no parameters and return types. Ex: int abc();**
3. **Functions with parameters and no return types. Ex: void abc(int a);**
4. **Functions with parameters and return types. Ex: int abc(int a);**

NOTE:

- From the above, 1 and 3 types does not return any value when we call the function. So, we use void return type while defining the function.
- When we call the function 2 and 4 types, they will return some value. So, we have to use the appropriate data type (int, float, double, etc.), as a return type while defining the function.

1. Function with No argument and No Return value

- In this method, we won't pass any arguments to main function to sub function.
- Sub function not return any value to main function

General syntax:

```
void function_name()
```

Ex: void add()

No return type

No parameter

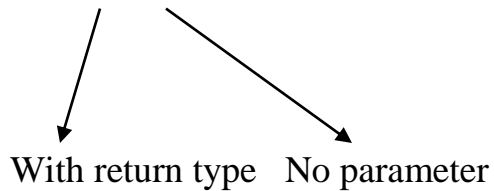
2. Function with no argument and with Return value

- In this method, we won't pass any arguments to main function to sub function.
- Sub function return value to main function

General syntax:

```
datatype function_name()
```

Ex: int add()



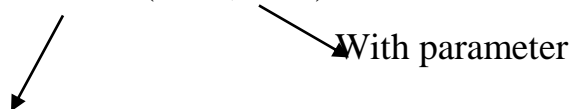
3. Function with argument and No Return value

- In this method, we pass any arguments to main function to sub function.
- Sub function not return any value to main function

General syntax:

```
Void function_name(arguments list)
```

Ex: void add(int a, int b)



No return type

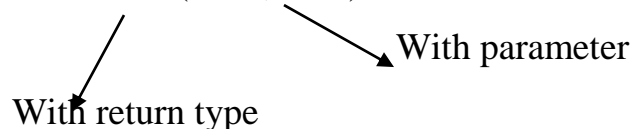
4. Function with argument and Return value

- In this method, we pass any arguments to main function to sub function.
- Sub function not return any value to main function

General syntax:

```
datatype function_name(arguments list)
```

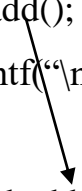
Ex: int add(int a, int b)



PROGRAMS

1. Functions with no parameters and no return types

```
#include<stdio.h>
void add();
void main()
{
    printf("\nGoing to calculate the sum of two numbers:\n");
    add();
    printf("\n END OF PROGRAM");
}
void add()
{
    int a,b,sum=0;
    printf("\nEnter two numbers");
    scanf("%d %d",&a,&b);
    sum=a+b;
    printf("\n The sum is: %d",sum);
}
```



Output:

Going to calculate the sum of two numbers:

Enter two numbers

10

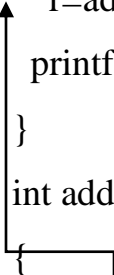
14

The sum is : 24

END OF PROGRAM

2. Functions with no parameters and return types

```
#include<stdio.h>
int add();
void main()
{
int r=0
    printf("\nGoing to calculate the sum of two numbers:\n");
    r=add();
    printf("\n The sum is: %d",r);
}
int add()
{
    int a,b,sum=0;
    printf("\nEnter two numbers");
    scanf("%d %d",&a,&b);
    sum=a+b;
    return sum;
}
```



Output:

Going to calculate the sum of two numbers:

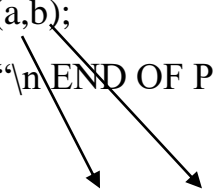
Enter two numbers 10

14

The sum is : 24

3. Functions with parameters and no return types

```
#include<stdio.h>
void add(int x, int y);
void main()
{
    int a,b;
    printf("\nGoing to calculate the sum of two numbers:");
    printf("\nEnter two numbers:");
    scanf("%d %d",&a,&b);
    add(a,b);
    printf("\nEND OF PROGRAM");
}
void add(int x, int y)
{
    int sum=0;
    sum=x+y;
    printf("\nThe sum is: %d",sum);
}
```



Output:

Going to calculate the sum of two numbers:

Enter two numbers 10

14

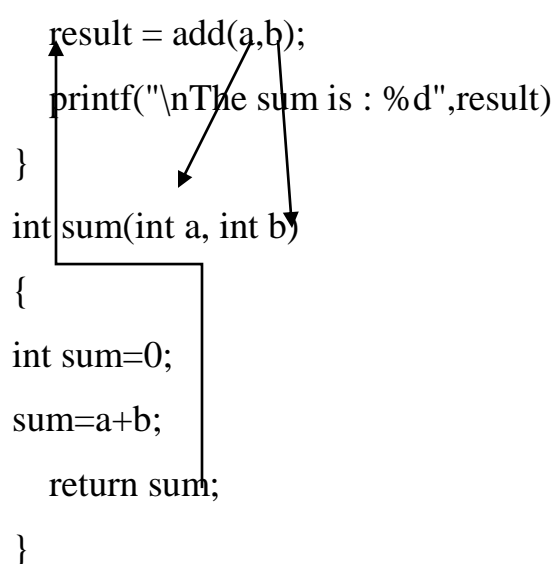
The sum is :24

END OF PROGRAM

4. Functions with parameters and return types

```
#include<stdio.h>
int sum(int a, int b);
void main()
{
    int a,b,result;
    printf("\nGoing to calculate the sum of two numbers:");
    printf("\nEnter two numbers:");
    scanf("%d %d",&a,&b);

    result = add(a,b);
    printf("\nThe sum is : %d",result);
}
int sum(int a, int b)
{
    int sum=0;
    sum=a+b;
    return sum;
}
```



Output:

Going to calculate the sum of two numbers:

Enter two numbers 10

14

The sum is :24

Passing parameters to functions

In C Programming Language, there are two methods to pass parameters from calling function to called function and they are as follows...

1. Call by Value
2. Call by Reference

1. Call by Value (Pass by Value)

Defination:

In **call by value** parameter passing method, the copy of actual parameter values are copied to formal parameters and these formal parameters are used in called function. The changes made on the formal parameters does not effect the values of actual parameters.

