Arrays

- Array is a collection of same type elements under the same identifier referenced by index number.
- Arrays are widely used within programming for different purposes such as sorting, searching and etc.
- Arrays are of two types single dimension array and multi-dimension array.
- Each of this array type can be of either static array or dynamic array.
- Static arrays have their sizes declared from the start and the size cannot be changed after declaration.
- Dynamic arrays that allow you to dynamically change their size at runtime, but they require more advanced techniques such as pointers and memory allocation.

Declaration of an Array

Arrays must be declared before they can be used in the program. Standard array declaration is as,

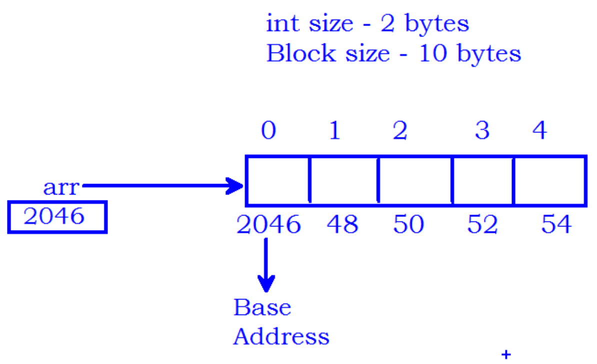
Syntax:

```
datatype identity [size];
```

For example

```
int arr[5];
```

- Here **int** specifies the type of the variable and the word **arr** specifies the name of the variable.
- The number **5** tells the number of elements of type int that will be in the array and is called the dimension of the array.
- Array elements get consecutive memory locations.
- Array variable stores base address of memory block
- Array elements can be processed using index
- Array index starts with 0 to size-1



Note: Address of the first element is called the base address and it is also the start address of an array.
The address of the **ith** element is given by the following formula,

$$\textbf{Address}_i = \textbf{base\_address} + \textbf{i*size\_of\_element}$$

Initializing Arrays

- Initializing of array is very simple in c programming.
- The initializing values are enclosed within the curly braces in the declaration and placed following an equal sign after the array name.
- Here is an example which declares and initializes an array of five elements of type int.
  int arr [5] = {1, 2, 3, 4, 5};

Note: The elements field within brackets [] which represents the number of elements the array is going to hold, must be a **constant** value, since arrays are blocks of non-dynamic memory whose size must be determined before execution. In order to create arrays with a variable length dynamic memory is needed, which is explained in dynamic memory allocation chapter.

Array can also be initialized after declaration.

```
int arr[4];
arr[0]=14;
arr[1]=13;
arr[2]=15;
arr[3]=16;
```

Accessing elements of an array:

- Once an array is declared, the individual elements in the array can be referred with subscript, the number in the brackets following the array name.
- This number specifies the element's position in the array.
- All the array elements are numbered, starting with 0. Thus arr[2] is not the second element of the array but the third.
- In the above declaration, we use the variable **i** as a subscript to refer to various elements of the array.
- This variable can take different values and hence can refer to the different elements in the array.
- This ability to use variables as subscripts marks arrays very useful.

If the array variable is local and int type, all the elements initializes with garbage values as follows:

```
#include<stdio.h>
int main()
{
        int arr[5],i;
        printf("Elements are :\n");
```

```
	for(i=0 ; i<5 ; i++)
	{
		printf("%d\n", arr[i]);
	}
}
```

If the variable is Global, all the elements initializes with default values as follows:

```
#include<stdio.h>
float arr[5];
int main()
{
	int i;
	printf("Elements are :\n");
	for(i=0 ; i<5 ; i++)
	{
		printf("%f\n", arr[i]);
	}
}
```

-> We can assign values directly to array variable.
-> If we assign at least one value, other elements initializes with default values.

```
#include<stdio.h>
int main()
{
	int arr[5]={10,20};
	int i;
	printf("Elements are :\n");
	for(i=0 ; i<5 ; i++)
	{
		printf("%d\n", arr[i]);
	}
}
```

Find the sum of all array elements:

```
#include<stdio.h>
int main()
{
	int arr[5]={10,20,30,40,50};
	int i, sum=0;

	for(i=0 ; i<5 ; i++)
	{
		sum=sum+arr[i];
	}
	printf("Sum is : %d\n", sum);
}
```

Count the even numbers in the given array:

```
#include<stdio.h>
int main()
{
	int arr[7]={5,2,9,6,3,8,7};
	int i, count=0;

	for(i=0 ; i<7 ; i++)
	{
		if(arr[i]%2==0)
			count++;
	}
	printf("Even count is : %d\n", count);
}
```

WAP to find largest element in the array :

```
#include<stdio.h>
void main()
{
	int arr[5]={5,17,13,27,19},i,large;
	large = arr[0];
	for(i=1 ; i<5 ; i++)
	{
		if(large<arr[i])
		{
			large=arr[i];
		}
	}
	printf("large : %d\n",large);
}
```

Pass Array as a parameter to a function:

- It is not possible to pass all the elements of array to a function.
- Passing array as a parameter means, passing only base address of array.
- By using base address, if we perform any operations on index elements in the "called funtion" directly affects in the locations of "calling function".

```
#include<stdio.h>
void main()
{
	int a[5]={10,20,30,40,50},i;
	modify(a,5);
	printf("elements are\n");
	for(i=0 ; i<5 ; i++)
	{
		printf("%d\n",a[i]);
	}
}
void modify(int x[], int n)
{
	int i;
	for(i=0 ; i<n ; i++)
	{
		x[i]=x[i]+100;
	}
}

/*
Read elements into array
*/
#include<stdio.h>
int main()
{
	int arr[50], n, i;

	printf("Enter size :");
	scanf("%d" , &n);

	printf("Enter %d elements : \n",n);
	for(i=0 ; i<n ; i++)
	{
		scanf("%d" , &arr[i]);
	}

	printf("Elements are :\n");
	for(i=0 ; i<n ; i++)
	{
		printf("%d\n" , arr[i]);
	}
	return 0;
}
/*
Input :
Enter 5 elements :
Enter Ele-1 : 10
Enter Ele-2 : 20
....

Output :
arr[0] : 10             address:2048
arr[1] : 20             address:2050
....
*/
#include<stdio.h>
int main()
{
	int arr[50], n, i;

	printf("Enter size :");
	scanf("%d" , &n);

	printf("Enter %d elements : \n",n);
	for(i=0 ; i<n ; i++)
	{
		printf("Enter Ele-%d : ", i+1);
		scanf("%d" , &arr[i]);
	}

	printf("Elements are :\n");
	for(i=0 ; i<n ; i++)
	{
		printf("arr[%d] : %d \t addr : %u\n", i, arr[i], &arr[i]);
	}
	return 0;
}
```

Strings in C

String :
-> One dimensional character array.
-> It is a collection of characters & symbols.

Syntax:

```
        char  identity[size];
```

Ex:

```
        char  name[20];
```

-> We represents the string with double quotes.

```
        char  name[20] = "amar" ;
```

%s :

-> A format specifier.
-> Used to read and display strings.

```
#include<stdio.h>
int main()
{
        char name[20] = "amar" ;
        printf("Hello %s \n", name);
        return 0;
}
```

We can assign values(char by char) to string type variable:

```
#include<stdio.h>
int main()
{
        char name[20] = {'a','n','n','i','e'} ;
        printf("Hello %s \n", name);
        return 0;
}
```

-> String ends with '\0' character.
-> '\0' character ASCII value is 0.
-> We can store only n-1 characters into a string size 'n'.

