FUNCTIONS

Definition : A function is self contained block of statements that performs a particular task.

C -functions are divided into two parts

1) Pre-defined functions(library functions)
1) User-defined functions

library functions OR pre-defined functions are not required to be written by us where as user define function has to be developed by the user at the time of writing program. Ex: printf(), scanf(), sqrt(), pow() etc.,,,

To make use of user defined function, we need to write three elements that are related to function.

1) Function declaration
1) Function definition
1) Function call

Function declaration / signature / prototype :

Just like variables , functions are also to be declared before we use them in the program. A function declaration consists of four parts.

1) Function type(return type)
1) Function name
1) Parameter List
1) Terminating semicolon.

The general format of declaring a function is :

return_type function_name(parameter list);

```
Example:  void display( );
          int square(int m);
         int mul(int m, int n);
```

A prototype declaration may be placed in two places in a program

1) Above all the functions (including main)
2) Inside a function definition.

- When we place the declaration above all the functions (in the global declaration section), the prototype is referred to as a global prototype. Such declarations are available for all the functions in the program.
- When we place it in a function definition (in the local declaration section), this prototype is called local prototype.

Function call : is done inside main(). Whenever function call is made control from main() will be transfered to the definition of that function. After executing all the statements in tha function control comes back to main().

General form of function call : **function_name( ) ;**

If arguments are being passed then : **function_name(arguments);**

Function definition : (also known as FUNCTION IMPLEMENTATION)

format of function definition is :-

```
return_type  fuction_name(arguments)
     {
               Local variable declarations;
               stmt 1;
               stmt 2;
               --------
               ----------
                return statement;
       }
```

return type **specifies the type of value that the function is going to return. Various return types are ………void, int, char, float, double, long int etc.,,**

function name **is the name of function. It is given the user.**

- A FUNCTION IN WHICH FUNCTION CALL IS MADE IS KNOWN AS CALLING FUNCTION.
- A FUNCION WHICH IS CALLED BY OTHER FUNCTION IS KNOWN AS CALLED FUNCTION.

Example :

```
main ()
{
  printf("hello");
  display();
}
```

In above sample code **display( ) is called function** and **main( ) is calling function.**

main ( ) is a **specially recognized user-defined function**. That is ,

- **Declaration** of main( ) is **pre-defined**
- **Definition** of main( ) is **user-defined**

Arguments :

The mechanism of passing values between calling function and called function is known as ARGUMENT or PARAMETER.

There are 2 types of arguments or parameters :

1. Actual arguments
2. Formal arguments

- ➔ Variables in function call are actual arguments.
- ➔ Variables in function definition are formal arguments.

The return statement can take one of the following forms

```
return;
  or
return(expression);
```

The first one does not return any value; it acts much as closing brace of the function. When a return is encountered, the control is immediately passed back to the calling function.

An example of the use of a simple return is as follows

```
if(error)
return;
```

The second form of the return with an expression returns the value of the expression. For example the function

```
int mul(int x,int y)
{
      int p;
      p=x*y;
      return(p);
}
```

returns the value of the p which is the product of the values of x and y. The last two statements can be combined into one statement as follows.

```
return(x*y);
```

A function may have more than one return statement this situation arises when the value returned is based on certain conditions. For example

```
if(x<=0)
  return(0);
```

```
        else
          return(1);
```

TYPES OF FUNCTIONS BASED ON ARGUMENTS AND RETURN VALUES :

1. Functions without arguments and without return values
2. Functions with arguments and without return values
3. Functions with arguments and with return values
4. Functions without arguments and with return values

Functions without arguments and without return values

- No values will be passed from calling function to called function.
- No values will be returned to calling function from called function.

Example program :

```
#include<stdio.h>
void add( ); // function declaration
main()  //  calling function
{
  add();  // function call
}

void add( )  // function definition and called function
{
 int x,y,z;
 printf("enter any two integers");
 scanf("%d %d", &x,&y);
 z=x+y;
 printf("sum = %d",z);
}
```

Functions with arguments and without return values

- (arguments) values will be passed from calling function to called function.
- No values will be returned to calling function from called function.

Example program :

```
#include<stdio.h>

void add( int a, int b ); // function declaration ;  'a' and 'b' are formal arguments

main()  // calling function

{

  int x,y ;

  printf("enter any two integers");

  scanf("%d %d", &x,&y);

  add(x,y);  // function call ;  'x' and 'y' are actual arguments

}

void add( int a, int b) // function definition  and called function

{

 int  c;

 c = a + b;

 printf("sum = %d",c );

}
```

Functions with arguments and with return values

- (arguments) values will be passed from calling function to called function.
- values will be returned to calling function from called function.

Example program :

```
#include<stdio.h>

int add(int a, int b ); // function declaration ;  'a' and 'b' are formal arguments

main() // calling function

{

  int x,y,z ;

  printf("enter any two integers");

  scanf("%d %d", &x,&y);

  z = add(x,y);  // function call ;  'x' and 'y' are actual arguments

  printf("sum = %d",z);

}

int  add( int x, int y) // function definition  and called function

{

 int c;

 c = x+y;

 return c;
}
```

Functions without arguments and with return values

- No values will be passed from calling function to called function.
- values will be returned to calling function from called function.

Example program :

```
#include<stdio.h>

int  add( ); // function declaration

main() // calling function

{

  int c;

  c = add( ); // function call

  printf("sum = %d",c);

}

int add( ) // function definition  and called function

{

  int x,y,z ;

  printf("enter any two integers");

  scanf("%d %d", &x,&y);

  z = x+y;
```

```
    return z;
  }
```

SCOPE OF A VARIABLE : is the extent to which it can be used in a program.

Based on this, variables are classified as LOCAL VARIABLES & GLOBAL VARIABLES.

- **Local variable** is restricted to a particular block in the program.
- **Global variable** is used in the entire program that is in all the functions present in the program.

Example :

```
#include<stdio.h>
int k=20; // global variable
void add ( );
main()
{
 int c =30; // 'c' is local variable to main( )
 add ( );
 printf("sum = %d", ( c+k ) ); // output is 50
}
```

```
void add( )

{

  int h = 40; // 'h' is local variable to add( )

  printf(" addition = %d", ( h+k ) ); // output is 60

}
```

RECURSION

It is a modular programming technique.

A function calling itself is known as recursive function i.e, function call statement for calling the same function appears inside the function.

In recursion, the same function acts as calling function and called function.

