A T M E College of Engineering

DATA STRUCTURES AND APPLICATIONS

Course Name	
Course Code	BCS304
Contact Hours	50
CIE Marks	50
SEE Marks	50

Course Coordinator

Mrs. MADHU NAGARAJ
Assistant Professor
Dept of CSE-DS
ATMECE, Mysuru

INTRODUCTION TO DATA STRUCTURES
Module 1

Mrs. Madhu Nagaraj
Assistant Professor
Dept of CSE-Data Science
ATMECE

Course Learning Objectives

CLO 1. To explain fundamentals of data structures and their applications.

CLO 2. To illustrate representation of Different data structures such as Stack, Queues, Linked Lists, Trees and Graphs.

CLO 3. To Design and Develop Solutions to problems using Linear Data Structures

CLO 4. To discuss applications of Nonlinear Data Structures in problem solving.

CLO 5. To introduce advanced Data structure concepts such as Hashing and Optimal Binary Search Trees

Course Outcomes

CO 1. Explain different data structures and their applications.

CO 2. Apply Arrays, Stacks and Queue data structures to solve the given problems.

CO 3. Use the concept of linked list in problem solving.

CO 4. Develop solutions using trees and graphs to model the real-world problem.

CO 5. Explain the advanced Data Structures concepts such as Hashing Techniques and Optimal Binary Search Trees.

Text Book

Ellis Horowitz and Sartaj Sahni, Fundamentals of Data Structures in C, 2nd Ed, Universities Press, 2014.

Seymour Lipschutz, Data Structures Schaum's Outlines, Revised 1st Ed, McGraw Hill, 2014.

Question Paper Pattern

The question paper will have ten questions.

Each full Question consisting of 20 marks.

There will be 2 full questions (with a maximum of four sub questions) from each module.

Each full question will have sub questions covering all the topics under a module.

The students will have to answer 5 full questions, selecting one full question from each module.

- "Get your data structures correct first, and the rest of the program will write itself."

- -Davids Johnson

Program = Data Structure + Algorithm

“What is data? “

DICT definition - The quantities, characters, or symbols on which a computer performs operations may be stored and transmitted in the form of electrical signals and recorded on magnetic, optical, or mechanical recording media.

- DATA and INFORMATION are often confusing, and we often interchange these two terms.
- UHDAM SI EMAN YM is just data because it is merely a collection of characters, and from this, the user cannot understand anything properly

What is information?”

- If data is arranged systematically, then it gets a structure and becomes meaningful.

The need for Data Structures?

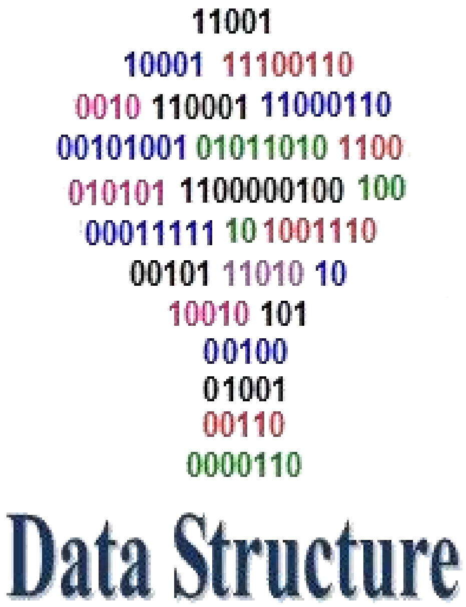
- We can understand very well that the data needs to be managed in such a way so that it can produce some meaningful information.
- Data structures give us the way to manage the data appropriately so that we can use it whenever possible.

Data Structures

Data Structure is a way to store and organize data so that it can be used efficiently .
There are many ways of organizing the data in the memory, i.e., array.