- That means, after the execution control comes back to the calling function, the actual parameter values remains same

For example :

```
#include<stdio.h>

void swap(int x,int y) ; // function declaration

void main()
{
    int a=10, b=20 ;
    printf("\nBefore swap: a = %d, b= %d", a, b) ;
    swap(a, b) ; // calling function
    printf("\nAfter swap: a = %d\ b = %d", a, b);
}

void swap(int x, int y) // called function
{
    int t ;
    t= x ;
```

```
x = y ;  
y= t ;  
printf("\n In swap: x = %d y = %d", x, y);  
}
```

Output:

Before swap: a = 10, b= 20

In swap: x =20 y = 10

After swap: a = 10 b = 20

Call by Reference(Pass by Reference)

- In Call by Reference parameter passing method, the memory location address of the actual parameters is copied to formal parameters.
- This address is used to access the memory locations of the actual parameters in called function.
- In this method of parameter passing, the formal parameters must be pointer variables.

Call by reference parameter passing method:

Definition:

The address of the actual parameters is passed to the called function and is received by the formal parameters (pointers). Whenever we use these formal parameters in called function, they directly access the memory locations of actual parameters. So the changes made on the formal parameters effects the values of actual parameters.

For example :

```
#include<stdio.h>

void swap(int *x,int *y) ; // function declaration

void main()
{
    int a=10, b=20 ;
    printf("\nBefore swap: a = %d, b= %d", a, b) ;
    swap(&a, &b) ; // calling function
    printf("\nAfter swap: a = %d\ b = %d", a, b);
}

void swap(int *x, int *y) // called function
{
    int t ;                //here x=&a, y=&b
    t= *x ;
    *x = *y ;
    *y= t ;
    printf("\n In swap: x = %d y = %d", *x, *y);
}
```

Output:

Before swap: a = 10, b= 20

In swap: x =20 y = 10

After swap: a = 20 b = 10

Differences between Pass by value and pass by Reference

Pass by value	Pass by Reference
1. When a function is called the value of variables are passed	1. When a function is called the addresses of variables are passed.
2. Change of formal parameters in the function will not affect the actual parameters in the calling function.	2. The actual parameters are changed since the formal parameters indirectly manipulate the actual parameters.
3. Execution is slower since all the values have to be copied into formal parameters.	3. Execution is faster since only addresses are copied.
4. Information transfer through parameters is not possible.	4. Information transfer through parameters is possible. That is more than one value we can transfer through the parameters. Note: Only one value can be sent using return statement.

PASS BY VALUE
VERSUS
PASS BY REFERENCE

PASS BY VALUE	PASS BY REFERENCE
Mechanism of copying the function parameter value to another variable	Mechanism of passing the actual parameters to the function
Changes made inside the function are not reflected in the original value	Changes made inside the function are reflected in the original value
Makes a copy of the actual parameter	Address of the actual parameter passes to the function
Function gets a copy of the actual content	Function accesses the original variable's content
Requires more memory	Requires less memory
Requires more time as it involves copying values	Requires a less amount of time as there is no copying
	Visit www.PEDIAA.com

Passing arrays to functions

- Like the values of variables, arrays can be passed to a function.
- As we know that the array name contains the address of the first element.
- Here, we must notice that **we need to pass only the name of the array** in the function which is intended to accept an array.

Methods to declare arrays to functions

There are 3 ways to declare the function

1. First way:

Syntax: return_type function(type arrayname[SIZE])

Ex: void sort(int max[20]);
int sum(int marks[20]);

2. Second way:

Syntax: return_type function(type *arrayname)

Ex: void sort(int *max);
int sum(int *marks);

The concept of a pointer. (Store the address of first element)

3. Third way:

Syntax: return_type function(type arrayname[])


- Here array size is not declared

Ex: void sort(int max[]);
int sum(int marks[]);

Example: C program to pass an array to the function that contains marks obtained by the student. Then display the total marks obtained by the student

```
#include <stdio.h>
int total_marks(int a[10]); //array to function
void main()
{
    int m[10]={40,80,75,90,88};
        int t= 0;
            t=total_marks(m); // function call(array to function)
printf("\n Total marks = %d",t);
}

int total_marks(int a[10])
{
    int sum=0,i;
    for (i=0;i<5;i++)
        sum=sum+a[i];
    return sum;
}
```



Output

Total marks = 373

Example 2: Write C program using function to sort the array

```
#include<stdio.h>

void sort_elements (int a[20]); //array to function

void main ()
{
    int arr[10] = { 10, 9, 7, 101, 23, 44, 12, 78, 34, 23};
    sort_elements (arr); // array elements are pass to function
}

void sort_elements (int a[20]) //array to function
{
    int i, j,temp;
    for(i = 0; i<10; i++)
    {
        for(j = i+1; j<10; j++)
        {
            if(a[i] > a[j])
            {
                temp = a[i];
                a[i] = a[j];
                a[j] = temp;
            }
        }
    }

    printf("\n Printing Sorted Element List ...\n");
    for(i = 0; i<10; i++)
    {
        printf("%d\n",a[i]);
    }
}
```

}

Output:

Printing Sorted Element List:

7

9

10

12

23

23

34

44

78

101

Passing pointers to functions

- Pointer as a function parameter is used to hold addresses of parameters/arguments passed during function call. This is known as call by reference.
- When a function is called by reference any change made to the reference variable will affect the original variable.

Example: Write c program to swap to numbers.

```
#include<stdio.h>

void swap(int *a, int *b);

int main()
{
    int m = 10, n = 20;
    printf("\n m = %d", m);
    printf("\n n = %d ", n);
    swap(&m, &n); //passing address of m & n to swap
    printf("\n After Swapping:\n\n");
    printf("\n m = %d", m);
    printf("\n n = %d\n ", n);
    return 0;
}/* pointer 'a' and 'b' holds and
   Points to the address of 'm' and 'n'
*/

void swap(int *a, int *b) // a=&m , b=&n
{
    int temp;
    temp = *a;
    *a = *b;
    *b = temp;
}
```

Output:

m = 10

n = 20

After Swapping:

m = 20

n = 10

PART -2 **Recursion**

Syllabus:

Recursion: Simple programs, such as Finding Factorial, Fibonacci series etc.,
Limitations of Recursive functions

Recursion

Definition:

1. The process in which a function calls itself directly or indirectly is called recursion
2. The function call itself is called recursion.