Ex:

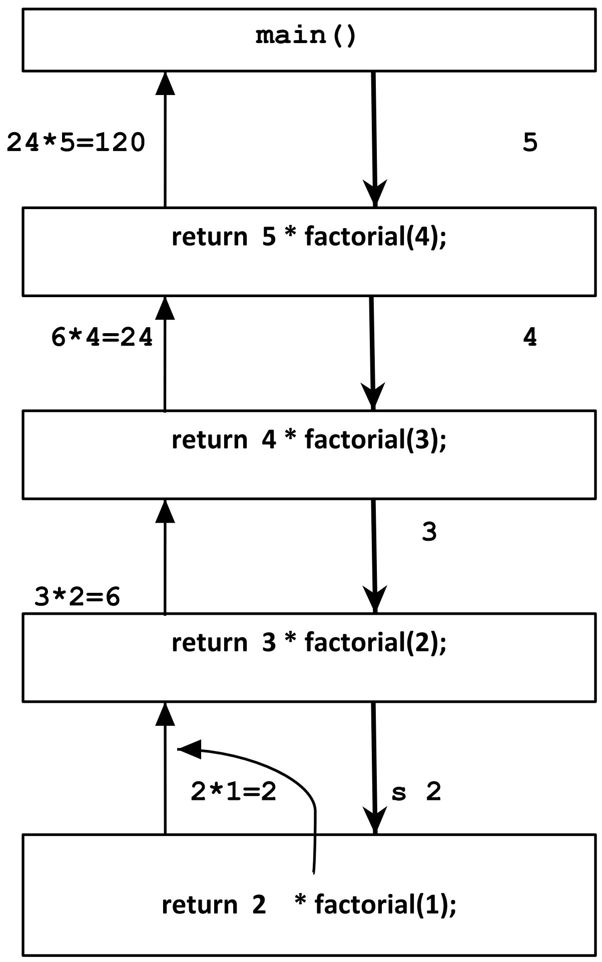
```
int main()
{
    Printf("here main() is a recursion function");
    main();
}
```

There must be an exclusive terminating condition or else the execution will continue indefinitely.

Program to find factorial of number using recursion

```
#include<stdio.h>
int factorial( int k);
int main()
{
  int s,n;
  printf("enter a no for printing factorials from 1 to entered no. ");
  scanf("%d",&n);
  s = factorial(n);
  printf(" factorial = %d", s);
}
 int  factorial (int k )
{
  if ( k==1)
   return  1;
  else
 return  k * factorial(k-1);
}
```

Consider k = 5



In the above program :-

- **main() is calling function for factorial (5)**
- **factorial (5) is calling function for factorial(4)**
- **factorial(4) is calling function for factorial(3)**
- **factorial(3) is calling function for factorial(2)**
- **factorial(2) is calling function for factorial(1)**

so, called function will return the value to its calling function.

Difference between recursion and iteration:

Both recursion and iteration are based on control structures.

- Iteration uses repetition structures to achieve looping and recursion uses selection structure to make repeated function calls.
- Both iteration and recursion use a condition to terminate. Iteration terminates when condition in loop fails. Recursion terminates when condition in 'if' statement becomes TRUE.
- Recursion helps us to solve a complex problem by breaking the program into smaller problems, that are similar to original problem. But each recursive call creates another copy of function in memory which consumes more and more memory. Thus, recursion should be applied only to larger programs.

Program to find " x power y " using recursion

```
#include<stdio.h>
int power(int,int);
int  main()
{
        int b,e,res;
        printf("Enter the value of base and exponent");
        scanf("%d%d",&b,&e);
        res = power(b,e);
        printf("value of %d power %d is %d",b,e,res);
        }

int power(int b, int e)
{
        if(e==0)
                return 1;
        else if(e==1)
                return b;
        else
                return  (b * power(b,e-1));
}
```

Program to find fibonacii series using recursion

```
#include<stdio.h>
int  main()
{
int i,n;
printf("Enter the n value:");
scanf("%d",&n);
printf("the fibinacci series is\n");
for(i=0;i<n;i++)
{
printf("%d\n",fib(i));
}
}


int fib(int n)
{
if(n==0)
{
return(0);
}
if(n==1)
{
return(1);
}
else
{
return(fib(n-1)+fib(n-2));
}
}
```

Program for binary search using recursion

```
#include  <stdio.h>
main()
{
               int a[50];
                int n,no,x,result;

                printf("Enter the number of terms : ");
                scanf("%d",&no);

                printf("Enter the elements :\n");
                for(x=0;x<no; x++)
                 scanf("%d",&a[x]);

                printf("Enter the number to be searched : ");
                scanf("%d",&n);

                result = binarysearch(a, n, 0, no-1);  // function call

                if(result == -1)
                 printf("Element not found");
                return 0;
}

binarysearch(int a[] ,int n, int low ,int high)
{
               int mid;
                if (low > high)
                 return -1;

                mid = (low + high)/2;

                if(n == a[mid])
                  {
                   printf("The element is at position %d\n",mid+1);
                   return 0;
                  }
                if(n < a[mid])
                  {
                   high = mid - 1;
                   binarysearch(a, n, low ,high);  // recursive call
                  }
                if(n > a[mid])
                  {
                     low = mid + 1;
                   binarysearch(a, n, low ,high);  // recursive call

                  }
 }
```

The concept of **dynamic memory allocation in c language** *enables the C programmer to allocate memory at runtime*. Dynamic memory allocation in c language is possible by 4 functions of stdlib.h header file.

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

Before learning above functions, let's understand the 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.

Now let's have a quick look at the methods used for dynamic memory allocation.

malloc()	allocates single block of requested memory.
calloc()	allocates multiple block of requested memory.
realloc()	reallocates the memory occupied by malloc() or calloc() functions.
free()	frees the dynamically allocated memory.

malloc() function in C

The malloc() function allocates single block of requested memory.

It doesn't initialize memory at execution time, so it has garbage value initially.

It returns NULL if memory is not sufficient.

The syntax of malloc() function is given below:

1. ptr=(cast-type*)malloc(byte-size)

calloc() function in C

The calloc() function allocates multiple block of requested memory.

It initially initialize all bytes to zero.

It returns NULL if memory is not sufficient.

The syntax of calloc() function is given below:

1. ptr=(cast-type*)calloc(number, byte-size)

realloc() function in C

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

Let's see the syntax of realloc() function.

1. ptr=realloc(ptr, **new**-size)

free() function in C

The memory occupied by malloc() or calloc() functions must be released by calling free() function. Otherwise, it will consume memory until program exit.

Let's see the syntax of free() function.

- free(ptr)

Example on Malloc()

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

```
▪  int main(){
▪    int n,i,*ptr,sum=0;
▪      printf("Enter number of elements: ");
▪      scanf("%d",&n);
▪      ptr=(int*)malloc(n*sizeof(int));  //memory allocated using malloc
▪      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);
▪          sum+=*(ptr+i);
▪      }
▪      printf("Sum=%d",sum);
▪      free(ptr);
▪  return 0;
▪  }
```

Example on calloc()

```
▪  #include<stdio.h>
▪  #include<stdlib.h>
▪  int main(){
▪   int n,i,*ptr,sum=0;
▪      printf("Enter number of elements: ");
▪      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);
        sum+=*(ptr+i);
    }
    printf("Sum=%d",sum);
    free(ptr);
return 0;
}
```

Command Line Arguments in C

The arguments passed from command line are called command line arguments. These arguments are handled by main() function.