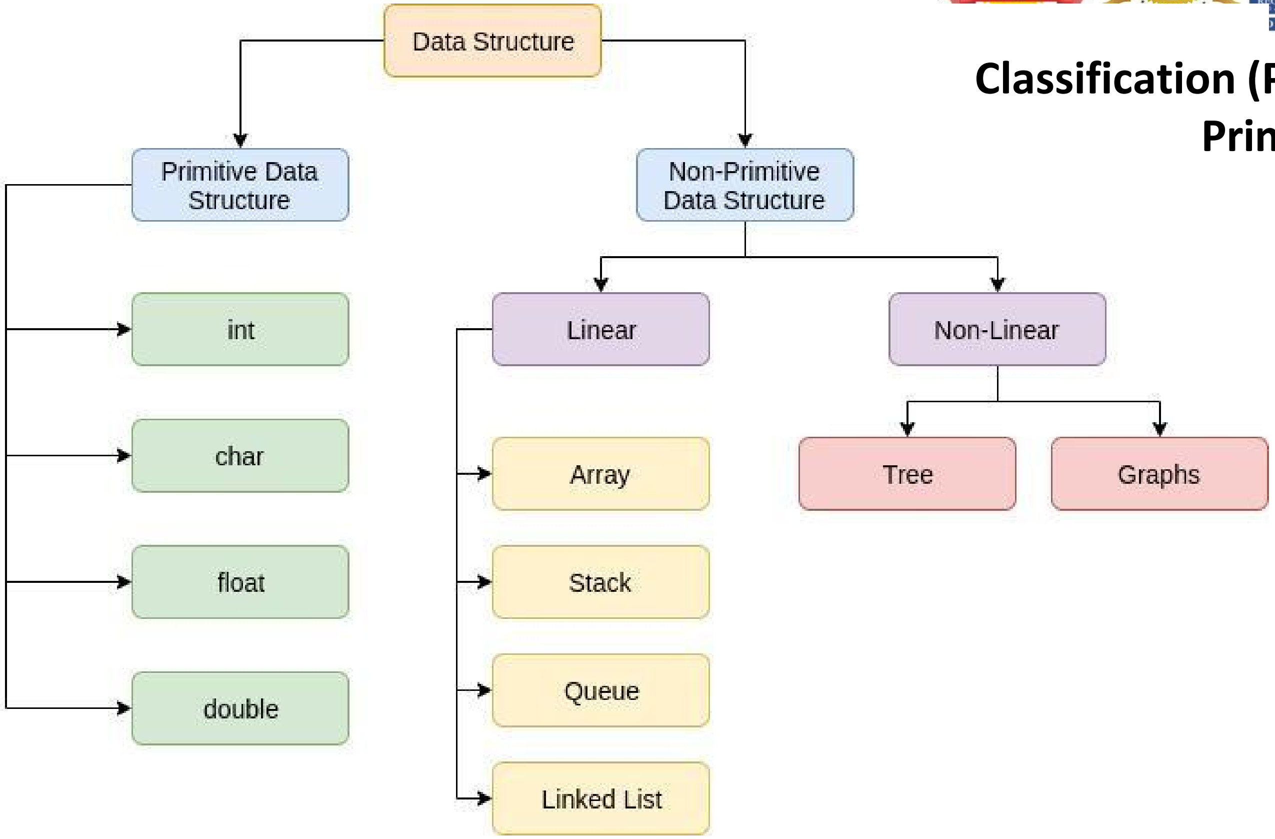
Array is a collection of memory elements in which data is stored sequentially, i.e., one after another.

There are also other ways to organize the data in memory.
Let's see the different types of data structures.