EXAMPLE:

```
void recursion()  
{  
... ..  
    recursion(); /* function calls itself */  
    ... ..  
}  
int main()  
{  
... ..  
    recursion();  
... .. }
```

How does recursion work?

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

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



There are two approaches to writing repetitive algorithms.

1. Loops (Iteration)
2. Recursion.

Iteration: Use for loops, do..while, while loops.

Recursion: The function call itself is called recursion.

Advantages of Recursion

1. Using recursion we can avoid unnecessary calling of functions.
2. The recursion is very flexible in data structure
3. Using recursion, the length of the program can be reduced.

Recursion Disadvantages: (Limitations of Recursive)

1. Recursive solution is always logical and it is very difficult to trace.
(debug and understand).
2. Recursion takes a lot of stack space,
3. Recursion uses more processor time.

Examples of such complex problems(Application of Recursive functions)

1. Towers of Hanoi (TOH),
2. Inorder/Preorder/Postorder Tree Traversals,
3. DFS of Graph, etc

1. Write c program to calculates the factorial of a given number using a function (Iteration)

```
#include<stdio.h>
long factorial(long n);
void main()
{
    long int n, fact;
    printf("\n Enter any positive number : \n");
    scanf("%ld",&n);
    fact = factorial(n);
    printf("\n The Factorial of %ld is %ld",n, fact);
}
long factorial(long n)
{
    int i,f=1;
    for(i=1;i<=n;i++)
    f=f*i;
    return f;
}
```

Output:

Enter any positive number:

5

The Factorial of 5 is 120

2. Write c program to calculates the factorial of a given number using a recursive function

```
#include<stdio.h>
long factorial(long n);
void main()
{
    long int n, fact;
    printf("\n Enter any positive number : ");
    scanf("%ld",&n);
    fact = factorial(n);
    printf("\n\tThe Factorial of %ld is %ld",n, fact);
}
long factorial(long n)
{
    if (n== 1)
        return 1;
    else
        return n*factorial(n-1);
}
```

Output:

Enter any positive number:

5

The Factorial of 5 is 120

Explanation:

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

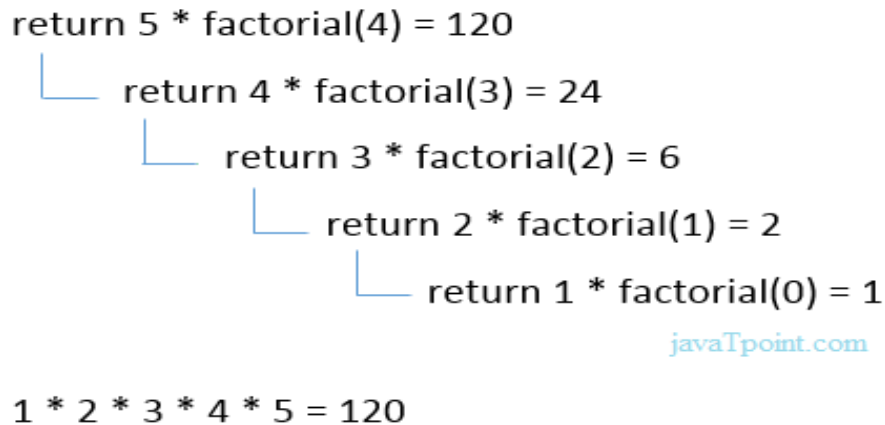


Fig: Recursion

Write c program to calculates the factorial of a given number using a recursive function

```
#include <stdio.h>

long factorial(long n);

void main()
{
    long int n,fact;

    printf("\n Enter a number to find it's Factorial:\n ");
    scanf("%d", &n);

    if (n< 0)
        printf("Factorial of negative number not possible\n");
    else
    {
        fact = factorial(n);
        printf("The Factorial of %ld is %ld.\n", n, fact);
    } }

long factorial(long n)
```

```

{
  if (n== 0 || n== 1)
  {
    return 1;
  }
  else
  {
    return n* factorial(n - 1);
  }
}

```

Output:

Enter a number to find it's Factorial:

5

The Factorial of 5 is 120.

3. C Program to print Fibonacci sequence using function (Iteration)

```

#include<stdio.h>
void fib(int n); // function declaration
void main()
{
  int n;
  printf("\nEnter a number for fibonacci series n:\n");
  scanf("%d",&n);
  fib(n); //function call
}
void fib(int n) //function defination
{
  int i,c=0;
  int a=0;
  int b=1;
  printf("\nFibonacci series for %d terms:-\n", n);
  printf("\n %d \t %d",a,b);

```

```

for(i=2;i<=n;i++)
{
c=a+b;
printf("%d\t",c);
a=b;
b=c;
}
}

```

Output:

Enter a number for fibonacci series n:

9

Fibonacci series for 9 terms

0 1 1 2 3 5 8 13 21

4. C Program to print Fibonacci sequence using recursion

Fibonacci sequence
0 1 1 2 3 5 8 13 21 34

```

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

void main()
{
int n,i;

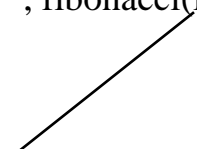
printf("\n Enter a number for fibonacci series n: ");
scanf("%d", &n);

printf ("\nThe fibonacci series is:\n");

for(i = 0; i < n; i++)
printf("%d\t ", fibonacci(i))
}

int fibonacci(int n) // recursive function
{
//base condition

```



```
if(n== 0 || n== 1)
{
    return n;
}

else
{
    // recursive call/ function
    return fibonacci(n-1) + fibonacci(n-2);
}
}
```

Output:

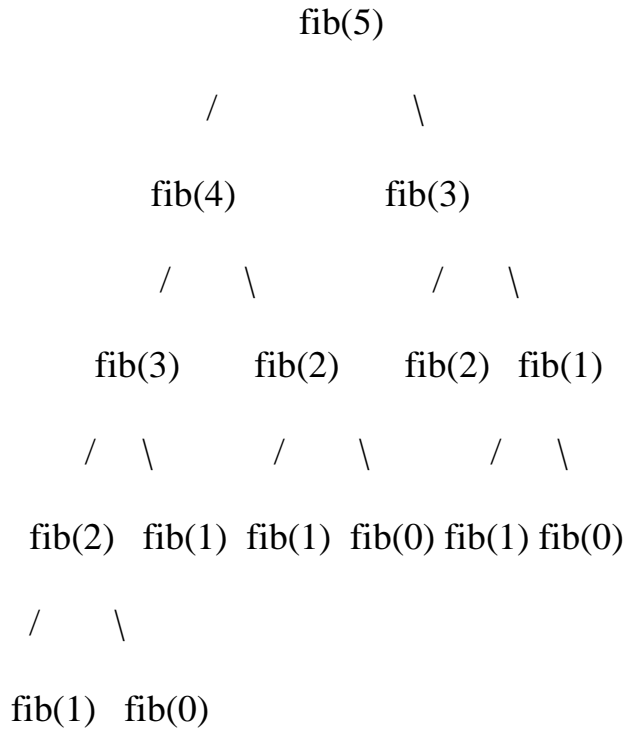
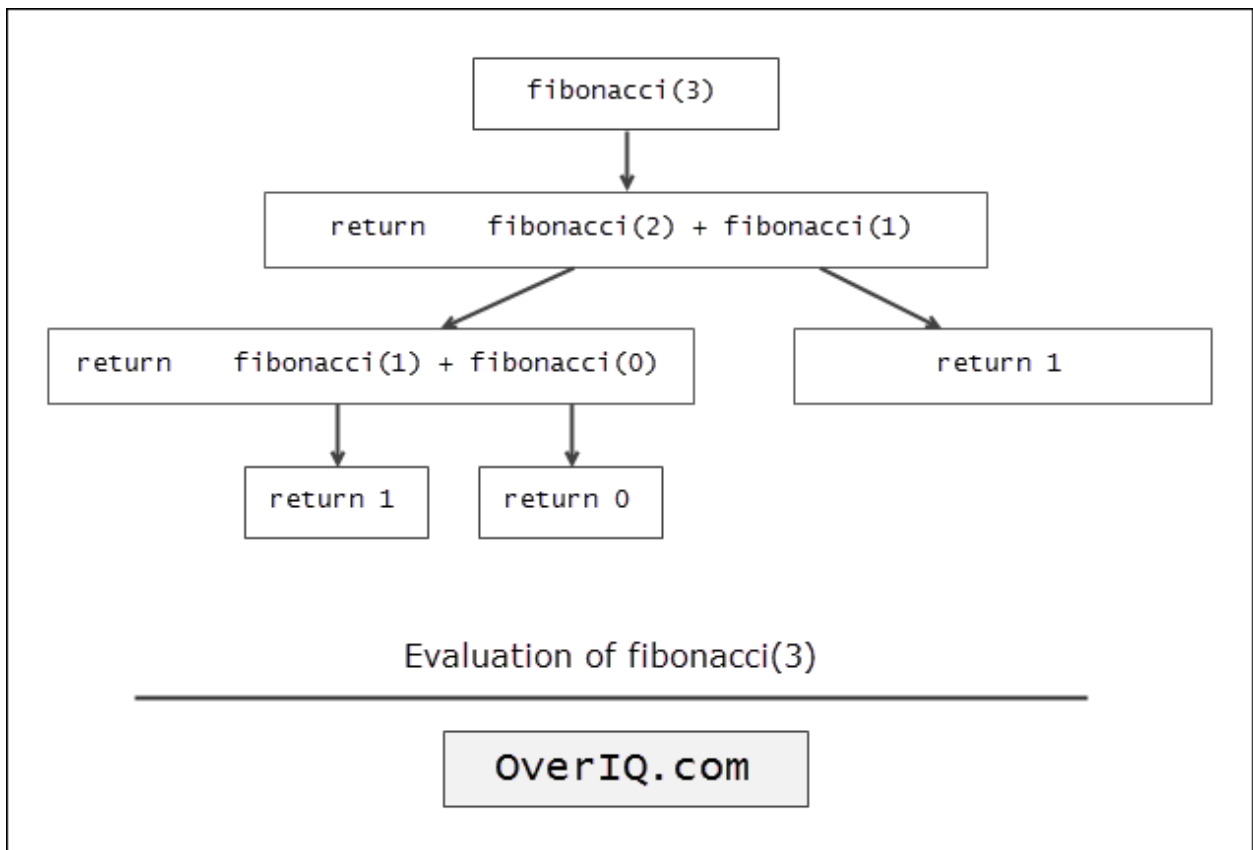
Enter a number to generate fibonacci series for first n terms:

9

The fibonacci series is:

0 1 1 2 3 5 8 13 21

Explanation:



GCD of Two Numbers in C

According to Mathematics, the Greatest Common Divisor (GCD) of two or more integers is the largest positive integer that divides the given integer values without the remainder.

For example, the GCD of two numbers in C, i.e., integer 8 and 12 is 4 because, both 8 and 12 are divisible by 1, 2, and 4 (the remainder is 0) and the largest positive integer among the factors 1, 2, and 4 is 4.

The Greatest Common Divisor (GCD) is also known as the Highest Common Factor (HCF), or Greatest Common Factor (GCF), or Highest Common Divisor (HCD), or Greatest Common Measure (GCM).

NOTE: To find the Greatest Common Divisor or GCD in C, we have to pass at least one non-zero value.