To support command line argument, you need to change the structure of main() function as given below.

```
int main(int argc, char *argv[] )
```

Here, **argc** counts the number of arguments. It counts the file name as the first argument.

The **argv[]** contains the total number of arguments. The first argument is the file name always.

Functions in C/C++

A function is a set of statements that take inputs, do some specific computation and produces output.

The idea is to put some commonly or repeatedly done task together and make a function so that instead of writing the same code again and again for different inputs, we can call the function.

Example:

```
#include <stdio.h>

// An example function that takes two parameters 'x' and 'y'
// as input and returns max of two input numbers
int max(int x, int y)
{
    if (x > y)
      return x;
    else
      return y;
}

// main function that doesn't receive any parameter and
// returns integer.
int main(void)
{
    int a = 10, b = 20;

    // Calling above function to find max of 'a' and 'b'
    int m = max(a, b);

    printf("m is %d", m);
    return 0;
}
```

Output:

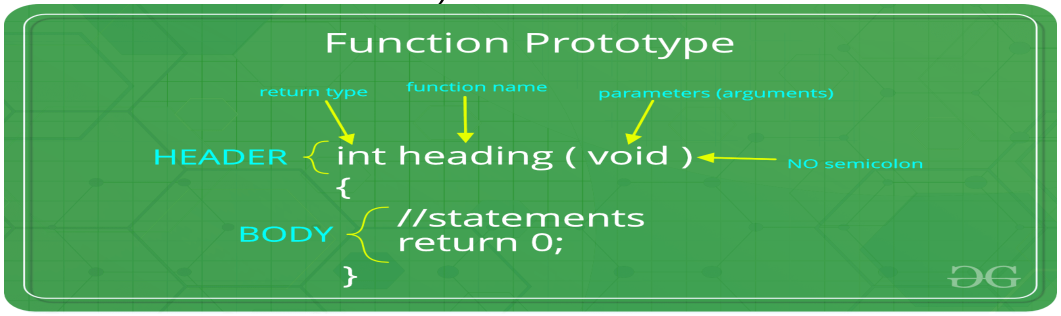
```
m is 20
```

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 codes. 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 working.

Function Declaration

A function declaration tells the compiler about the number of parameters function takes, data-types of parameters and return type of function. Putting parameter names in function declaration is optional in the function declaration, but it is necessary to put them in the definition. Below are an example of function declarations. (parameter names are not there in below declarations)



```
int heading ( void )
{
  //statements
  return 0;
}
```

```
// A function that takes two integers as
parameters
// and returns an integer
int max(int, int);

// A function that takes a int pointer
and an int variable as parameters
// and returns an pointer of type int
int *swap(int*,int);

// A function that takes a charas
parameters
// and returns an reference variable
char *call(char b);

// A function that takes a char and an
int as parameters
// and returns an integer
int fun(char, int);
```

It is always recommended to declare a function before it is used

In C, we can do both declaration and definition at the same place, like done in the above example program.

C also allows to declare and define functions separately, this is especially needed in case of library functions. The library functions are declared in header files and defined in library files. Below is an example declaration.

Parameter Passing to functions

The parameters passed to function are called ***actual parameters***. For example, in the above program 10 and 20 are actual parameters.
The parameters received by function are called ***formal parameters***. For example, in the above program x and y are formal parameters.

There are two most popular ways to pass parameters.

Pass by Value: 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.

Pass by Reference Both actual and formal parameters refer to same locations, so any changes made inside the function are actually reflected in actual parameters of caller.

In C, parameters are always passed by value. Parameters are always passed by value in C. For example. in the below code, value of x is not modified using the function fun().

```
#include <stdio.h>
void fun(int x)
{
   x = 30;
}

int main(void)
{
    int x = 20;
    fun(x);
    printf("x = %d", x);
    return 0;
}
```

Output:

```
x = 20
```

However, in C, we can use pointers to get the effect of pass by reference. For example, consider the below program. The function fun() expects a pointer ptr to an integer (or an address of an integer). It modifies the value at the address ptr. The dereference operator * is used to access the value at an address. In the statement '*ptr = 30', value at address ptr is changed to 30. The address operator & is used to get the address of a variable of any data type. In the function call statement 'fun(&x)', the address of x is passed so that x can be modified using its address.

```
# include <stdio.h>
void fun(int *ptr)
{
    *ptr = 30;
}

int main()
{
  int x = 20;
  fun(&x);
  printf("x = %d", x);

  return 0;
}
```

Output:

```
x = 30
```

Following are some important points about functions in C.
1) Every C program has a function called main() that is called by operating system when a user runs the program.

2) Every function has a return type. If a function doesn't return any value, then void is used as return type. Moreover, if the return type of the function is void, we still can use return statement in the body of function definition by not specifying any constant, variable, etc. with it, by only mentioning the 'return;' statement which would symbolise the termination of the function as shown below:

```
void function name(int a)
{
.......  //Function Body
return;  //Function execution would get
terminated
}
```

3) In C, functions can return any type except arrays and functions. We can get around this limitation by returning pointer to array or pointer to function.

4) Empty parameter list in C mean that the parameter list is not specified and function can be called with any parameters. In C, it is not a good idea to declare a function like fun(). To declare a function that can only be called without any parameter, we should use “void fun(void)”.
As a side note, in C++, empty list means function can only be called without any parameter. In C++, both void fun() and void fun(void) are same.

5)If in a C program, a function is called before its declaration then the C compiler automatically assumes the declaration of that function in the following way:
int function name();
And in that case if the return type of that function is different than INT ,compiler would show an error.

Recursion

What is Recursion?
The process in which a function calls itself is called recursion and the corresponding function is called as recursive function. Using recursive algorithm, certain problems can be solved quite easily. Examples of such problems are Towers of Hanoi (TOH), Inorder/Preorder/Postorder Tree Traversals, DFS of Graph, etc.

What is base condition in recursion?
In the recursive program, the solution to the base case is provided and the solution of the bigger problem is expressed in terms of smaller problems.

```
int fact(int n)
{
   if (n < = 1) // base case
      return 1;
   else
      return n*fact(n-1);
}
```

In the above example, base case for n < = 1 is defined and larger value of number can be solved by converting to smaller one till base case is reached.

How a particular problem is solved using recursion?
The idea is to represent a problem in terms of one or more smaller problems, and add one or more base conditions that stop the recursion. For example, we compute factorial n if we know factorial of (n-1). The base case for factorial would be n = 0. We return 1 when n = 0.

Why Stack Overflow error occurs in recursion?
If the base case is not reached or not defined, then the stack overflow problem may arise. Let us take an example to understand this.

```
int fact(int n)
{
   // wrong base case (it may cause    // stack overflow).
   if (n == 100)
      return 1;

   else
      return n*fact(n-1);
}
```

If fact(10) is called, it will call fact(9), fact(8), fact(7) and so on but the number will never reach 100. So, the base case is not reached. If the memory is exhausted by these functions on the stack, it will cause a stack overflow error.