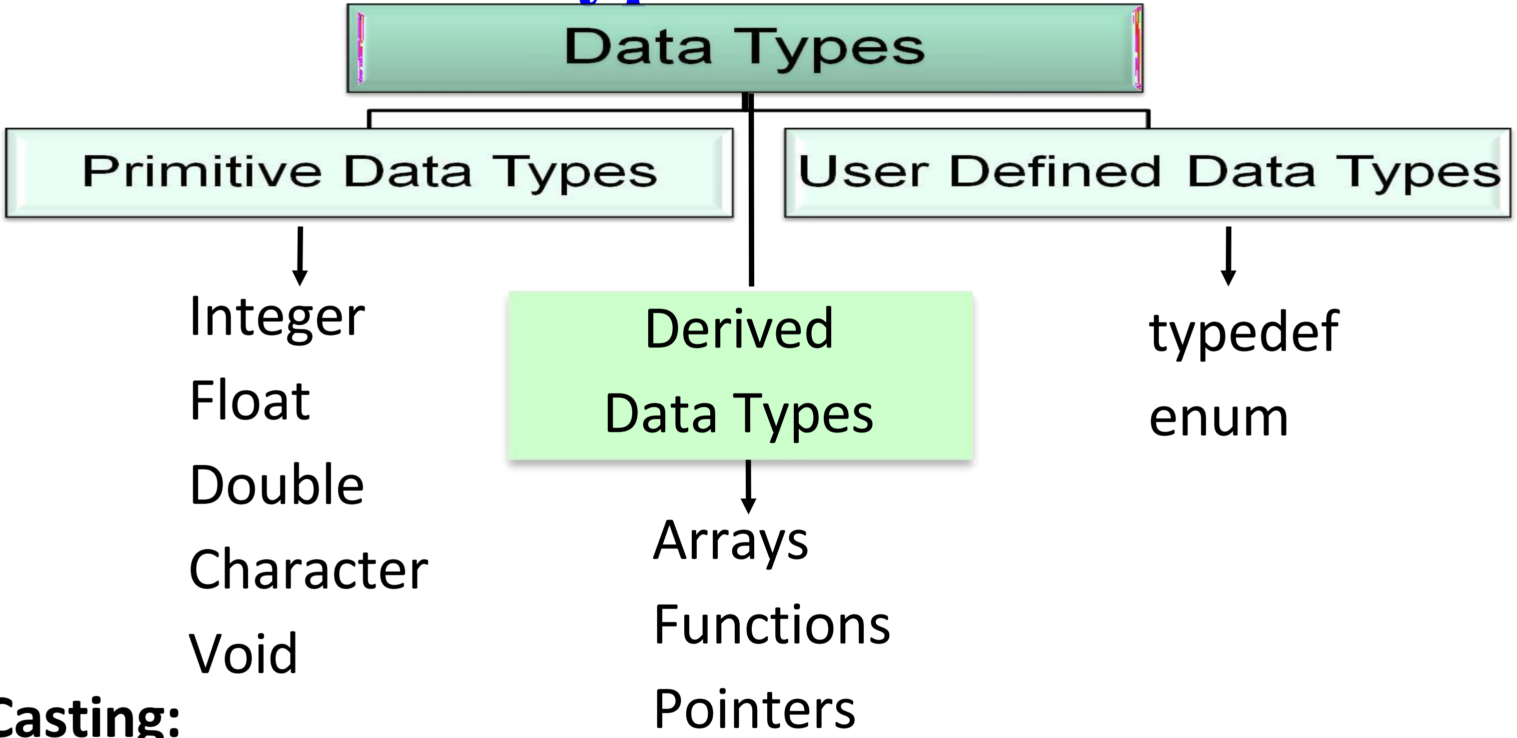


- Deals with how
- - organization of data in memory
- - efficient Storage of data in memory
- - efficient Retrieved & manipulated
- - logical relationships among different data items

Classification (Primitive and Non Primitive)



Data Types in C



Type Casting:

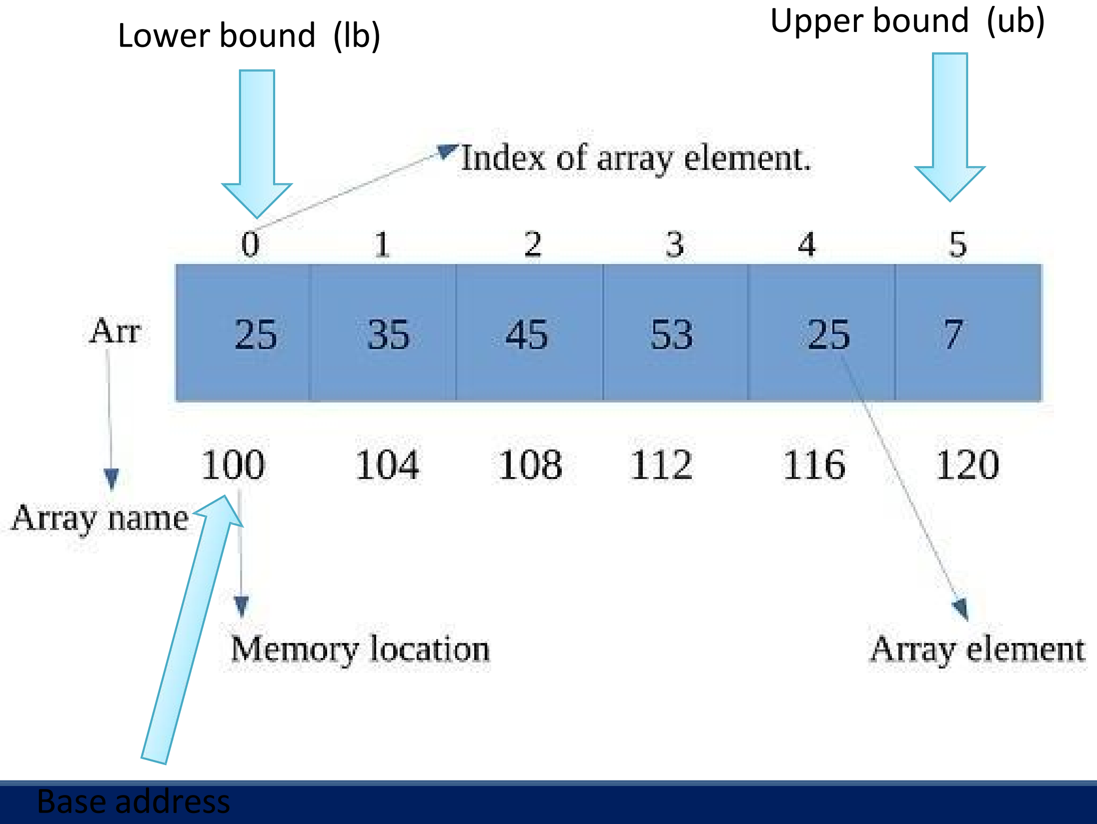
float div=float(a/b); //a & b are integer variables