5. To find the GCD (greatest common divisor) of two given integers using function:

```
#include <stdio.h>
void gcd(int x, int y);
void main()
{
    int x, y;
    printf("\n Enter two integer values: \n");
    scanf("%d%d", &x, &y);
    gcd(x,y);
}

void gcd(int x, int y);
{
    int i,gcd;
    for (i = 1; i <= x && i <= y; i++)
    {
        if (x % i == 0 && y % i == 0)
            gcd = i;
    }
    printf("\n GCD of %d and %d is: %d\n", x, y, gcd);
}
```


Output:

Enter two integer values:

16

40

GCD of 16 and 40 is: 8

Program Explanation

In this program, two integers entered by the user are stored in variable x and y. Then, for loop is iterated until i is less than x and y.

In each iteration, if both x and y are exactly divisible by i, the value of i is assigned to gcd.

When the for loop is completed, the greatest common divisor of two numbers is stored in variable gcd.

To find the GCD (greatest common divisor) of two given integers using function

```
#include <stdio.h>
int gcd(int a, int b);
void main()
{
    int a, b, result;
    printf("Enter the two numbers to find their GCD:\n");
    scanf("%d%d", &a, &b);
    result = gcd(a, b);
    printf("The GCD of %d and %d is %d.\n", a, b, result);
}
int gcd(int a, int b)
{
    while (a != b)
    {
```

```
if (a > b)
{
    return gcd(a - b, b);
}
else
{
    return gcd(a, b - a);
}
}
return a;
}
```

Output:

Enter the two numbers to find their GCD: 100 70

The GCD of 100 and 70 is 10.

Program Explanation

In this C Program, we are reading the two integer numbers using 'a' and 'b' variable. The gcd() function is used to find the GCD of two entered integers using recursion.

While loop is used to check that both the 'a' and 'b' variable values are not equal. If the condition is true then execute the loop. Otherwise, if the condition is false return the value of 'a' variable. If else condition statement is used to check the value of 'a' variable is greater than the value of 'b' variable.

If the condition is true, then return two integer variable values. Otherwise, if the condition is false, then execute else statement and return the values of two integer variable. Print the GCD of a given number using printf statement.

1. C program to find the GCD (greatest common divisor) of two given integers using recursive function

```
#include <stdio.h>
long gcd(long x, long y);
void main()
{
    int n,m;
    printf("\n Please Enter two positive integer Values: \n");
    scanf("%d %d", &n, &m);
    printf("\n GCD of %d and %d is = %d",n, m, gcd(n, m));
}
long gcd(long x, long y)
{
    if (y == 0)
    {
        return x;
    }
    else
    {
        return gcd(y, x % y);
    }
}
```

Output:

Please Enter two positive integer Values:

16

40

GCD of 16 and 40 is: 8

2. C program to find the GCD (greatest common divisor) of two given integers using recursive function

```
#include <stdio.h>
/* Function declaration */
int gcd(int a, int b);
void main()
{
    int n, m, g;
    /* Input two numbers from user */
    printf("Enter any two numbers to find GCD:\n");
    scanf("%d%d", &n, &m);
    g = gcd(n, m);
    printf("GCD of %d and %d = %d", n, m,g)
}
/**
 * Recursive approach to find GCD of two numbers
 */
int gcd(int a, int b)
{
    if(b == 0)
        return a;
    else
        return gcd(b, a%b);
}
```

Output:

Enter any two numbers to find GCD: 12

30

GCD of 12 and 30 = 6

PART-3:

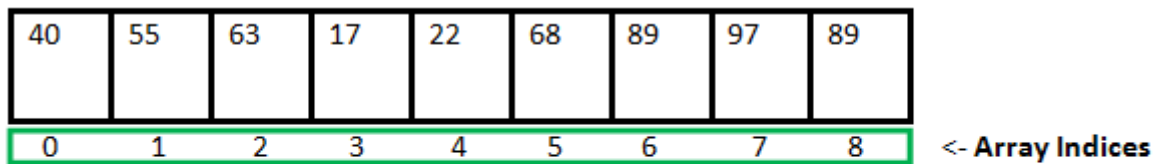
Dynamic memory allocation

Syllabus:

Dynamic memory allocation: Allocating and freeing memory, Allocating memory for arrays of different data types.

Notes:

Since C is a structured language, it has some fixed rules for programming. One of it includes changing the size of an array. An array is collection of items stored at continuous memory locations.



Array Length = 9

First Index = 0

Last Index = 8

As it can be seen that the length (size) of the array above made is 9. But what if there is a requirement to change this length (size).

Static memory allocation have the two problem:

Problem 1: where only 5 elements are needed to be entered in this an array. In this case, the remaining 4 indices are just wasting memory in this array. So there is a requirement to lessen the length (size) of the array from 9 to 5.

Problem 2: In this, there is an array of 9 elements with all 9 indices filled. But there is a need to enter 3 more elements in this array. In this case 3 indices more are required. So the length (size) of the array needs to be changed from 9 to 12.

Sometimes the size of the array you declared may be insufficient. To solve this issue, you can allocate memory manually during run-time. This is known as dynamic memory allocation in C programming.

The process of allocating memory during program compile time is called static memory allocation.

DYNAMIC MEMORY ALLOCATION IN C:

The process of allocating memory during program execution is called dynamic memory allocation.

C language offers 4 dynamic memory allocation functions. They are,

1. malloc()
2. calloc()
3. realloc()
4. free()

Difference between static memory allocation and dynamic memory allocation

Static memory allocation	Dynamic memory allocation
Memory is allocated at compile time.	Memory is allocated at run time.
Memory can't be increased while executing program.	Memory can be increased while executing program.
Used in array.	Used in linked list.