Reading a string:
-> Using '%s' we can read a string.
-> We no need to use loops to read.
-> No need to specify & operator in the scanf() function to read.

```
#include<stdio.h>
int main()
```

```
{
        char name[20];

        printf("Enter your name : ");
        scanf("%s", name);
        printf("Hello %s, welcome\n", name);
        return 0;
}
```

Why we are not using loops to read strings?
-> In case of array we read specified number of elements into array.
-> We use loops to read more than one element.
-> We mention the address of each location to read the element.

```
#include<stdio.h>
int main()
{
        int arr[5],i;

        printf("Enter 5 elements : ");
        for(i=0 ; i<5 ; i++)
        {
                scanf("%d",&arr[i]);
        }
        return 0;
}
```

-> If we know the length of string, we use loops to read string elements.

```
#include<stdio.h>
int main()
{
        char vowels[6], i;

        printf("Enter vowels : ");
        for(i=0 ; i<6 ; i++)
        {
                scanf("%c", &vowels[i]);
        }

        printf("Vowels are : \n");
        for(i=0 ; i<5 ; i++)
        {
                printf("%c\n", vowels[i]);
        }
        return 0;
}
```

-> When we read a string from the console, we cannot specify its length.
-> %s collects characters one by one and place into locations from base address.

-> When we stop input, it place '\0' character at the end automatically.

```
#include<stdio.h>
int main()
{
	char name[20];

	printf("Enter name : ");
	scanf("%s", name);

	printf("Hello %s \n", name);
	return 0;
}
```

Output :

```
	Enter name : hari haran
	Hello hari
```

- **"%s"** can read a single word strings.

gets(): We can read multi word strings(sentense, paragraph...), we use gets().
gets() functions stops reading characters into string only when we press enter.

```
#include<stdio.h>
int main()
{
	char name[20];

	printf("Enter name : ");
	gets(name);

	printf("Hello %s \n", name);
	return 0;
}
```

Representing Strings with Quotes:
-> We represents strings with double quotes.
-> If we want to display a sub string or entire string in double quotes, we use escape characters.

```
/*
Output :
	It is 'C-Online' class
*/
#include<stdio.h>
int main()
{
	char str[50] = "It is 'C-Online' class";
	printf("String : %s\n", str);
	return 0;
}


/*
Output :
	It is "Live" session

Escape characters:
	\n = new line
	\t = tab space
	\b = back space
	\r = carriage return
	\' = '
	\" = "
	\\ = \
*/
#include<stdio.h>
int main()
{
	char str[50] = "It is \"Live\" session";
	printf("String : %s\n", str);
	return 0;
}
```

We can read the string directly with quotes:

```
#include<stdio.h>
int main()
{
	char str[50];

	printf("Enter String : ");
	gets(str);

	printf("String : %s\n", str);
	return 0;
}
```

Output:

```
	Enter String : "it is live session"
	String : "it is live session"
```

-> '\0' represents end of string
-> printf() function stops printing the string when it reaches '\0'

```
#include<stdio.h>
int main()
{
	printf("Hello\0World\n");
```

```
        return 0;
}
```

Output : \0 character ASCII value is : 0

```
#include<stdio.h>
int main()
{
        printf("\0 ASCII value is : %d \n", '\0');
        return 0;
}
```

```
#include<stdio.h>
int main()
{
        printf("\\0 ASCII value is : %d \n", '\0');
        return 0;
}
```

-> Just like array variables, strings also store base address of memory block.
-> Different strings(including duplicates) will get memory in different locations.

```
#include<stdio.h>
int main()
{
        char s1[10] = "Hello" ;
        char s2[10] = "Hello" ;
        if(s1==s2)
                printf("Strings are equal \n");
        else
                printf("Strings are not equal \n");
        return 0;
}
```

string.h :

-> Pre defined header file.
-> Providing functionality to process strings.

Finding the length:
-> strlen() function returns length of string.
-> it excludes null character.
-> size_t is returntype and represents unsigned integer value.

```
                size_t  strlen(char s[ ]) ;
```

```
#include<stdio.h>
#include<string.h>
int main()
```

```
{
        char s[20];
        size_t l;

        printf("Enter string : ");
        gets(s);

        l = strlen(s);
        printf("Length is : %u \n", l);
        return 0;
}
```

Finding the length without using library function.

```
#include<stdio.h>
size_t length(char[]);
int main()
{
        char s[20];
        size_t l;

        printf("Enter string : ");
        gets(s);

        l = length(s);
        printf("Length is : %u \n", l);
        return 0;
}
size_t length(char s[])
{
        size_t i=0 ;
        while(s[i] != '\0')
        {
                ++i;
        }
        return i;
}
```

How to reverse the string:

```
#include<stdio.h>
#include<string.h>
int main()
{
        char s[20];
        printf("Enter string : ");
        gets(s);

        strrev(s);
        printf("Reverse string is : %s \n", s);
        return 0;
}
```

Reverse the string without using library function:

```
#include<stdio.h>
#include<string.h>
int main()
{
        char s[20], t;
        int i,j ;
        printf("Enter string : ");
        gets(s);

        i=0;
        j=strlen(s)-1;
        while(i<j)
        {
                t = s[i];
                s[i] = s[j];
                s[j] = t;
                i++;
                j--;
        }
        printf("Reverse string is : %s \n", s);
        return 0;
}
```

How to convert upper case characters into lower case:

-> strlwr() converts all upper cases characters into lower case.

```
#include<stdio.h>
#include<string.h>
int main()
{
        char s[20];
        printf("Enter string : ");
        gets(s);

        strlwr(s);
        printf("Lower case string is : %s \n", s);
        return 0;
}
```

Without using library function:

```
#include<stdio.h>
#include<string.h>
int main()
```

```
{
	char s[20];
	int i=0;
	printf("Enter string : ");
	gets(s);

	while(s[i] != '\0')
	{
		if(s[i]>='A' && s[i]<='Z')
		{
			s[i]=s[i]+32;
		}
		++i;
	}
	printf("Lower case string is : %s \n", s);
	return 0;
}
```

STRUCTURES

Data types classified into 3 types.

Primitive types: Drawback of primitive data type variable is, it can holds only one element at a time.

```
int a ;
a = 10;
a = 20; //10 will be replaced with 20
```

Derived types:

Using Arrays we can store more than one element into a single varible.

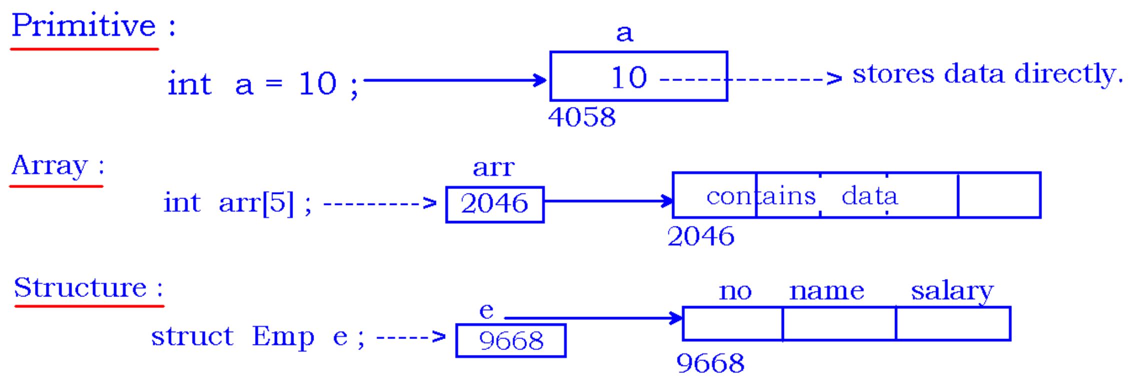
```
int a[5] = {10,20,30,40,50};
```

Problem with arrays is, it allows to store homogenous data only.

