functions in C

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

To perform any task, we can create function. A function can be called many times. It provides *modularity* and code *reusability*.

Advantage of functions in C

There are many advantages of functions.

1) Code Reusability

By creating functions in C, you can call it many times. So we don't need to write the same code again and again.

2) Code optimization

It makes the code optimized, we don't need to write much code.

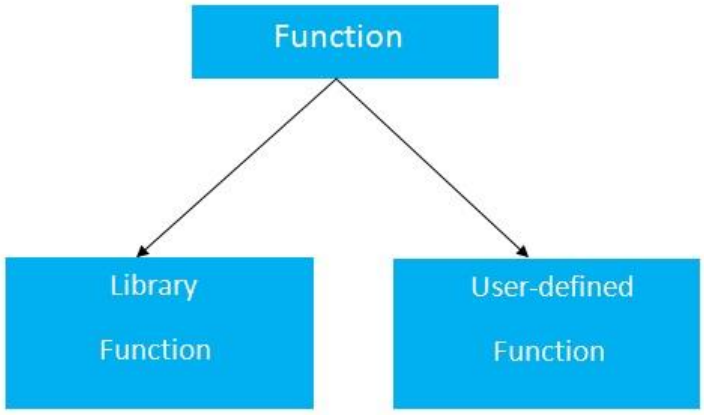
Suppose, you have to check 3 numbers (781, 883 and 531) whether it is prime number or not. Without using function, you need to write the prime number logic 3 times. So, there is repetition of code.

But if you use functions, you need to write the logic only once and you can reuse it several times.

Types of Functions

There are two types of functions in C programming:

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



Declaration of a function

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

```
return_type function_name(data_type parameter...){
//code to be executed
}
```

Return Value

A C function may or may not return a value from the function. If you don't have to return any value from the function, use void for the return type.

Let's see a simple example of C function that doesn't return any value from the function.

Example without return value:

```
void hello(){
printf("hello c");
}
```

If you want to return any value from the function, you need to use any data type such as int, long, char etc. The return type depends on the value to be returned from the function.

Let's see a simple example of C function that returns int value from the function.

Example with return value:

```
int get(){
return 10;
}
```

In the above example, we have to return 10 as a value, so the return type is int. If you want to return floating-point value (e.g. 10.2, 3.1, 54.5 etc), you need to use float as the return type of the method.

```
float get(){
return 10.2;
}
```

Now, you need to call the function, to get the value of the function.

Parameters in C Function

A c function may have 0 or more parameters. You can have any type of parameter in C program such as int, float, char etc. The parameters are also known as **formal arguments**.

Example of a function that has 0 parameter:

```
void hello(){
printf("hello c");
}
```

Example of a function that has 1 parameter:

```
int cube(int n){
return n*n*n;
}
```

Example of a function that has 2 parameters:

```
int add(int a, int b){
return a+b;
}
```

Calling a function in C

If a function returns any value, you need to call function to get the value returned from the function. The syntax of calling a function in c programming is given below:

1. variable=function_name(arguments...);

1) variable: The variable is not mandatory. If function return type is *void*, you must not provide the variable because void functions doesn't return any value.

2) function_name: The function_name is name of the function to be called.

3) arguments: You need to provide arguments while calling the C function. It is also known as **actual arguments**.

Example to call a function:

1. hello();//calls function that doesn't return a value
2. **int** value=get();//calls function that returns value
3. **int** value2=add(10,20);//calls parameterized function by passing 2 values

Example of C function with no return statement

Let's see the simple program of C function that doesn't return any value from the function.

```
#include<stdio.h>
void hello(){
printf("hello c programming \n");
}
int main(){
hello();//calling a function
hello();
hello();
return 0;
}
```

Output

```
hello c programming
hello c programming
hello c programming
```

Example of C function with return statement

Let's see the simple program of function in c language.

```
#include<stdio.h>
//defining function
int cube(int n){
return n*n*n;
}
int main(){
int result1=0,result2=0;
```