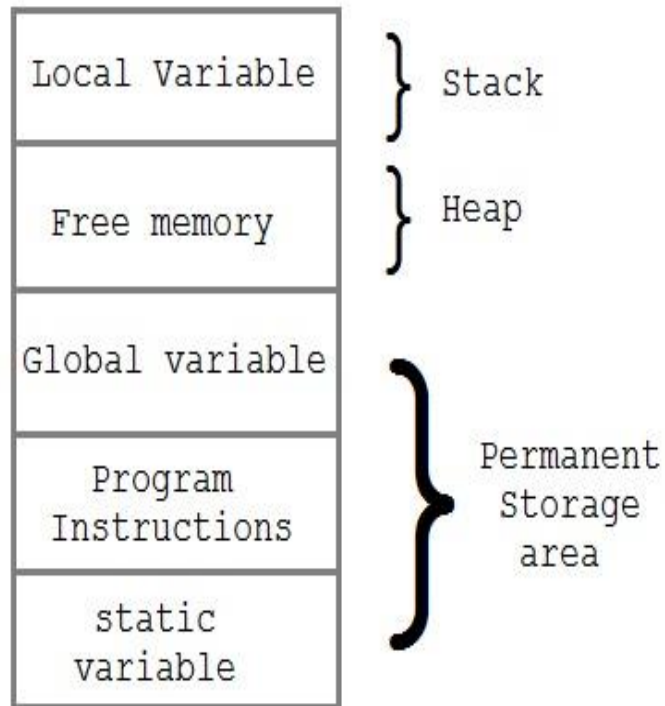
Memory Allocation Process

Global variables, static variables and program instructions get their memory in permanent storage area whereas **local variables** are stored in a memory area **called Stack**.

The memory space between these two regions is known as **Heap area**. This region is used for **dynamic memory allocation** during execution of the program. The sizes of **heap area** keep changing.

Therefore, C Dynamic Memory Allocation can be defined as a procedure in which the size of a data structure (like Array) is changed during the **runtime**.

C



There are 4 library functions provided by defined under `<stdlib.h>`

header file to facilitate dynamic memory allocation in C programming.

They are:

1. `malloc()`
2. `calloc()`
3. `free()`
4. `realloc()`

1. **malloc()-(memory allocation)**

- The name "malloc" stands for memory allocation.
- `malloc ()` function is used to allocate single block of memory during the execution of the program.
- `malloc ()` does not initialize the memory allocated, It carries garbage value by default
- `malloc ()` function returns null pointer if it couldn't able to allocate requested amount of memory.

The syntax of malloc() function is given below:

```
datatype *ptr;
```

```
ptr=(cast-type*)malloc(byte-size)
```

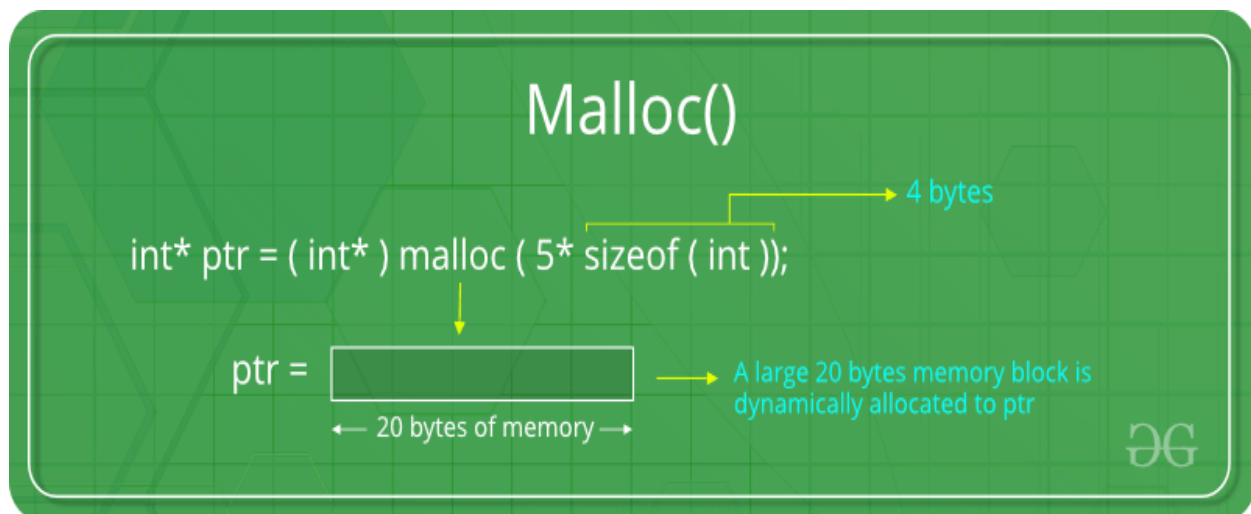
For Example:

```
int *ptr;
```

```
ptr = (int*) malloc(100 * sizeof(int));
```

Since the size of int is 4 bytes, this statement will allocate 400 bytes of memory.

And, the pointer ptr holds the address of the first byte in the allocated memory.



Example:

```
int *x;
```

```
x = (int*)malloc(50 * sizeof(int));
```

```
//memory space allocated to variable x
```

```
free(x); //releases the memory allocated to variable x
```

Note: If space is insufficient, allocation fails and returns a NULL pointer.