Programs:

1)Number Factorial

The following example calculates the factorial of a given number using a recursive function −

```
#include <stdio.h>

unsigned long long int factorial(unsigned int i) {

   if(i <= 1) {
      return 1;
   }
   return i * factorial(i - 1);
}

int  main() {
   int i = 12;
   printf("Factorial of %d is %d\n", i, factorial(i));
   return 0;
}
```

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

```
Factorial of 12 is 479001600
```

2)Fibonacci Series

The following example generates the Fibonacci series for a given number using a recursive function −

```c
#include <stdio.h>

int fibonacci(int i) {

   if(i == 0) {
      return 0;
   }

   if(i == 1) {
      return 1;
   }
   return fibonacci(i-1) + fibonacci(i-2);
}

int  main() {

   int i;

   for (i = 0; i < 10; i++) {
      printf("%d\t\n", fibonacci(i));
   }

   return 0;
}
```

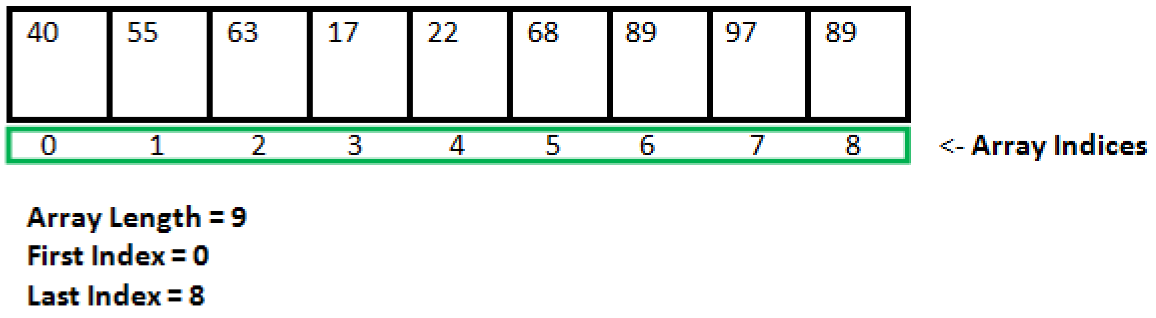
When the above code is compiled and executed, it produces the following result −

```
0
1
1
2
3
5
8
13
21
34
```

Dynamic Memory Allocation in C

Dynamic Memory Allocation in C using malloc(), calloc(), free() and realloc()

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.



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). For Example,

- If there is a situation where only 5 elements are needed to be entered in this 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.

- Take another situation. 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.

This procedure is referred to as **Dynamic Memory Allocation in C**.
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 provides some functions to achieve these tasks. There are 4 library functions provided by C defined under **<stdlib.h>** header file to facilitate dynamic memory allocation in C programming. They are:

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

Let’s look at each of them in greater detail.

1. C malloc() method

“malloc” or **“memory allocation”** method in C is used to dynamically allocate a single large block of memory with the specified size. It returns a pointer of type void which can be cast into a pointer of any form.

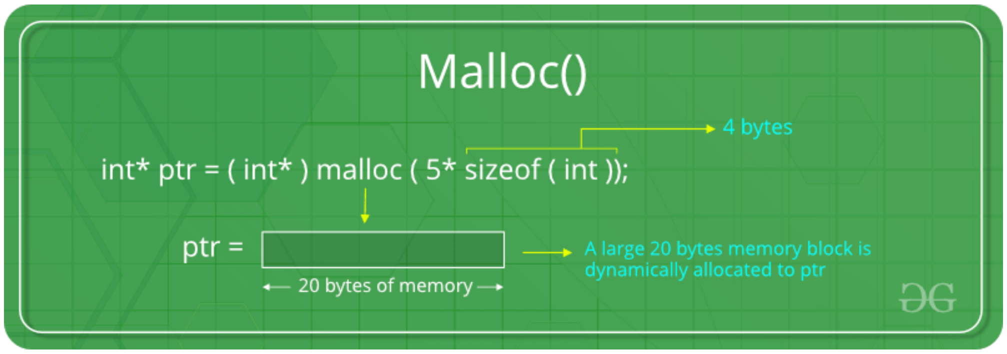
Syntax:

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

For Example:

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.


Malloc()
4 bytes
int* ptr = (int*) malloc (5* sizeof (int));
ptr =
20 bytes of memory
A large 20 bytes memory block is dynamically allocated to ptr

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

Example:

filter_none
edit
play_arrow
brightness_4

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

int main()
{

    // This pointer will hold the
    // base address of the block
created
    int* ptr;
    int n, i;

    // Get the number of elements for
the array
    n = 5;
    printf("Enter number of elements:
%d\n", n);

    // Dynamically allocate memory
using malloc()
    ptr = (int*)malloc(n *
sizeof(int));

    // Check if the memory has been
successfully
    // allocated by malloc or not
    if (ptr == NULL) {
        printf("Memory not
allocated.\n");
        exit(0);
    }
    else {

        // Memory has been successfully
allocated
        printf("Memory successfully
allocated using malloc.\n");

        // Get the elements of the
array
        for (i = 0; i < n; ++i) {
            ptr[i] = i + 1;
        }

        // Print the elements of the
array
        printf("The elements of the
array are: ");
        for (i = 0; i < n; ++i) {
            printf("%d, ", ptr[i]);
        }
    }

    return 0;
}
```
Output:
```
Enter number of elements: 5
Memory successfully allocated using malloc.
The elements of the array are: 1, 2, 3, 4, 5,
```

2. C calloc() method

“calloc” or **“contiguous allocation”** 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’.

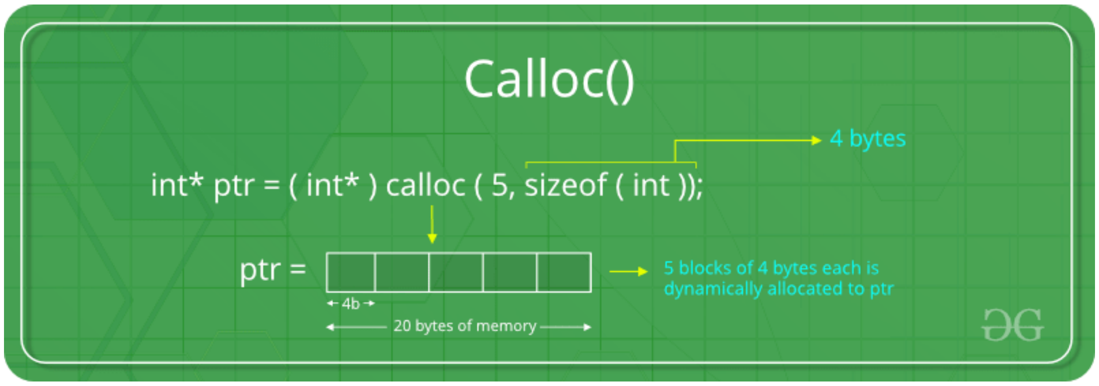
Syntax:

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

For Example:

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

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



Calloc()
4 bytes
int* ptr = (int*) calloc (5, sizeof (int));
ptr =
4b
20 bytes of memory
5 blocks of 4 bytes each is dynamically allocated to ptr

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

Example:

filter_none
edit
play_arrow
brightness_4

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

int main()
{

    // This pointer will hold the
    // base address of the block created
    int* ptr;
    int n, i;

    // Get the number of elements for the array
    n = 5;
    printf("Enter number of elements: %d\n", n);

    // Dynamically allocate memory using calloc()
    ptr = (int*)calloc(n, sizeof(int));

    // Check if the memory has been successfully
    // allocated by calloc or not
    if (ptr == NULL) {
        printf("Memory not allocated.\n");
        exit(0);
    }
    else {

        // Memory has been successfully allocated
        printf("Memory successfully allocated using calloc.\n");

        // Get the elements of the array
        for (i = 0; i < n; ++i) {
            ptr[i] = i + 1;
        }

        // Print the elements of the array
        printf("The elements of the array are: ");
        for (i = 0; i < n; ++i) {
            printf("%d, ", ptr[i]);
        }
    }

    return 0;
}
```