User defined types:

In some cases, if we need to store employee details or student records(contains different types of data), we use user-defined data types such as structure or union.

Structure is a user-defined template that represents data type and allows to store more than one element of different data types.



"struct" is a pre-defined keyword allows to define structure data types.

syntax :

```
struct <structure_name>
{
        data_type ele-1;
        data_type ele-2;
        ....
        ....
        data_type ele-n;
};
```

Note: structure should ends with " ; "

Example:
```
        struct Emp
        {
                int eno;
                char ename[20];
                float esal;
        };
```

Note: In the above declaration, "struct Emp" represents a form of user-defined data type.

By defining the structure, memory will not be allocated to store the elements.

When we declare a variable to the template, variable gets memory allocation to store the data.

Note: structure variable always holds base address of allocated memory.

syntax:
```
        struct <identifier> <var1>, <var2>...
```

ex:
```
        struct emp e1,e2,e3.....
```

Size of structure: size of structure is always equals to sum of sizes of individual elements defined inside the structure template.

```
#include<stdio.h>
struct emp
{
        int eno;
        char ename[20];
        float esal;
};
void main()
{
        struct emp e;
        printf("size of emp : %d\n",sizeof(struct emp));
        printf("size of emp : %d\n",sizeof(e));
}
```

Initialization of structure:

- Structure variables can be initialized directly using Assignment operator.
- Structure elements can be accessed using only " . " Operator.

```
#include<stdio.h>
struct emp
{
        int eno;
        char ename[20];
        float esal;
};
void main()
{
        struct emp e = {1001, "amar", 50000};
        printf("emp details are\n");
        printf("eno : %d\n",e.eno);
        printf("ename : %s\n",e.ename);
        printf("esal : %f\n",e.esal);
}
```

Note: structure templates either we can define globally or locally. If we define a structure inside a function, we can create variables for that structure template only within that function.

```
#include<stdio.h>
void fun(void);
void main()
{
        struct st
        {
                int a;
        };
        struct st s; //allowed (structure template has declared within the same
function).
}
void fun(void)
{
        struct st s; // not allowed.
}
```

```
#include<stdio.h>
struct GLOBAL
{
        int x, y ;
};
void fun(void);
void main()
{
        struct GLOBAL g ;
}
void fun(void)
{
        struct GLOBAL g ;
}
```

Note: It is allowed to declare a variable for user-defined data types only after definition of that data_type.

```
#include<stdio.h>
void fun(void);
void main()
{
	struct st s; //not allowed
}
struct st
{
	int a;
};
void fun(void)
{
	struct st s; //allowed
}
```

Reading elements into structure variable:

We can read the elements using " . " operators only.

```
#include<stdio.h>
struct emp
{
	int eno;
	char ename[20];
	float esal;
};
void main()
{
	struct emp e;
	printf("enter emp details\n");
	scanf("%d%s%f",&e.eno, e.ename, &e.esal);
}
```

Structure as Parameter to function:

- It is possible to pass user defined data type as Parameter.
- To collect this Parameter we need to declare structure type variable as an argument in the called function.
- In the process of passing only variable address will be stored but elements will not be duplicated.

```
#include<stdio.h>
struct emp
{
        int eno;
        char ename[20];
        float esal;
};
void display(struct emp);
void main()
{
        struct emp e;
        printf("enter emp details\n");
        scanf("%d%s%f",&e.eno, e.ename, &e.esal);
        display(e);
}
void display(struct emp x)
{
        printf("emp details are \n");
        printf("%d\n%s\n%f\n",x.eno, x.ename, x.esal);
}
```

Function returning structure :

```
#include<stdio.h>
struct emp
{
        int eno;
        char ename[20];
        float esal;
};
struct emp read(void);
void main()
{
        struct emp e;
        e = read();
        printf("emp details are \n");
        printf("%d\n%s\n%f\n",e.eno, e.ename, e.esal);
}
struct emp read(void)
{
        struct emp x;
        printf("enter emp details\n");
        scanf("%d%s%f",&x.eno, x.ename, &x.esal);
        return x;
}
```

Array of structures :

```
#include<stdio.h>
struct emp
{
        int eno;
        char ename[20];
        float esal;
};
```

```
void main()
{
        struct emp e[3];
        int i;
        printf("Enter 3 emp details\n");
        for(i=0 ; i<3 ; i++)
        {
                printf("enter emp%d details\n",i+1);
                scanf("%d%s%f",&e[i].eno, e[i].ename, &e[i].esal);
        }
        printf("Emp details are\n");
        for(i=0 ; i<3 ; i++)
        {
                printf("Emp%d details : \t",i+1);
                printf("%d\t%s\t%f\n\n",e[i].eno, e[i].ename, e[i].esal);
        }
}
```

Arrays in structures :

```
#include<stdio.h>
struct Student
{
        int sno ;
        char sname[20];
        int smarks[4];
};
void main()
{
        struct Student s[3];
        int i,j ;

        printf("Enter 3 student details :\n");
        for(i=0 ; i<3 ; i++)
        {
                printf("Enter student-%d details : \n", i+1);
                scanf("%d%s",&s[i].sno, s[i].sname);
                for(j=0 ; j<4 ; j++)
                {
                        scanf("%d", &s[i].smarks[j]);
                }
        }

        printf("Student details : ");
        for(i=0 ; i<3 ; i++)
        {
                printf("Student-%d details : \n", i+1);
                printf("%d\t%s\t",s[i].sno, s[i].sname);
                for(j=0 ; j<4 ; j++)
                {
                        printf("%d\t", s[i].smarks[j]);
                }
                printf("\n");
        }
}
```

Copying structure elements :

```
#include<stdio.h>
struct emp
{
        int eno;
        char ename[20];
        float esal;
};
void main()
{
        struct emp e1={1001,"amar",50000};
        struct emp e2,e3;

        //element by element copying
        e2.eno = e1.eno;
        strcpy(e2.ename, e1.ename);
        e2.esal = e1.esal;

        //all at once
        e3 = e2; //should be of same type

        printf("%d\t%s\t%f\n",e1.eno, e1.ename, e1.esal);
        printf("%d\t%s\t%f\n",e2.eno, e2.ename, e2.esal);
        printf("%d\t%s\t%f\n",e3.eno, e3.ename, e3.esal);
}
```

Nested structures :

- Defining the structure inside another structure.
- Using nested structures we can define complex data types.
- Outer structure elements we can refer directly.
- Inner structure elements must be accessed with the help of Outer structure elements.

```
#include<stdio.h>
struct emp
{
        int eno;
        float esal;
};
struct employee
{
        struct emp e;
        char ename[20];
};
```

```
void main()
{
        struct employee e1;
        printf("Enter emp details\n");
        scanf("%d%s%f",&e1.e.eno, e1.ename, &e1.e.esal);
}
```