1. Write c program read n elements from key find the sum.Using dynamic memory allocation

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
    int n,i,*ptr,sum=0;
    printf("Enter number of elements:\n ");
    scanf("%d",&n);
    ptr=(int*)malloc(n*sizeof(int));
//memory allocated using malloc
    if(ptr==NULL)
    {
        printf("\n Sorry! unable to allocate memory\n");
        exit(0);
    }
    printf("Enter elements of array:\n ");
    for(i=0;i<n;i++)
    {
        scanf("%d",ptr+i); // ptr contain address of block
        sum=sum+*(ptr+i); // like 1-D array
    }
    printf("\n Sum=%d",sum);
    free(ptr);
return 0;
}
```

Output:

Enter elements of array: 3

Enter elements of array:

3

4

5

Sum=12

2. calloc or contiguous allocation

Calloc method in C is used to dynamically allocate the specified number of blocks of memory of the specified type. It initializes each block with a default value '0'.

- The calloc() function allocates multiple block of requested memory.
- It initially initializes all bytes to zero.
- It returns NULL if memory is not sufficient.

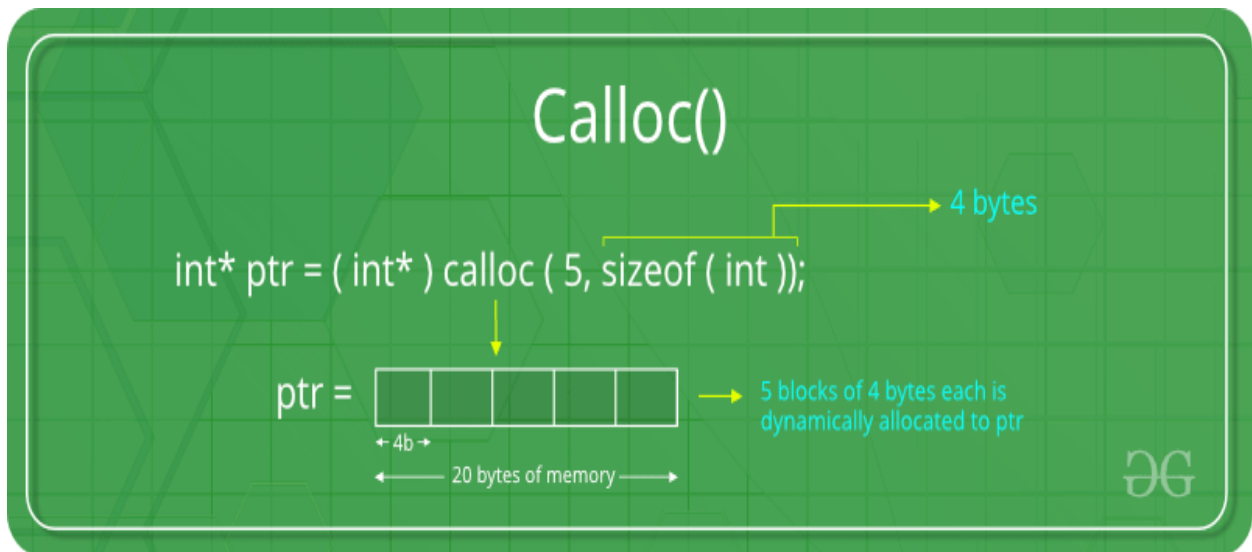
The syntax of calloc() function is given below:

```
datatype *ptr;  
ptr = (cast-type*)calloc(n, element-size);
```

For Example:

```
float *ptr;  
ptr = (float*) calloc(25, sizeof(float));
```

This statement allocates contiguous space in memory for 25 elements each with the size of the float.



Note: If space is insufficient, allocation fails and returns a NULL pointer.

Write c program read n elements from key find the sum. Using dynamic memory allocation (calloc())

```
#include<stdio.h>
#include<stdlib.h>
void main()
{
int n,i,*ptr,sum=0;
printf("\n Enter number of elements:\n ");
scanf("%d",&n);
ptr=(int*)calloc(n,sizeof(int));
//memory allocated using calloc
if(ptr==NULL)
{
printf("Sorry! unable to allocate memory");
exit(0);
}
printf("Enter elements of array: ");
for(i=0;i<n;i++)
{
scanf("%d",ptr+i); // like 1-D array
sum+=*(ptr+i);
}
printf("\n Sum=%d",sum);
free(ptr);
}
```

Output:

Enter elements of array: 3

Enter elements of array:

3

4

5

Sum=12

3. realloc() means re-allocation

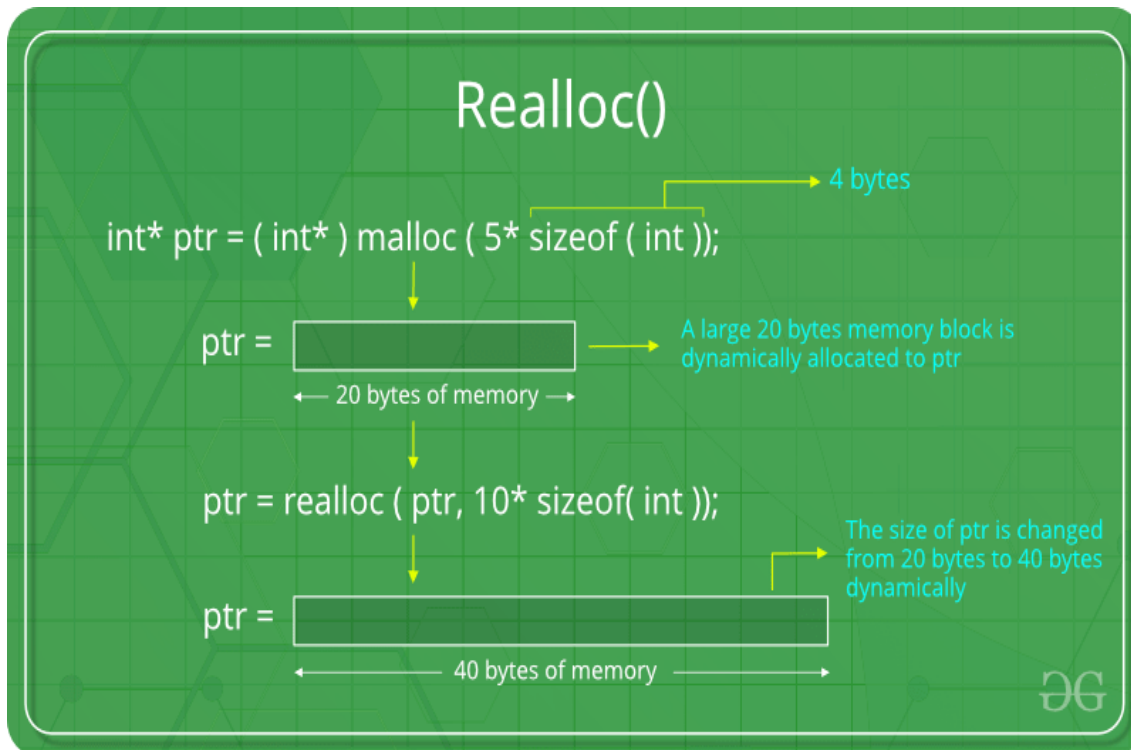
If memory is not sufficient for malloc() or calloc(), you can reallocate the memory by realloc() function. In short, it changes the memory size.

“Re-allocation” method in C is used to dynamically change the memory allocation of a previously allocated memory.

The syntax of realloc() function.