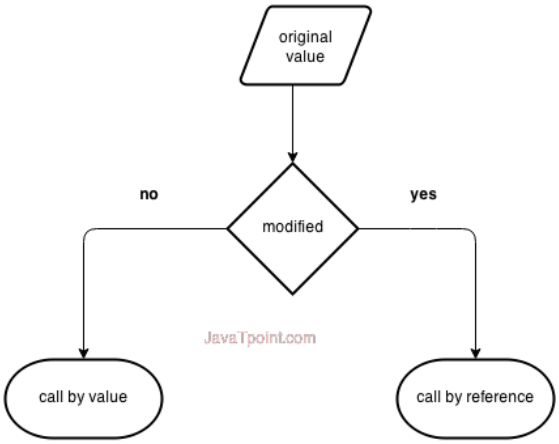
```
result1=cube(2);//calling function
result2=cube(3);

printf("%d \n",result1);
printf("%d \n",result2);
return 0;
}
```

Output

```
8
27
```

Call by value and call by reference in C

There are two ways to pass value or data to function in C language: *call by value* and *call by reference*. Original value is not modified in call by value but it is modified in call by reference.



Let's understand call by value and call by reference in c language one by one.

Call by value in C

In call by value, **original value is not modified**.

In call by value, value being passed to the function is locally stored by the function parameter in stack memory location. If you change the value of function parameter, it is changed for the current function only. It will not change the value of variable inside the caller method such as main().

Let's try to understand the concept of call by value in c language by the example given below:

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

```
void change(int num) {
    printf("Before adding value inside function num=%d \n",num);
    num=num+100;
    printf("After adding value inside function num=%d \n", num);
}
int main() {
    int x=100;
    printf("Before function call x=%d \n", x);
    change(x);//passing value in function
    printf("After function call x=%d \n", x);
return 0;
}
```

Output

```
Before function call x=100
Before adding value inside function num=100
After adding value inside function num=200
After function call x=100
```

Call by reference in C

In call by reference, **original value is modified** because we pass reference (address).

Here, address of the value is passed in the function, so actual and formal arguments shares the same address space. Hence, value changed inside the function, is reflected inside as well as outside the function.

Note: To understand the call by reference, you must have the basic knowledge of pointers.

Let's try to understand the concept of call by reference in c language by the example given below:

```
#include<stdio.h>
void change(int *num) {
    printf("Before adding value inside function num=%d \n",*num);
    (*num) += 100;
    printf("After adding value inside function num=%d \n", *num);
}
int main() {
    int x=100;
    printf("Before function call x=%d \n", x);
    change(&x);//passing reference in function
    printf("After function call x=%d \n", x);
```

```
    return 0;
    }
```

Output

```
Before function call x=100
Before adding value inside function num=100
After adding value inside function num=200
After function call x=200
```

Difference between call by value and call by reference in c

No.	Call by value	Call by reference
1	A copy of value is passed to the function	An address of value is passed to the function
2	Changes made inside the function is not reflected on other functions	Changes made inside the function is reflected outside the function also
3	Actual and formal arguments will be created in different memory location	Actual and formal arguments will be created in same memory location

Recursive Functions in C

In C programming language, function calling can be made from main() function, other functions or from same function itself. The recursive function is definedd as follows...

A function called by itself is called recursive function.

The recursive functions should be used very carefully because, when a function called by itself it enters into infinite loop. And when a function enters into the infinite loop, the function execution never gets completed. We should define the condition to exit from the function call so that the recursive function gets terminated.

When a function is called by itself, the first call remains under execution till the last call gets invoked. Every time when a function call is invoked, the function returns the execution control to the previous function call.