Pointers

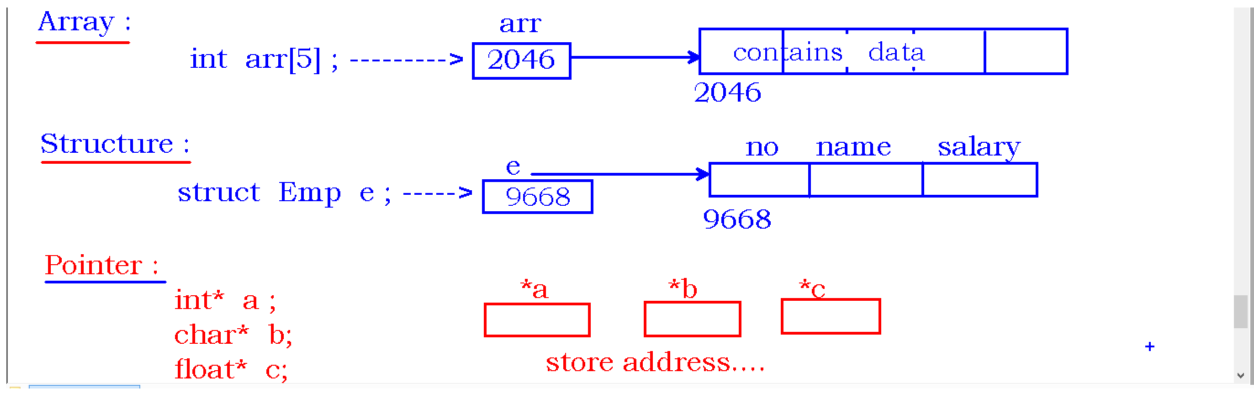
- Pointer is an aggregative data type.
- Pointer stores address(reference) of another variable.
- Pointers are called indirection operators.
- Pointers are used to access the information of a variable indirectly.

syntax :

```
        data_type  *<identifier>;
```

ex :

```
        int *ptr    (or)    int* ptr;
```



Typed pointers : These pointers can point to specific type of data.

```
                int* ----> int data
                char* ---> char data
                function* ----> function
```

Un typed pointer : An untyped pointer can points to any type of data. It is also called Generic pointer.

```
                void* ----> any data
```

In pointers every operation can be performed using two operators...

" & " :

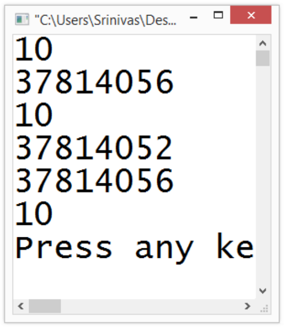
It is called address operator.

It returns memory address of specific variable.

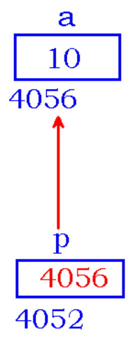
" * " :

It is called pointer operator.
It returns the data which has stored inside the specific address.

```c
#include<stdio.h>
int main()
{
    int a=10;
    int* p=&a;

    printf("%u \n", a);
    printf("%u \n", p);
    printf("%u \n", *p);
    printf("%u \n", &p);
    printf("%u \n", *(&p));
    printf("%u \n", **(&p));
    return 0;
}
```



```
10
37814056
10
37814052
37814056
10
Press any ke
```



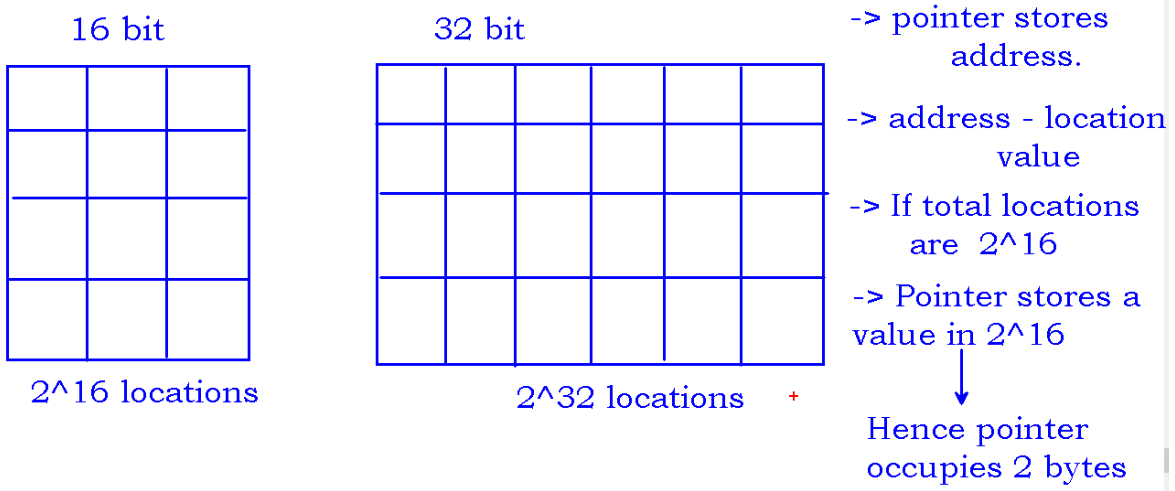
Address operator : Returns memory address of specified variable.

Pointer operator : Returns the data inside the memory address.

Size of pointer :

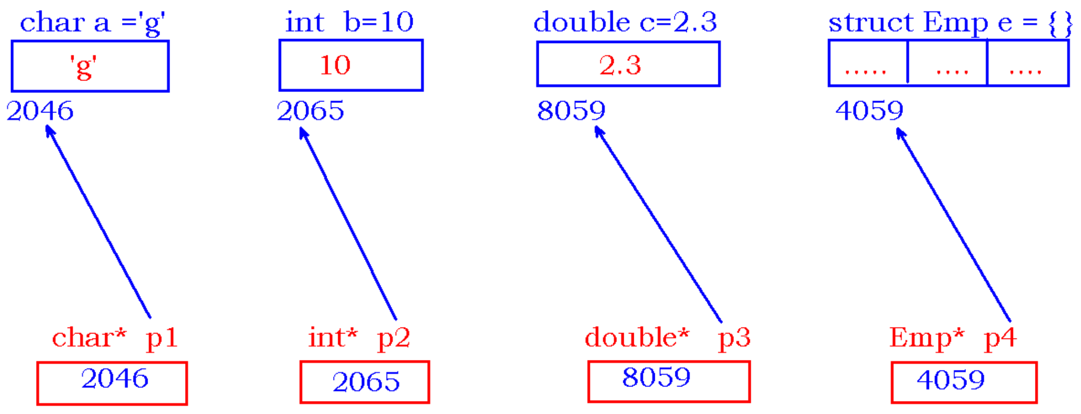
- Size of pointer is equals to size of integer on the machine.
- Pointer size also varies from compiler to compiler like integer.