```
datatype *ptr;  
ptr = realloc(ptr, newSize);
```

where ptr is reallocated with new size 'newSize'.



4. Write a c program read elements from key board display them using calloc and realloc functions

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
    int n,n2,i,*ptr;

    printf("\n Enter number of elements:\n ");
    scanf("%d",&n);

    ptr=(int*)calloc(n,sizeof(int));
//memory allocated using calloc
    if(ptr==NULL)
    {
        printf("Sorry! unable to allocate memory");
        exit(0);
    }

    printf("\nEnter elements of array:\n ");
    for(i=0;i<n;i++)
        scanf("%d",ptr+i); // like 1-D array
    printf("\n The elements of array are:\n ");
    for(i=0;i<n;i++)
        printf("%d\n",*(ptr+i));
    printf("\n Enter the new size:\n ");
    scanf("%d", &n2);

    // relocating the memory
    ptr = realloc(ptr, n2 * sizeof(int));

    printf("\n Enter elements of new array :\n ");
    for(i=0;i<n2;i++)
        scanf("%d",ptr+i);
    printf("\n The elements of new array are:\n ");
```

```
for(i=0;i<n2;i++)
printf("%d\n",*(ptr+i));
    free(ptr);
return 0;
}
```

Output:

Enter number of elements:

3

Enter elements of array:

1

2

3

The elements of array are:

1

2

3

Enter the new size:

5

Enter elements of new array :

10

20

30

40

50

The elements of new array are:

10

20

30

40

50

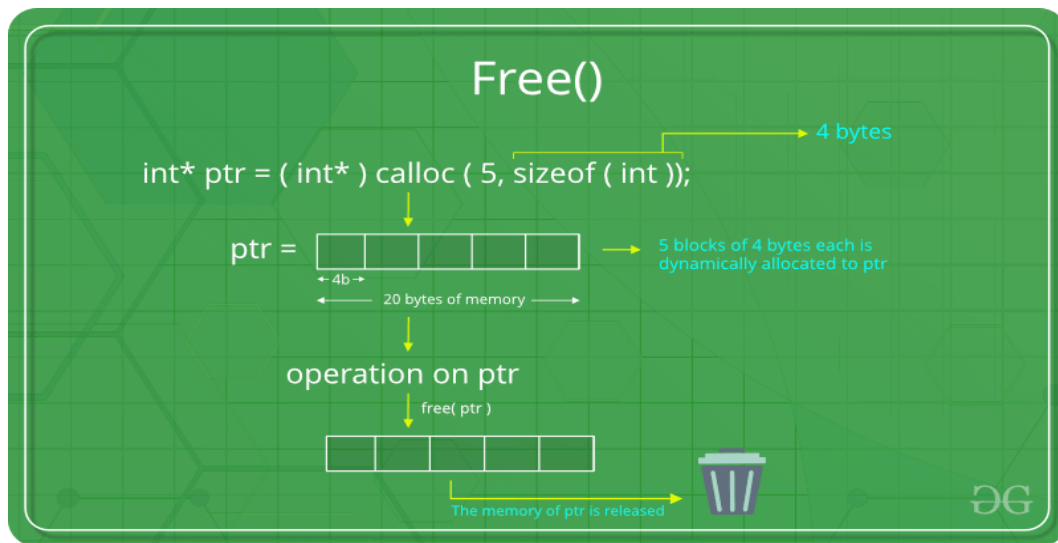
4. free () function

“free” method in C is used to dynamically de-allocate the memory. The memory allocated using functions malloc() and calloc() is **not de-allocated on their own.**

Hence the free() method is used, whenever the dynamic memory allocation takes place. It helps to reduce wastage of memory by freeing it.

Syntax:

```
free(ptr);
```



DIFFERENCE BETWEEN MALLOC() & CALLOC() FUNCTIONS IN C:

malloc()	calloc()
It allocates only single block of requested memory	It allocates multiple blocks of requested memory
<pre>int *ptr; ptr = malloc(20 * sizeof(int));</pre> For the above, 20*4 bytes of memory only allocated in one block. Total = 80 bytes	<pre>int *ptr; ptr = calloc(20, 20 * sizeof(int));</pre> For the above, 20 blocks of memory will be created and each contains 20*4 bytes of memory. Total = 1600 bytes
malloc () doesn't initialize the allocated memory. It contains garbage values	calloc () initializes the allocated memory to zero
type cast must be done since this function returns void pointer int *ptr; <pre>ptr = (int*)malloc(sizeof(int)*20);</pre>	Same as malloc () function int *ptr; <pre>ptr = (int*)calloc(20, 20 * sizeof(int));</pre>

UNIT-4

(FUNCTIONS AND DYNAMIC MEMORY ALLOCATION)

QUESTION BANK (IMP QUESTIOS)

SHORT ANSWER QUESTIONS (PART-A)

1. Define functions. What are types of functions?
2. How to Declaring a function? Illustrate a example.
3. What need of function in c?
4. Write short note on modular programming.
5. List the application and advantages of functions.
6. Distinguish between built – in and user – defined functions.
7. **What is the difference between actual and formal parameters?
8. Differentiate between a macro and a function.
9. Discuss the need of pointers to functions.
10. Define recursion, what are the advantages and limitations of recursion?
11. **Write brief notes on memory allocation functions.
12. *What is the difference between calloc and malloc functions?

LONG ANSWER QUESTIONS (PART-B)

1. Explain the different standard library functions used in C.
2. Explain the different categories of functions with an example.
3. With illustrative examples explain parameter passing techniques.
4. **Explain the difference between call by value and call by reference.
5. **Write a C program on call by value and call by reference.
6. **What are parameter passing, Explain call by value and call by reference.
7. *Write a recursive function to print Fibonacci sequence.
8. **Write recursive and iterative approaches programs to find factorial of a given number.
9. How to pass arrays as argument to a function? Illustrate.
10. How to declare a pointer to a function? What is its use? Illustrate. // call by reference
11. **Write in detail about the various dynamic memory allocation functions.
12. Write a ‘C’ program on malloc, calloc, realloc and free functions.