Output:

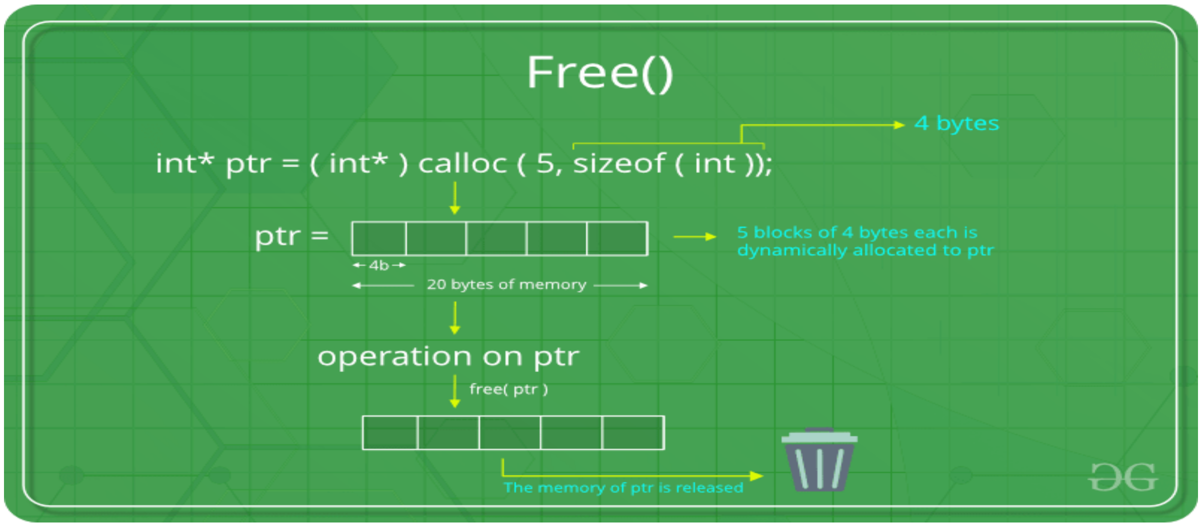
```
Enter number of elements: 5
Memory successfully allocated using calloc.
The elements of the array are: 1, 2, 3, 4, 5,
```

C free() method

“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);
```


Free()
4 bytes
int* ptr = (int*) calloc (5, sizeof (int));
ptr =
4b
20 bytes of memory
5 blocks of 4 bytes each is dynamically allocated to ptr
operation on ptr
free(ptr)
The memory of ptr is released

Example:

filter_none
edit
play_arrow
brightness_4

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

int main()
{

    // This pointer will hold the
    // base address of the block
created
    int *ptr, *ptr1;
    int n, i;

    // Get the number of elements for
the array
    n = 5;
    printf("Enter number of elements:
%d\n", n);

    // Dynamically allocate memory
using malloc()
    ptr = (int*)malloc(n *
sizeof(int));

    // Dynamically allocate memory
using calloc()
    ptr1 = (int*)calloc(n,
sizeof(int));

    // Check if the memory has been
successfully
    // allocated by malloc or not
    if (ptr == NULL || ptr1 == NULL) {
        printf("Memory not
allocated.\n");
        exit(0);
    }
    else {

        // Memory has been successfully
allocated
        printf("Memory successfully
allocated using malloc.\n");

        // Free the memory
        free(ptr);
        printf("Malloc Memory
successfully freed.\n");

        // Memory has been successfully
allocated
        printf("\nMemory successfully
allocated using calloc.\n");

        // Free the memory
        free(ptr1);
        printf("Calloc Memory
successfully freed.\n");
    }

    return 0;
}
```
Output:
```
Enter number of elements: 5
Memory successfully allocated using malloc.
Malloc Memory successfully freed.
```

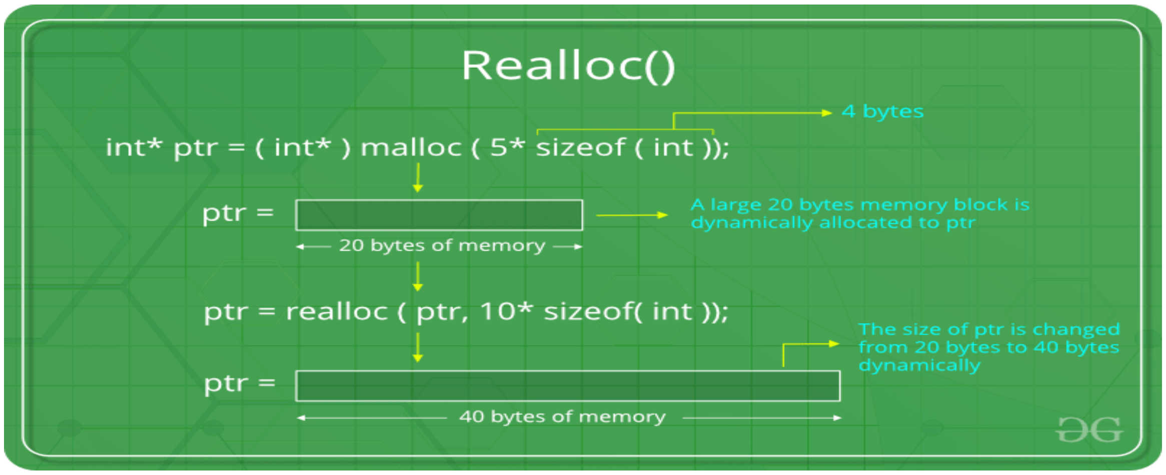
```
Memory successfully allocated using calloc.
Calloc Memory successfully freed.
```

3. C realloc() method

“realloc” or **“re-allocation”** method in C is used to dynamically change the memory allocation of a previously allocated memory. In other words, if the memory previously allocated with the help of malloc or calloc is insufficient, realloc can be used to **dynamically re-allocate memory**.