- If compiler is 16 bit --> Pointer size is 2 bytes
- If compiler is 32 bit --> Pointer size is 4 bytes


16 bit
2^16 locations
32 bit
2^32 locations
-> pointer stores address.
-> address - location value
-> If total locations are 2^16
-> Pointer stores a value in 2^16
Hence pointer occupies 2 bytes

- In a 16-bit compiler we have totally 2^16 memory locations.
- Pointer variable holds address. Hence it is any one of 2^16 locations.
- To hold any one of these addresses(integer) pointer occupies 2 types.

```
#include<stdio.h>
struct st
{
        int i ;
        char c ;
        float f ;
};
void main()
{
        short* sp ;
        char* cp ;
        struct st* stp;

        printf("size of short* : %d\n", sizeof(sp));
        printf("size of char* : %d\n", sizeof(cp));
        printf("size of float* : %d\n", sizeof(float*));
        printf("size of int* : %d\n", sizeof(int*));
        printf("size of struct st* : %d\n", sizeof(stp));
}
```



Call by value:

- Calling a function by passing values as parameters.
- These values will be collected in to formal arguments.
- Formal arguments are working like local variables of function.
- Data processed inside the Called function will modify formal arguments.

Draw back of "Call by value" is, the processed data in the "Called function" cannot be accessed in the "Calling function".

```
#include<stdio.h>
void swap(int,int);
void main()
{
        int a,b;
        printf("Enter two numbers\n");
        scanf("%d%d",&a,&b);
        printf("Before swap : \t%d\t%d\n",a,b);
        swap(a,b);
}

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

Problems of call by value:

- if we modify the variables in the called function(swap), if effects the local variables of called function only.
- The processed data in the called function cannot be accessed from calling function (main) and can't return to calling function.

Call by reference:

- Calling a function by passing address(reference) as parameter.
- We collect these parameters into arguments of Pointer type.
- Using these pointers we can process the information of calling function directly.
- We can access the processed information either from calling function or called function.

```
#include<stdio.h>
void swap(int*,int*);
void main()
{
        int a,b;
        printf("Enter two numbers\n");
        scanf("%d%d",&a,&b);
        printf("Before swap : \t%d\t%d\n",a,b);
        swap(&a,&b);
        printf("after swap : \t %d \t %d \n",a,b);
}
void swap(int* x, int* y)
{
        int temp;
        temp = *x;
        *x = *y;
        *y = temp;
        printf("after swap : \t %d \t %d \n",*x,*y);
}
```

Pointer to function:

- It is also possible to create pointers to Derived data types such as function, array, string .....
- To execute block of instructions, OS allocates the memory in stack area is called "FRAME".
- Every "FRAME" has a base address, that is working like entry point to execute function.
- Pointer to function variable holds a reference of FRAME.

Syntax :

```
	<return_type> (*<ptr_name>)(args_list);
```

Example :

```
	int (*fptr)(int,int);
```

- In the above declaration "fptr" is a pointer can points to any function which is taking two integer arguments and returns integer-data.
- Function-pointer declaration completely depends on prototype of function.
- Prototype of function for above pointer variable :
  <return_type> <fun_name>(<args_list>)

```
#include<stdio.h>
int add(int,int);
int sub(int,int);
void main()
{
	int r1,r2,r3,r4;
	int (*fptr)(int,int);
	r1=add(10,20);
	r2=sub(10,20);
	printf("r1 : %d\nr2 : %d\n",r1,r2);

	fptr = &add; /*pointing to add function*/
	r3=fptr(30,50);

	fptr = &sub; /*pointing to sub function*/
	r4=fptr(30,50);
	printf("r3 : %d\nr4 : %d\n",r3,r4);
}
int add(int x, int y)
{
	return x+y;
}
int sub(int x, int y)
{
	return x-y;
}
```

Question: Why we need to place "fptr" inside paranthesis.
Answer:

```
	int (*fptr)(int,int);

	re-write as..
		int  *fptr(int,int);

	will be considered as..
		int* fptr(int, int);
```

The above declaration describes "fptr" is a function which is taking two integer arguments and returning "address of integer data".

A function returning address :
A function can return address of aggregate data type variable or user defined data type variable to access the data.

```
#include<stdio.h>
int* add(int,int);
void main()
{
	int a,b;
	int* c;
	printf("enter two numbers : ");
	scanf("%d%d",&a,&b);
	c = add(a,b);
	printf("sum : %d\n",*c);
}
int* add(int x, int y)
{
	int z;
	z=x+y;
	return &z;
}
```

Pointer type-casting :

- Typed pointers can points to specific type of data.
- We cannot point one type of data using another type of pointer.
- We need to type cast pointers to make it points to another kind of data.
- Generic pointer(void*) can implicitly type casted to any other pointer type. Hence type conversion is not required for void*.

```
#include<stdio.h>
void main()
{
	int i = 100;
	int* ip;
	char* cp;
```

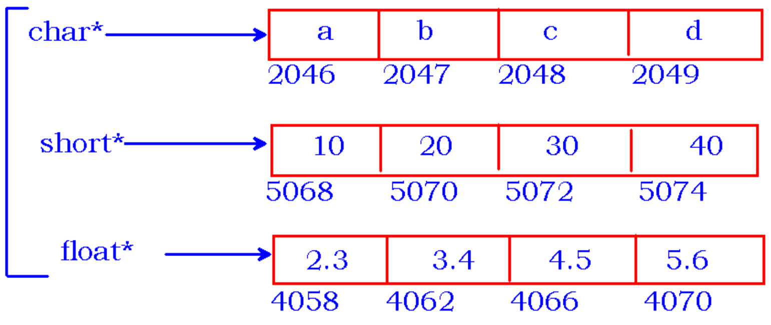
```
        ip = &i;
        cp = (char*)ip;
        printf("i value : %d\n", *ip);
        printf("i value : %d\n", *cp);
}
```

Pointer arithmetic :

- Modifying operators used to increase or decrease the value of a variable.
- Increment operators increases the value of a variable by 1
- Decrement operators decreases the value of a variable by 1
- When we modify a pointer variable using modifying operators, the value of variable will be increased or decreased by "size" bytes depends on data stored inside the location.

What is the use of pointer arithmetic?
-> To process(read, display or modify) the values of arrays using pointers.

Using pointer arithmetic, we can move the control from one location to another location to process the data easily.



```
#include<stdio.h>
void main()
{
        short sa[5]={10, 20, 30, 40, 50};
        char ca[5]={'a', 'b', 'c', 'd', 'e'};

        short* sp;
        char* cp ;

        sp = sa ;
        cp = ca ;

        printf("Initial loc of short* : %u\n", sp);
        printf("Locaction after modify short* : %u\n", ++sp);

        printf("Initial loc of char* : %u\n", cp);
        printf("Locaction after modify char* :%u\n", ++cp);
}
```

Pointer to array : When a pointer pointing to the array we use many ways to access elements.

```
	arr[i] ----> *(arr+i);
	i[arr] ----> *(i+arr);
```

```
#include<stdio.h>
void main()
{
	int iarr[5] = {10,20,30,40,50},i;
	int* ptr;
	ptr = iarr;
	printf("elements are :\n");
	for(i=0 ; i<5 ; i++)
	{
		printf("%d,%d,%d,%d,%d\n", iarr[i], ptr[i], *(ptr+i), *(i+ptr), i[ptr]);
	}
}
```

Note : Priority of operators need to be considered while evaluating the expressions to access the elements of array using pointers.

First priority : ++ , --
Second priority : *
Third priority : arithmetic operators
Note : ( ) having higher priority than all.