Data type	Purpose	No. of bytes allocated	Range
int	To hold integer constant	2 bytes	-32768 to +32767
float	To hold real constant	4 bytes	-3.4e38 to +3.4e38
double	To hold real constant	8 bytes	-1.7e308 to +1.7e308
char	To hold character constant	1 byte	-128 to +127
void	non-specific	No memory is allocated	------

Array

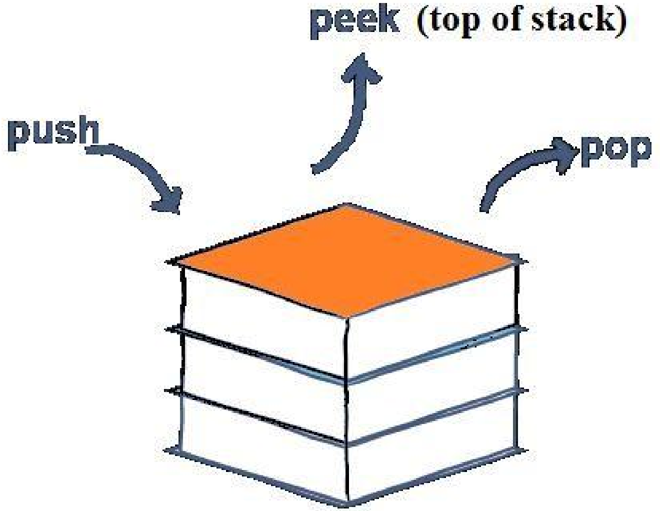
- *An array is a* ***data structure for storing more than one data item that has a similar data type****.*

- *The items of an array are allocated at adjacent memory locations.*

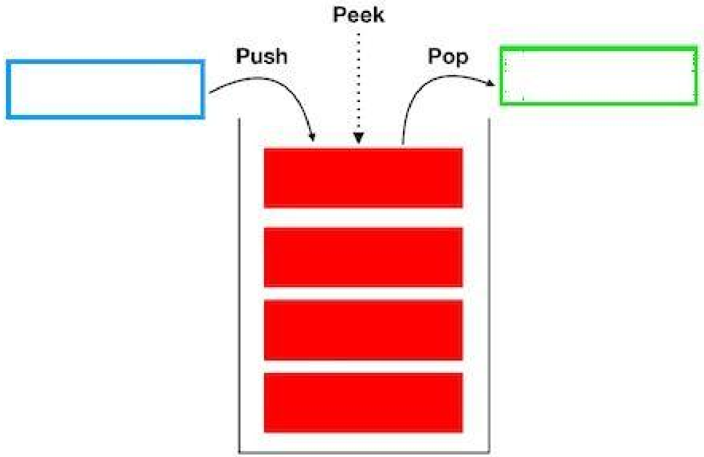


Stack

Stack is a linear data structure which follows a particular order in which the operations are performed. The order may be ***LIFO(Last In First Out) or FILO(First In Last Out)***