Syntax:

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


where ptr is reallocated with new size 'newSize'.
```


Realloc()
4 bytes
int* ptr = (int*) malloc (5* sizeof (int));
ptr =
A large 20 bytes memory block is dynamically allocated to ptr
20 bytes of memory
ptr = realloc (ptr, 10* sizeof(int));
The size of ptr is changed from 20 bytes to 40 bytes dynamically
ptr =
40 bytes of memory

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

Example:

filter_none
edit
play_arrow
brightness_4

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

int main()
{

    // This pointer will hold the
    // base address of the block created
    int* ptr;
    int n, i;

    // Get the number of elements for the array
    n = 5;
    printf("Enter number of elements: %d\n", n);

    // Dynamically allocate memory using calloc()
    ptr = (int*)calloc(n, sizeof(int));

    // Check if the memory has been successfully
    // allocated by malloc or not
    if (ptr == NULL) {
        printf("Memory not allocated.\n");
        exit(0);
    }
    else {

        // Memory has been successfully allocated
        printf("Memory successfully allocated using
calloc.\n");

        // Get the elements of the array
        for (i = 0; i < n; ++i) {
            ptr[i] = i + 1;
        }

        // Print the elements of the array
        printf("The elements of the array are: ");
        for (i = 0; i < n; ++i) {
            printf("%d, ", ptr[i]);
        }

        // Get the new size for the array
        n = 10;
        printf("\n\nEnter the new size of the array:
%d\n", n);
```

```
        // Dynamically re-allocate memory using
realloc()
        ptr = realloc(ptr, n * sizeof(int));

        // Memory has been successfully allocated
        printf("Memory successfully re-allocated
using realloc.\n");

        // Get the new elements of the array
        for (i = 5; i < n; ++i) {
            ptr[i] = i + 1;
        }

        // Print the elements of the array
        printf("The elements of the array are: ");
        for (i = 0; i < n; ++i) {
            printf("%d, ", ptr[i]);
        }

        free(ptr);
    }

    return 0;
}
```

Output:

```
Enter number of elements: 5
Memory successfully allocated using calloc.
The elements of the array are: 1, 2, 3, 4, 5,

Enter the new size of the array: 10
Memory successfully re-allocated using realloc.
The elements of the array are: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
```

FUNCTIONS

Definition : A function is self contained block of statements that performs a particular task.

C -functions are divided into two parts

1) Pre-defined functions(library functions)
2) User-defined functions

library functions OR pre-defined functions are not required to be written by us where as user define function has to be developed by the user at the time of writing program. Ex: printf(), scanf(), sqrt(), pow() etc.,,,

To make use of user defined function, we need to write three elements that are related to function.

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

Function declaration / signature / prototype

Just like variables , functions are also to be declared before we use them in the program. A function declaration consists of four parts.

1. Function type(return type)
2. Function name
3. Parameter List
4. Terminating semicolon.

The general format of declaring a function is :

return_type function_name(parameter list);

```
Example: void display( );
           int square(int m);
          int mul(int m, int n);
```

A prototype declaration may be placed in two places in a program

1) Above all the functions (including main)
2) Inside a function definition.

- When we place the declaration above all the functions (in the global declaration section), the prototype is referred to as a global prototype. Such declarations are available for all the functions in the program.
- When we place it in a function definition (in the local declaration section), this prototype is called local prototype.

Function call : is done inside main(). Whenever function call is made control from main() will be transfered to the definition of that function. After executing all the statements in tha function control comes back to main().

General form of function call : function_name( ) ;

If arguments are being passed then : **function_name(arguments);**

Function definition : (also known as FUNCTION IMPLEMENTATION)

Format of function definition is :-

```
return_type  fuction_name(arguments)
     {
              Local variable declarations;
              stmt 1;
              stmt 2;
              – – – – – –
              – – – – – – – – .
               return statement;
       }
```

return type **specifies the type of value that the function is going to return. Various return types are ………void, int, char, float, double, long int etc.**

Function name is the name of function. It is given the user.

- A FUNCTION IN WHICH FUNCTION CALL IS MADE IS KNOWN AS CALLING FUNCTION.
- A FUNCION WHICH IS CALLED BY OTHER FUNCTION IS KNOWN AS CALLED FUNCTION.

Example :

```
main ()
{
  printf("hello");
  display();
}
```

In above sample code **display( ) is called function** and **main( ) is calling function.**

main( ) is a specially recognized user-defined function. That is ,

- Declaration of main( ) is pre-defined
- Definition of main( ) is user-defined

Arguments :

The mechanism of passing values between calling function and called function is known as ARGUMENT or PARAMETER.

There are 2 types of arguments or parameters :

1. Actual arguments
2. Formal arguments

- Variables in function call are actual arguments.
- Variables in function definition are formal arguments.

The return statement can take one of the following forms

```
return;
   or
return(expression);
```

The first one does not return any value; it acts much as closing brace of the function. When a return is encountered, the control is immediately passed back to the calling function.

An example of the use of a simple return is as follows

```
if(error)
return;
```

The second form of the return with an expression returns the value of the expression. For example the function

```
int mul(int x,int y)
{
      int p;
      p=x*y;
      return(p);
}
```

returns the value of the p which is the product of the values of x and y. The last two statements can be combined into one statement as follows.

```
return(x*y);
```

A function may have more than one return statement this situation arises when the value returned is based on certain conditions. For example

```
if(x<=0)
  return(0);
  else
   return(1);
```

TYPES OF FUNCTIONS BASED ON ARGUMENTS AND RETURN VALUES

1. Functions without arguments and without return values
2. Functions with arguments and without return values
3. Functions with arguments and with return values
4. Functions without arguments and with return values

Functions without arguments and without return values

- No values will be passed from calling function to called function.
- No values will be returned to calling function from called function.

Example program :

```
#include<stdio.h>
void add( ); // function declaration
int main() // calling function
{
  add(); // function call
}

void add( ) // function definition and called function
{
 int x,y,z;
 printf("enter any two integers");
 scanf("%d %d", &x,&y);
 z=x+y;
 printf("sum = %d",z);
}
```

Example 2:

```
// no return and no arguments
#include <stdio.h>
void sum(void);
void sum()
{
    int n,sum=0,i;
    printf("enter the range");
    scanf("%d",&n);
    for(i=1;i<=n;i++)
    {
        sum=sum+i;
    }
    printf("sum is %d",sum);
}
int main()
{
sum();
 }
```

Functions with arguments and without return values

- (arguments) values will be passed from calling function to called function.
- No values will be returned to calling function from called function.