```
#include<stdio.h>
void main()
{
	int arr[5] = {10,20,30,40,50},i;
	int* ptr;
	ptr = arr;
	printf("%u\n", *++ptr + 3);
	printf("%u\n", *(ptr-- + 2) + 5);
	printf("%u\n", *(ptr+3)-10);
}
```

```
#include<stdio.h>
void main()
{
	int arr[5] = {8, 3, 4, 9, 2},i;
	int* ptr;
	ptr = arr;
	printf("%u\n", *(--ptr+2) + 3);
	printf("%u\n", *(++ptr + 2) - 4);
```

```
        printf("%u\n", *(ptr-- +1 ) + 2);
}
```

Array of pointers : Array of pointers variable holds more than one element address.

```
#include<stdio.h>
void main()
{
        int iarr[5] = {10,20,30,40,50},i;
        int* ptr[5];

        for(i=0 ; i<5; i++)
        {
                ptr[i] = &iarr[i];
                //ptr[i] = iarr+i;
        }
        printf("array elements are \n");
        for(i=0 ; i<5; i++)
        {
                printf("%d\n", *ptr[i]);
                //printf("%d\n", *(*(ptr+i)));
        }
}
```

Pointer to structure:

- It is allowed to create pointers to user-defined data types also such as structures and unions.
- When a pointer pointing to structure, we use "->" operator to access the elements instead of "." operator.

```
#include<stdio.h>
struct emp
{
        int eno;
        char ename[20];
        float esal;
};
void main()
{
        struct emp e;
        struct emp* ptr;
        ptr = &e;

        printf("enter emp details\n");
        scanf("%d%s%f",&ptr->eno, ptr->ename, &e.esal);

        printf("Emp details are :\n");
        printf("%d\n%s\n%f\n",e.eno, ptr->ename, ptr->esal);
}
```

Pointer to string :

A char* can points to

1) a single character data
2) or a block of characters(String)

```
#include<stdio.h>
void main()
{
        char* s = "Naresh" ;
        printf("%s \n", s);
        printf("%c \n", s);
        printf("%c \n", *s);
        printf("%c \n", *(s+3));
        printf("%c \n", *s+3);
}
```

```
#include<stdio.h>
void main()
{
        char* str = "learnown";
        printf("%c\n", *str++ + 3);
        printf("%s\n", ++str+2);
}
```

```
#include<stdio.h>
void main()
{
        char* str = "learnown";
        printf("%c\n" , *(str++ + 2)+3);
        printf("%c\n" , *++str+2);
        printf("%s\n" , --str-1);
}
```

```
#include<stdio.h>
void main()
{
        char sport[ ]= "cricket";
        int x=1 , y;
        y=x++ + ++x;
        printf("%c",sport[++y]);
}
```

```
#include<stdio.h>
char* read(void);
void main()
{
        char* name;
        name = read();
        printf("name is : %s\n",name);
}
char* read(void)
{
        char* name;
        printf("Enter one name : ");
        gets(name);
        return name;
}
```

Two dimensional array of characters:

```
#include<stdio.h>
void main()
{
        char* names[ ] = {"one", "two", "three", "four"};
        int i;
        for(i=0 ; i<4 ; i++)
        {
                printf("%s\n",names[i]);
                printf("%c\n",*names[i]);
        }
}
```

```
#include<stdio.h>
void main()
{
        char* s[5] = {"one", "two", "three", "four","five"};
        int i=0,j=4;
        char* t;
        while(++i < j--)
        {
                t = s[i];
                s[i] = s[j];
                s[j] = t;
        }
        printf("%s",*(s + ++i));
}
```

Pointer to pointer:

- A pointer to pointer type variable holds address of another pointer variable.

- It is also called double pointer.
- Using these type of pointers, we can define complex pointers programming.

```
#include<stdio.h>
void main()
{
	int i = 100;
	int* ip;
	int** ipp;
	ip = &i;
	ipp = &ip;
	printf("%d,%d,%d",i,*ip,**ipp);
}
```

```
#include<stdio.h>
void main()
{
	char* s[] = {"abc","cde","efg"} ;
	int i ;
	for(i=0 ; i<3 ; i++)
	{
		printf("%c \n", *(*(s+i)+i));
	}
}
```

```
#include<stdio.h>
void main()
{
    char *s[ ] = {"black", "white", "pink", "violet"};
    char **ptr[ ] = {s+3, s+2, s+1, s};
    char ***p;
    p = ptr;
    ++p;
    printf("%s\n", (*(*p+1)+1)+2);
}
```

Functions

- A function is a block of code that performs a specific task.
- It has a name and it is reusable.
- Function taking input, processing input and produce output.
- It can be accessed from as many different parts in a C Program as required.

Syntax :

```
	int identity(arguments)
```

```
        {
                -> collect input(args)
                -> process input
                -> return output
        }
```

Ex :

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

Ex : ATM transaction

```
        message  withdraw(card details , pin , amount)
        {
                -> card is authroized or not
                -> pin is valid or not
                -> amount is present or not
                -> process , release and release card.
        }
```

Why functions?

- Use of functions is code re-usability.
- Using functions we can implement modularity programming.
- The use of user-defined functions allows a large program to be down into a number of smaller, self-contained components, each of which has some unique, identifiable purpose.
- Thus a C program can be modularized through the intelligent use of such functions.

Types of Functions:

1. Built In Functions
2. User Defined Functions

Built in Functions:

- C-language having library.
- Library is a collection of header files.
- Header file is a collection of pre-defined functions.
- These functions are also called as 'library functions'.
- **Examples...**

```
        scanf()
        printf()
        clrscr()
```

User Defined Functions:

- A programmer defined function depends on application requirement.
- Need to follow syntactical rules to define a function.
- It is possible for the programmer to define user header files.
- **Examples…**

```
add();
sub();
withdraw();
deposit();
```

Classification of functions:

1. No arguments and no return values
2. With arguments and no return values
3. With arguments and with return values
4. No arguments and with return values
5. Recursive function

Every function having 3 things

1. Prototype
2. Definition
3. Function call

Prototype:

- It is a statement.
- It provides the information to the compiler and executor about user-defined functions.
- It is equals to declaration of variable.
- It is used to decide how much memory is required to execute function.

Definition:

- Block of statements.
- Contains function logic.

Function call:

- It is a statement.
- Used to access function logic.

Note the followings:

- Every function gets executed inside the stack area.
- When function execution starts, amount of space will be allocated inside the stack area to execute function logic is called FRAME.
- FRAME gets destroyed once function execution has been completed.

- "STACK memory full" error causes abnormal termination of program.

No arguments & No return vals	With arguments & No return vals	With arguments & With return vals	No arguments & With return vals
void func(void) ;	void func(int, int);	float func(int);	char func(void);
void func(void) { logic ; }	void func(int a, int b) { logic ; }	float func(int x) { float y = 2.3 ; return y ; }	char func(void) { char ch = 'g' ; return ch ; }
func( ) ;	func(10 , 20);	float a = func(10);	char sym = func( ) ;

No arguments & No return values :

```
#include<stdio.h>
void sayHi(void);
int main(void)
{
        printf("main \n");
        sayHi();
        sayHi();
        sayHi();
        return 0 ;
}
void sayHi(void)
{
        printf("Hi to all \n");
}
```

With arguments and No return values :

```
#include<stdio.h>
void printMessage(char*);
int main(void)
{
        printf("main \n");
        printMessage("Hello");
        printMessage("all");
        printMessage("How r u");
        return 0 ;
}
void printMessage(char* s)
{
        printf("Message is : %s\n", s);
}
```

```
#include<stdio.h>
void isEven(int);
int main(void)
{
	int n;
	printf("Enter an int : ");
	scanf("%d", &n);
	isEven(n);
	return 0 ;
}
void isEven(int n)
{
	if(n%2==0)
		printf("%d is even \n", n);
	else
		printf("%d is not even \n", n);
}
```

Note: Arguments of a function working like local variables and gets memory allocation inside the frame. Once Frame execution has been completed, all the local variables gets destroyed.