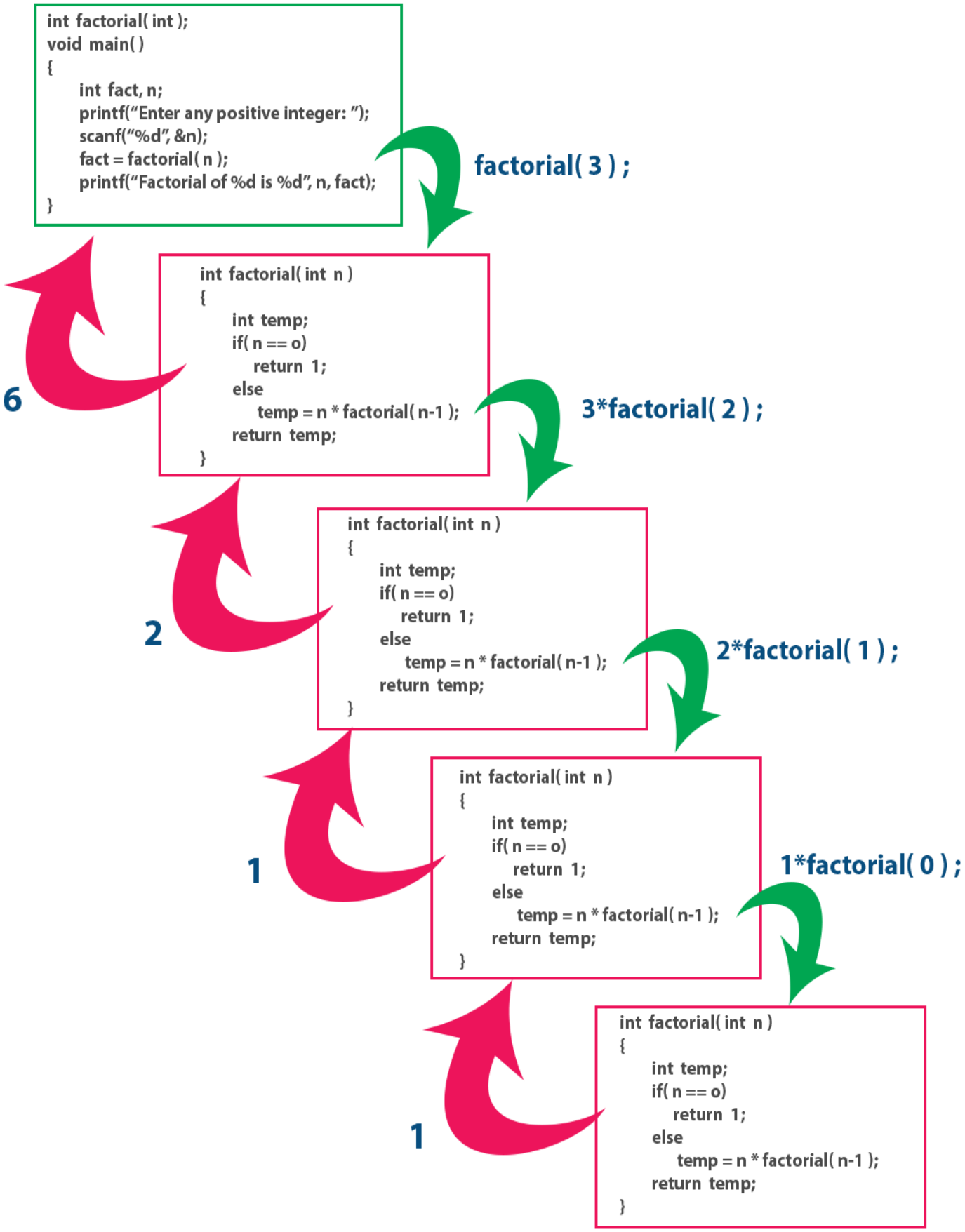
```
#include <stdio.h>
#include<conio.h>
int  factorial( int ) ;
void  main( )
{
```

```
        int  fact, n ;
        printf("Enter any positive integer: ") ;
        scanf("%d", &n) ;
        fact = factorial( n ) ;
        printf("Factorial of %d is %d", n, fact) ;
}
int  factorial( int  n )
{
        int  temp ;
        if( n == 0)
            return  1 ;
        else
             temp = n * factorial( n-1 ) ; // recursive function call
        return  temp ;
}
```

Output:

Enter any positive integer: 3
Factorial of 3 is 6

In the above example program, the **factorial()** function call is initiated from main() function with the value 3. Inside the factorial() function, the function calls factorial(2), factorial(1) and factorial(0) are called recursively. In this program execution process, the function call factorial(3) remains under execution till the execution of function calls factorial(2), factorial(1) and factorial(0) gets completed. Similarly the function call factorial(2) remains under execution till the execution of function calls factorial(1) and factorial(0) gets completed. In the same way the function call factorial(1) remains under execution till the execution of function call factorial(0) gets completed. The complete execution process of the above program is shown in the following figure...


int factorial(int);
void main( )
{
int fact, n;
printf("Enter any positive integer: ");
scanf("%d", &n);
fact = factorial(n);
printf("Factorial of %d is %d", n, fact);
}
factorial(3) ;
int factorial(int n)
{
int temp;
if(n == o)
return 1;
else
temp = n * factorial(n-1);
return temp;
}
6
3*factorial(2) ;
int factorial(int n)
{
int temp;
if(n == o)
return 1;
else
temp = n * factorial(n-1);
return temp;
}
2
2*factorial(1) ;
int factorial(int n)
{
int temp;
if(n == o)
return 1;
else
temp = n * factorial(n-1);
return temp;
}
1
1*factorial(0) ;
int factorial(int n)
{
int temp;
if(n == o)
return 1;
else
temp = n * factorial(n-1);
return temp;
}
1

Fibonacci series using recursive function

```
#include<stdio.h>

int Fibonacci(int);

void main()
{
   int n, i = 0, c;

printf(“enter n value:n”);
   scanf("%d",&n);

   printf("Fibonacci series\n");

   for ( c = 1 ; c <= n ; c++ )
   {
      printf("%d\n", Fibonacci(i));
      i++;
   }

   return 0;
}

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

GCD using normal function:

```
#include <stdio.h>

#include<conio.h>

void gcd(int n1,int n2);

void main()

{

   int n1, n2;

   clrscr();

   printf("Enter two integers: ");

   scanf("%d %d", &n1, &n2);
```

```
   gcd(n1,n2);
   getch();
}
void gcd(int n1,int n2)
{
int gcd,i;
for(i=1; i <= n1 && i <= n2; ++i)
   {
      // Checks if i is factor of both integers
      if(n1%i==0 && n2%i==0)
         gcd = i;
   }
   printf("G.C.D of %d and %d is %d", n1, n2, gcd);
}
```

GCD using Recursion:

```
#include <stdio.h>
#include<conio.h>
int gcd(int n1, int n2);
void main()
{
  int n1, n2;
  clrscr();
  printf("Enter two positive integers: ");
  scanf("%d %d", &n1, &n2);
  printf("G.C.D of %d and %d is %d.", n1, n2, gcd(n1,n2));
  getch();
}
int gcd(int n1, int n2)
{
```

```
   if (n1!=0 && n2 != 0)
      return gcd(n2, n1%n2);
   else
      return n1;
}
```

Fibonacci series using Recursive function:

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

Limitations of Recursive function:

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

C – Dynamic memory allocation

DYNAMIC MEMORY ALLOCATION IN C:
The process of allocating memory during program execution is called dynamic memory allocation.

DYNAMIC MEMORY ALLOCATION FUNCTIONS IN C:
C language offers 4 dynamic memory allocation functions. They are,

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

Function	Syntax
malloc ()	malloc (number *sizeof(int));
calloc ()	calloc (number, sizeof(int));
realloc ()	realloc (pointer_name, number * sizeof(int));
free ()	free (pointer_name);

1. MALLOC() FUNCTION IN C:

- malloc () function is used to allocate space in memory during the execution of the program.
- malloc () does not initialize the memory allocated during execution. It carries garbage value.

- malloc () function returns null pointer if it couldn't able to allocate requested amount of memory.

EXAMPLE PROGRAM FOR MALLOC() FUNCTION IN C:

```
1  #include <stdio.h>
2  #include <string.h>
3  #include <stdlib.h>
4
5  int main()
6  {
7      char *mem_allocation;
8      /* memory is allocated dynamically */
9      mem_allocation = malloc( 20 * sizeof(char) );
10     if( mem_allocation== NULL )
11     {
12       printf("Couldn't able to allocate requested memory\n");
13     }
14     else
15     {
16       strcpy( mem_allocation,"fresh2refresh.com");
17     }
18     printf("Dynamically allocated memory content : " \
19           "%s\n", mem_allocation );
20     free(mem_allocation);
21 }
```

OUTPUT:

```
Dynamically allocated memory content : fresh2refresh.com
```

2. CALLOC() FUNCTION IN C:

- calloc () function is also like malloc () function. But calloc () initializes the allocated memory to zero. But, malloc() doesn't.

EXAMPLE PROGRAM FOR CALLOC() FUNCTION IN C:

```
1  #include <stdio.h>
2  #include <string.h>
3  #include <stdlib.h>
4
5  int main()
6  {
7      char *mem_allocation;
8      /* memory is allocated dynamically */
9      mem_allocation = calloc( 20, sizeof(char) );
10     if( mem_allocation== NULL )
```

```
11    {
12       printf("Couldn't able to allocate requested memory\n");
13    }
14    else
15    {
16       strcpy( mem_allocation,"fresh2refresh.com");
17    }
18       printf("Dynamically allocated memory content   : " \
19            "%s\n", mem_allocation );
20       free(mem_allocation);
21 }
```

OUTPUT:

```
Dynamically allocated memory content : fresh2refresh.com
```

3. REALLOC() FUNCTION IN C:

- realloc () function modifies the allocated memory size by malloc () and calloc () functions to new size.
- If enough space doesn't exist in memory of current block to extend, new block is allocated for the full size of reallocation, then copies the existing data to new block and then frees the old block.

4. FREE() FUNCTION IN C:

- free () function frees the allocated memory by malloc (), calloc (), realloc () functions and returns the memory to the system.

EXAMPLE PROGRAM FOR REALLOC() AND FREE() FUNCTIONS IN C:

```
1  #include <stdio.h>
2  #include <string.h>
3  #include <stdlib.h>
4
5  int main()
6  {
7     char *mem_allocation;
8     /* memory is allocated dynamically */
9     mem_allocation = malloc( 20 * sizeof(char) );
10    if( mem_allocation == NULL )
11    {
12       printf("Couldn't able to allocate requested memory\n");
13    }
14    else
15    {
```

```
16      strcpy( mem_allocation,"fresh2refresh.com");
17    }
18    printf("Dynamically allocated memory content  : " \
19          "%s\n", mem_allocation );
20    mem_allocation=realloc(mem_allocation,100*sizeof(char));
21    if( mem_allocation == NULL )
22    {
23       printf("Couldn't able to allocate requested memory\n");
24    }
25    else
26    {
27       strcpy( mem_allocation,"space is extended upto " \
28                         "100 characters");
29    }
30    printf("Resized memory : %s\n", mem_allocation );
31    free(mem_allocation);
32 }
```

OUTPUT:

```
Dynamically allocated memory content : fresh2refresh.com
Resized memory : space is extended upto 100 characters
```

DIFFERENCE BETWEEN STATIC MEMORY ALLOCATION AND DYNAMIC MEMORY ALLOCATION IN C:

Static memory allocation	Dynamic memory allocation
In static memory allocation, memory is allocated while writing the C program. Actually, user requested memory will be allocated at compile time.	In dynamic memory allocation, memory is allocated while executing the program. That means at run time.
Memory size can't be modified while execution. Example: array	Memory size can be modified while execution. Example: Linked list

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

malloc()	calloc()
It allocates only single block of requested memory	It allocates multiple blocks of requested memory

int *ptr;ptr = malloc(20 * sizeof(int));For the above, 20*4 bytes of memory only allocated in one block. Total = 80 bytes	int *ptr;Ptr = calloc(20, 20 * sizeof(int));For the above, 20 blocks of memory will be created and each contains 20*4 bytes of memory. Total = 1600 bytes
malloc () doesn't initializes 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;ptr = (int*)malloc(sizeof(int)*20);	Same as malloc () function int *ptr;ptr = (int*)calloc(20, 20 * sizeof(int));