Example program :

```
#include<stdio.h>
void add( int a, int b ); // function declaration ; 'a' and 'b' are formal arguments
int main() // calling function
{
  int x,y ;
  printf("enter any two integers");
  scanf("%d %d", &x,&y);
  add(x,y); // function call ; 'x' and 'y' are actual arguments
}
void add( int a, int b) // function definition and called function
{
 int  c;
 c = a + b;
 printf("sum = %d",c );
}
```

Example 2:

```
// no return and with arguments
#include<stdio.h>
void sum(int);
void sum(int n1)
{
    int sum=0,i;
    for(i=1;i<=n1;i++)
    {
        sum=sum+i;
    }
    printf("sum is %d",sum);
}
int main()
{
    int n;
printf("enter the range");
scanf("%d",&n);
sum(n);
}
```

Functions with arguments and with return values

- (arguments) values will be passed from calling function to called function.
- values will be returned to calling function from called function.

Example program :

```
#include<stdio.h>
int add(int a, int b ); // function declaration ; 'a' and 'b' are formal arguments
int main() // calling function
{
  int x,y,z ;
  printf("enter any two integers");
  scanf("%d %d", &x,&y);
  z = add(x,y); // function call ; 'x' and 'y' are actual arguments
  printf("sum = %d",z);
}
int add( int x, int y) // function definition and called function
{
 int c;
 c = x+y;
 return c;
}
```

Example 2:

```
// with return and with arguments
#include<stdio.h>
int sum(int);
int sum(int n1)
{
    int sum=0,i;

    for(i=1;i<=n1;i++)
    {
        sum=sum+i;
    }
    return(sum);
}
int main()
{
    int n,s;
printf("enter the range");
scanf("%d",&n);
s=sum(n);
printf("sum is %d",s);
}
```

Functions without arguments and with return values

- No values will be passed from calling function to called function.
- values will be returned to calling function from called function.

Example program :

```
  #include<stdio.h>
  int add( ); // function declaration
  int main() // calling function
  {
  int c;
  c = add( ); // function call
  printf("sum = %d",c);
  }
int add( ) // function definition and called function

{
  int x,y,z ;
  printf("enter any two integers");
  scanf("%d %d", &x,&y);
  z = x+y;
  return z;
}
```

Example 2:

```
#include <stdio.h>
int sum(void);
// with return and no arguments
int sum()
{
    int n,sum=0,i;
    printf("enter the range");
    scanf("%d",&n);
    for(i=1;i<=n;i++)
    {
        sum=sum+i;
    }
    return(sum);
}
int main()
{
int s;
s=sum();
printf("sum is %d",s);
 }
```

SCOPE OF A VARIABLE : **is the extent to which it can be used in a program.**

Based on this, variables are classified as LOCAL VARIABLES & GLOBAL VARIABLES.

- Local variable is restricted to a particular block in the program.
- Global variable is used in the entire program that is in all the functions present in the program.

Example :

```
#include<stdio.h>
int k=20; // global variable
void add ( );
main()

{

  int c =30; // 'c' is local variable to main( )
  add ( );
  printf("sum = %d", ( c+k ) ); // output is 50

}

void add( )

{

  int h = 40; // 'h' is local variable to add( )
  printf(" addition = %d", ( h+k ) ); // output is 60
}
```

Parameter Passing Techniques in Functions

When a function gets executed in the program, the execution control is transferred from calling-function to called function and executes function definition, and finally comes back to the calling function. When the execution control is transferred from calling-function to called-function it may carry one or number of data values. These data values are called as **parameters**.

Parameters are the data values that are passed from calling function to called function.

In C, there are two types of parameters and they are as follows...

- Actual Parameters
- Formal Parameters

The actual parameters are the parameters that are speficified in calling function. The formal parameters are the parameters that are declared at called function. When a function gets executed, the copy of actual parameter values are copied into formal parameters.

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

- Call by Value
- Call by Reference

Call by Value

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 consider the following program...

```
//call by value

#include<stdio.h>
void swap(int,int);
void main()
{
   int a,b;
   printf("enter a and b value\n");
   scanf("%d %d",&a,&b);
   swap(a,b);
   printf("\n after swap a=%d b=%d",a,b);
```

```
 }
void swap(int x,int y)
{
    int temp;
    temp =x;
    x=y;
    y= temp;
    printf("\n a=%d b=%d",x,y);
}
```

Call 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.

That means in call by reference parameter passing method, the address of the actual parameters is passed to the called function and is recieved 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 consider the following program...

```
// call by reference
#include<stdio.h>
void swap(int*,int*);
void main()
{
    int a,b;
    printf("enter a and b value\n");
    scanf("%d %d",&a,&b);
    swap(&a,&b);
    printf("\n after swap a=%d b=%d",a,b);
}
void swap(int *a,int *b)
{
    int temp;
    temp =*a;
    *a=*b;
    *b= temp;
    printf("\n after swap a=%d b=%d",a,b);
}
```

Recursion

It is a modular programming technique.

A function calling itself is known as recursive function i.e, function call statement for calling the same function appears inside the function.

In recursion, the same function acts as calling function and called function.

Ex:

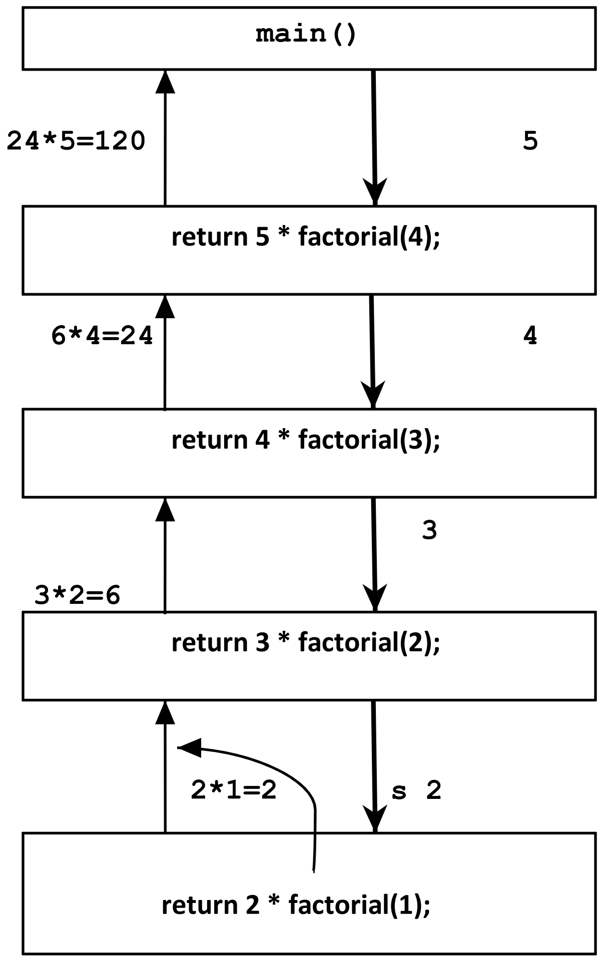
```
int main()
{
        printf("here main() is a recursion function");
         main();
}
```

There must be an exclusive terminating condition or else the execution will continue indefinitely.