```
#include<stdio.h>
void ascii_value(char);
int main(void)
{
	char sym ;
	printf("Enter symbol : ");
	scanf("%c", &sym);
	ascii_value(sym);
	return 0;
}
void ascii_value(char sym)
{
	printf("ASCII value is : %d \n", sym);
}
```

Program to check input number is Even or not

```
#include<stdio.h>
void isEven(int);
void main(void)
{
	int n;
	printf("Enter n value : \n");
	scanf("%d",&n);
	isEven(n);
}
void isEven(int x)
{
        if(x%2==0)
                printf("%d is Even \n");
        else
                printf("%d is not Even \n");
}
```

With arguments and With return values:

```
#include <stdio.h>
int add(int, int);
int main()
{
   int a, b, res;
   printf("Enter two numbers : \n");
   scanf("%d%d", &a, &b);
   res = add(a,b);
   printf("%d + %d = %d\n", a, b, res);
   return 0;
}
int add(int x, int y)
{
        int z=x+y ;
   return z;
}
```

Biggest of 2 numbers :

```
#include <stdio.h>
int big(int, int);
int main()
{
   int a,b ;
   printf("Enter 2 numbers : \n");
   scanf("%d%d", &a, &b);
   printf("Big one is : %d \n", big(a,b));
   return 0;
}
int big(int a, int b)
{
        return a>b ? a : b ;
}
```

Program to check input number is Perfect Number or Not

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

```
int isPerfect(int);
void main(void)
{
        int n,res;
        printf("Enter n value : \n");
        scanf("%d",&n);
        res = isPerfect(n);
        if(res)
                printf("Perfect Number \n");
        else
                printf("Not a Perfect Number \n");
}
int isPerfect(int x)
{
        int i, sum=0;
        for(i=1 ; i<x ; i++)
        {
                if(x%i==0)
                {
                        sum+=i;
                }
        }
        if(sum==x)
                return 1 ;
        else
                return 0 ;
}
```

No arguments and With return values :

```
float fun(void);
void main(void)
{
        float f;
        f = fun();
        printf("returned value : %f\n",f);
}
float fun(void)
{
        return 34.56;
}
```

Performing all arithmetic operations using functions:

```
#include <stdio.h>
float add(float,float);
float subtract(float,float);
float multiply(float,float);
float divide(float,float);
int main()
{
        int choice;
        float a,b,c;
   while(1)
   {
        printf("1.Add\n");
        printf("2.Subtract\n");
        printf("3.Multiply\n");
        printf("4.divide\n");
        printf("5.Exit\n");

        printf("Enter your choice : ");
        scanf("%d" , &choice);

        if(choice>=1 && choice<=4)
        {
                printf("Enter 2 numbers :\n");
                scanf("%f%f", &a, &b);
        }

        switch(choice)
        {
                case 1 :        c = add(a,b);
                                        printf("Result : %f\n\n",c);
                                        break ;

                        case 2 :        c = subtract(a,b);
                                        printf("Result : %f\n\n",c);
                                        break ;

                        case 3 :        c = multiply(a,b);
                                        printf("Result : %f\n\n",c);
                                        break ;

                        case 4 :        c = divide(a,b);
                                        printf("Result : %f\n\n",c);
                                        break ;

                        case 5 :        exit(1);
                        default         :       printf("Invalid choice \n\n");
        }
   }
   return 0;
}
float add(float x, float y)
{
        return x+y ;
}
float subtract(float x, float y)
{
        return x-y ;
}
float multiply(float x, float y)
{
        return x*y ;
}
float divide(float x, float y)
{
        return x/y ;
}
```

Recursion:

- Function calling itself

  or
- Calling a function from the same function defintion.

```
#include<stdio.h>
void main(void)
{
        printf("main\n");
        main();
        printf("end of main\n");
}
```

```
#include<stdio.h>
void func(void);
void main()
{
        printf("Main \n");
        func();
}
void func()
{
        printf("func \n");
        main();
}
```

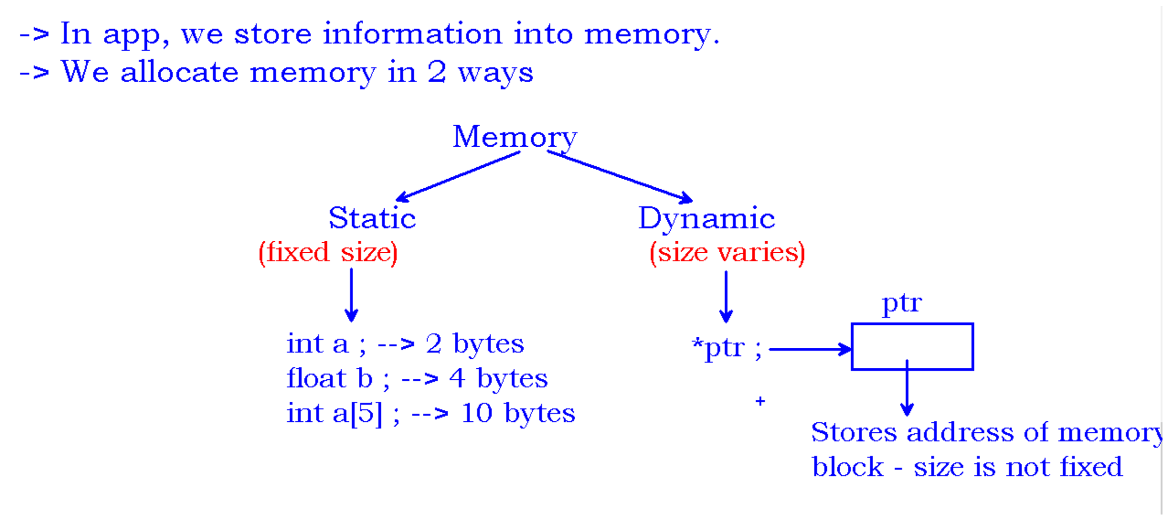
Find factorial using Recursion :