Program to find factorial of number using recursion

```
#include<stdio.h>

int factorial( int k);

int main()

{

 int s,n;

 printf("enter a no for printing factorials from 1 to

 entered no. ");

 scanf("%d",&n);

 s = factorial(n);

 printf(“ factorial = %d”, s);

}

 int factorial (int k )

{

 if ( k==1)

 return 1;

 else

 return k * factorial(k-1);

}
```

Consider   k = 5



In the above program :-

- main() is calling function for factorial (5)
- factorial (5) is calling function for factorial(4)
- factorial(4) is calling function for factorial(3)
- factorial(3) is calling function for factorial(2)
- factorial(2) is calling function for factorial(1)

so, called function will return the value to its calling function.

Difference between recursion and iteration:

Both recursion and iteration are based on control structures.

- Iteration uses repetition structures to achieve looping and recursion uses selection structure to make repeated function calls.
- Both iteration and recursion use a condition to terminate. Iteration terminates when condition in loop fails. Recursion terminates when condition in 'if' statement becomes TRUE.
- Recursion helps us to solve a complex problem by breaking the program into smaller problems, that are similar to original problem. But each recursive call creates another copy of function in memory which consumes more and more memory. Thus, recursion should be applied only to larger programs.

Program to find " x power y " using recursion

```
 #include<stdio.h>
 int power(int,int);
 int main()
 {
        int b,e,res;
        printf("Enter the value of base
        and exponent");
        scanf("%d%d",&b,&e);
        res = power(b,e);
        printf("value of %d power %d is %d",b,e,res);
        }
int power(int b, int e)
 {
        if(e==0)
               return 1;
        else if(e==1)
              return  b;
              else
              return(b*power(b,e-1));
}
```

Program to find fibonacii series using recursion

```
#include<stdio.h>
int main()
{
        int i,n;
        printf("Enter the n value:");
        scanf("%d",&n);
        printf("the fibinacci series is\n");
        for(i=0;i<n;i++)
        {
        printf("%d\n",fib(i));
        }
}
int fib(int n)
{
        if(n==0)
        {
        return(0);
        }
        if(n==1)
        {
        return(1);
        }
        else
        {
        return(fib(n-1)+fib(n-2));
        }
}
```

Dynamic Memory Allocation

In C programming language, when we declare variables memory is allocated in space called stack. The memory allocated in stack is fixed at the time of compilation and remains till end the program execution.

When we create an array, we must specify the size at the time of declaration itself and it cannot be changed during the program execution. This is a major problem when we do not know the number of values to be stored in an array.

To solve this we use the concept of Dynamic Memory Allocation. The dynamic memory allocation allocates memory from heap storage. Dynamic memory allocation is defined as follow...

Allocation of memory during the program execution is called dynamic memory allocation.

Or

Dynamic memory allocation is the process of allocating the memory manually at the time of program execution

An array is a collection of fixed number of values of a single type. That is, you need to declare the size of an array before you can use it.
Sometimes, the size of 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.
There are 4 library functions defined under <stdlib.h> makes dynamic memory allocation in C programming. They are malloc(), calloc(), realloc() and free().

We use pre-defined or standard library functions to allocate memory dynamically. There are FOUR standard library functions that are defined in the header file known as "stdlib.h". They are as follows...

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

1.malloc()

malloc() is the standard library function used to allocate a memory block of specified number of bytes and returns void pointer. The void pointer can be casted to any datatype. If malloc() function unable to allocate memory due to any reason it returns NULL pointer.

Syntax of malloc()

void* malloc(size**_in_bytes)**

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

Example:

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

Considering the size of int is 4 bytes, this statement allocates 400 bytes of memory. And, the pointer ptr holds the address of the first byte in the allocated memory.

However, if the space is insufficient, allocation fails and returns a NULL pointer.

2.free()

Dynamically allocated memory created with either calloc() or malloc() doesn't get freed on their own. You must explicitly use free() to release the space.

Syntax of free()

```
free(ptr);
```

This statement frees the space allocated in the memory pointed by ptr.

Example Programs for malloc() and free().

This program calculates the sum of n numbers entered by the user. To perform this task, memory is dynamically allocated using malloc(), and memory is freed using free() function.

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int n, i, *p;
    printf("Enter number of elements: ");
    scanf("%d", &n);
    p = (int*) malloc(n * sizeof(int));
    if(p == NULL)
    {
        printf("Error! memory not allocated.");
        exit(0);
    }
    printf("Enter elements: ");
    for(i = 0; i < n; ++i)
    {
        p[i]=i+1;
    }
    printf("The value assigned are \n");
       for(i = 0; i < n; ++i)
       printf("%d", *(p + i));
       free(p);
       return 0;
}
```

3. calloc()

calloc() is the standard library function used to allocate multiple memory blocks of specified number of bytes and initializes them to ZERO. calloc() function returns void pointer. If calloc() function unable to allocate memory due to any reason it returns NULL pointer. Generelly calloc() is used to allocate memory for array and structure. calloc() function takes two arguments and they are

1. Number of blocks to allocate,
2. Size of each block in bytes

C calloc()

The name "calloc" stands for contiguous allocation.

The malloc() function allocates a single block of memory. Whereas, calloc() allocates multiple blocks of memory and initializes them to zero.

Syntax of calloc()

```
void* calloc(number_of_blocks,size_of_each_block_in_bytes)
```

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

Example:

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

Example Program for *calloc()*.

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int n, i, *p;
    printf("Enter number of elements: ");
    scanf("%d", &n);
    p = (int*) calloc(n , sizeof(int));
    if(p == NULL)
    {
        printf("Error! memory not allocated.");
        exit(0);
    }
    printf("Enter elements: ");
    for(i = 0; i < n; ++i)
    {
        p[i]=i+1;
    }
    printf("The value assigned are \n");
       for(i = 0; i < n; ++i)
       printf("%d", *(p + i));
       free(p);
       return 0;
}
```

4. realloc()

realloc() is the standard library function used to modify the size of memory blocks that were previously allocated using malloc() or calloc(). realloc() function returns void pointer. If calloc() function unable to allocate memory due to any reason it returns NULL pointer.

If the dynamically allocated memory is insufficient or more than required, you can change the size of previously allocated memory using realloc() function

void* realloc(*pointer, new_size_of_each_block_in_bytes)

Syntax of realloc()

```
ptr = realloc(ptr, x);
```

Here, `ptr` is reallocated with new size `x`.

Example

Example Program for *realloc()*.

```
#include <stdio.h>
#include<stdlib.h>
#include <string.h>
int main()
{
    char *p1;
    int m1, m2;
    m1 = 15;
    m2 = 20;
    p1 = (char*)malloc(m1);
    strcpy(p1, "Mahatma Gandhi ");
    p1 = (char*)realloc(p1, m2);
    strcat(p1, "Institute of Technology");
    printf("%s\n", p1);
    free(p1);
    return 0;
}
```

*****