```
#include<stdio.h>
int fact(int);
void main(void)
{
        int n,factorial;
        printf("Enter one number : ");
        scanf("%d",&n);
        factorial = fact(n);
        printf("result : %d\n",factorial);
}
int fact(int n)
{
        int factorial;
        if(n==0)
        {
                factorial=1;
        }
        else
        {
                factorial = n*fact(n-1);
        }
        return factorial;
}
```

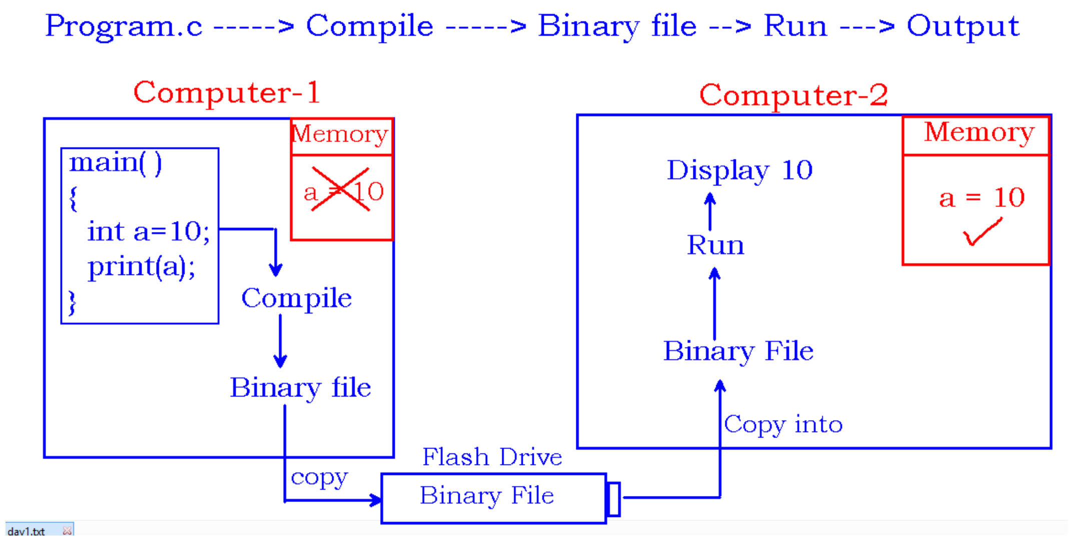
```
#include<stdio.h>
void abc(int);
void main()
{
        printf("Starts @ main \n");
        abc(3);
        printf("Ends @ main \n");
}
void abc(int n)
{
        printf("n : %d\n",n);
        if(n>0)
                abc(n-1);
}
```

```
#include<stdio.h>
void xyz(int);
void main()
{
        int x=6;
        xyz(x);
}
void xyz(int n)
{
	if(n>0)
		xyz(n-2);

	printf("n : %d\n",n);
}

#include<stdio.h>
void xyz(int);
void main()
{
	int x=3;
	xyz(x++);
	printf("x : %d\n", x);
}
void xyz(int n)
{
	printf("n : %d\n",n);

	if(n)
		xyz(--n);

	printf("n : %d\n",n);
}
```

Dynamic Memory Allocation

- In C, it is allowed to allocate the memory to variables either at compile time(static) or at runtime(dynamic).
- 
-> In app, we store information into memory.
-> We allocate memory in 2 ways
Memory
Static
(fixed size)
int a ; --> 2 bytes
float b ; --> 4 bytes
int a[5] ; --> 10 bytes
Dynamic
(size varies)
*ptr ;
ptr
Stores address of memory block - size is not fixed

- Memory will be allocated to variables at runtime only
- Compiler only converts the source code into binary code
- Static memory is nothing but fixed size but allocates at runtime only
- Dynamic memory varies depends on application requirement



- Primitive variables always get static memory allocation only.
- DMA is applicable only to Aggregative data types(arrays) and user-defined data types(structures).
- We use DMA, only when we don't know about how much memory has to be allocated for a variable at compile time.
- stdlib.h library contains pre-defined functions to allocate and de-allocate the memory at runtime.

malloc() --> structure variables
calloc() --> array variables
realloc() --> to re-aquire the space
free() --> to release the memory

- DMA refers allocating block of memory from available space and stores the reference into a pointer variable. Hence we can work with that block throughout the program using that pointer variable.

malloc() : Allocate memory to structures.

prototype :

```
	void* malloc(size_t size);
```

- "size_t" represents "unsigned" value.
- malloc() function allocates number of bytes specified by the "size".
- on success, it returns base address of allocated block.
- on failure, it returns NULL.

```
#include<stdio.h>
#include<stdlib.h>
struct emp
{
	int eno;
	char ename[20];
	float esal;
};
void main()
{
	struct emp *ptr;
	ptr = (struct emp*)malloc(sizeof(struct emp));
	if(ptr==NULL)
	{
		printf("out of memory\n");
		exit(1);
	}
	printf("Enter emp details\n");
	scanf("%d%s%f",&ptr->eno, ptr->ename, &ptr->esal);

	printf("Emp details\n");
	scanf("%d\n%s\n%f\n",ptr->eno, ptr->ename, ptr->esal);
}
```

calloc() :

```
	void* calloc(size_t n , size_t size);
```

- used to allocate the memory for array variables through pointers.
- first argument specifies, number of elements in the array.
- second argument specifies, size of each element.
- if memory is available, it allocate n*size bytes and returns first byte address.
- if no memory, it returns NULL.
- After allocating the memory all the byte locations are initialized with "0" by default.

```
#include<stdio.h>
#include<stdlib.h>
void main()
{
	int *ptr, n,*i;
	printf("enter size of array : ");
	scanf("%d",&n);

	ptr = (int*)calloc(n,sizeof(int));
	if(ptr == NULL)
	{
		printf("out of memory\n");
		exit(1);
	}
	printf("Array elements are\n");
	for(i=ptr ; i<ptr+n ; i++)
	{
		printf("%d\n",*i);
	}
}
```

```
	ptr = (int*)calloc(n,sizeof(int));
			or
	ptr = (int*)malloc(n*sizeof(int));
```

- malloc() funtion can also allocates the memory for array variables but all the byte locations initilaized with garbage values.

realloc :

```
	void* realloc(void* ptr, size_t size);
```

- Using malloc function, we allocate memory for structure variables.
- Mostly data structure algorithms implementation like linked lists, trees and graphs uses struture data type to create nodes.
- for example in a double linked list, node structure will be...

```
struct node
{
        int data ;
        struct node* left;
        struct node* right;
};
```

- In case of structure data type, we allocate the block of memory if required and deletes the block if not required.
- But in case of arrays, once if we allocate the memory using calloc() function, that will be the fixed memory to store and process the elements
- We can't increase and decrease the size of the memory block.
- realloc( ) functions is used to increase or decreaase the size of memory block.
- realloc( ) is mostly used to modify the size of array which has created dynamically.
- first argument specifies the address of memory block from which it has to re-aquire the space
- second argument specifies new size of block.
- on success, returns first byte address.
- on failure, returns NULL pointer.

```
#include<stdlib.h>
void main()
{
        int *p1,*p2, m,n;
        printf("enter size of array : ");
        scanf("%d",&m);

        p1 = (int*)calloc(m,sizeof(int));
        if(p1 == NULL)
        {
                printf("calloc failed\n");
                exit(1);
        }
        else
        {
                printf("memory allocated..\n");
        }

        printf("Enter new size of array : ");
        scanf("%d",&n);

        p2=(int*)realloc(p1 , n*sizeof(int));
        if(p2 == NULL)
        {
                printf("realloc failed\n");
                exit(1);
        }
        else
        {
                printf("memory re-aquired..\n");
        }
        free(p1);
        free(p2);
}
```

free() :

- The memory which has allocated dynamically must be released explicitly using free function.
  ```
  void free(void* ptr);
  ```

- using free() funtion , we can de-allocate the memory of any type of pointer.