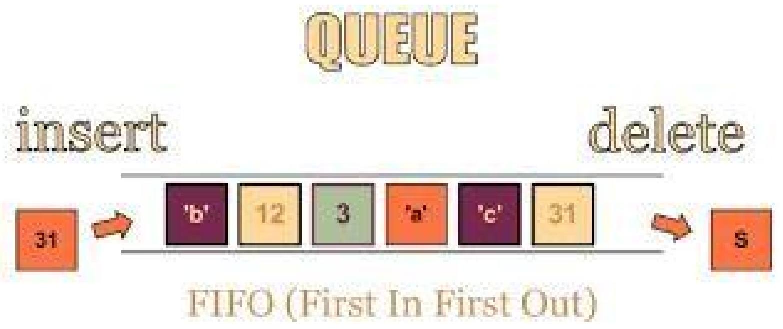
Real Life Stack



Stack Data Structure

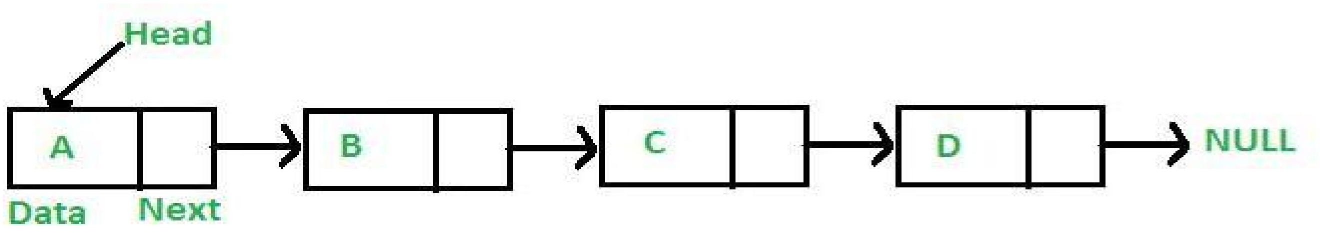
Queue

A Queue is a linear structure which follows a particular order in which the operations are performed. The order is **First In First Out (FIFO).** *A good example of a queue is any queue of consumers for a resource where the consumer that came first is served first.*



Linked List

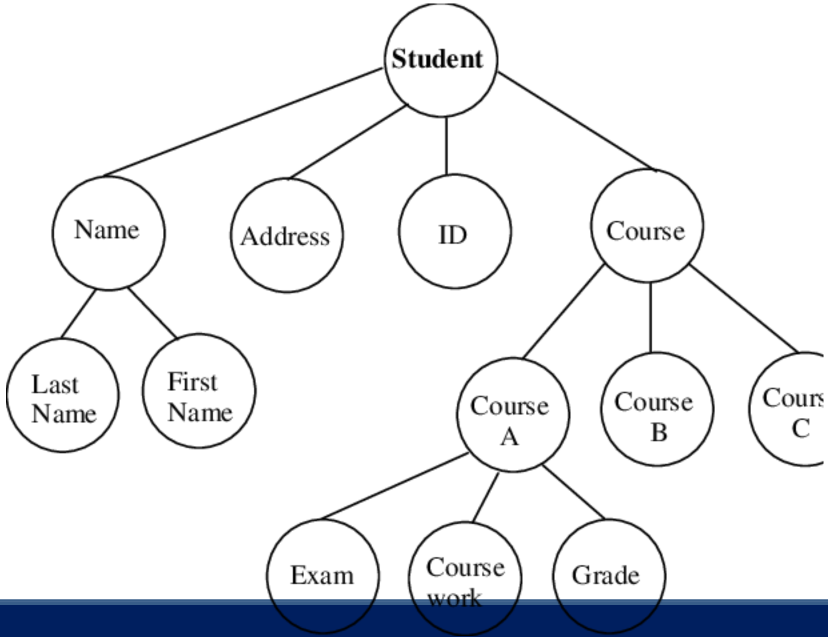
A linked list is a linear data structure, in which the elements are not stored at contiguous memory locations. The elements in a linked list are linked using pointers as shown in the below image:

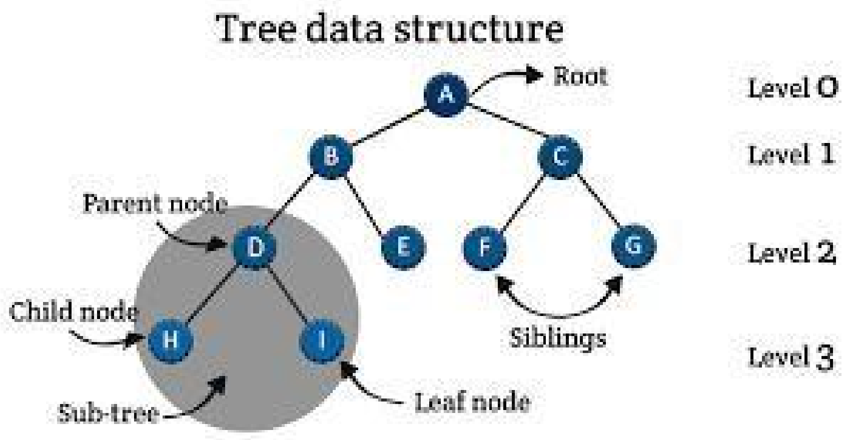


In simple words, a linked list consists of nodes where each node contains a data field and a reference(link) to the next node in the list.

Tree

Data frequently contain a hierarchical relationship between various elements. The data structure which reflects this relationship is called a rooted tree graph or a tree.

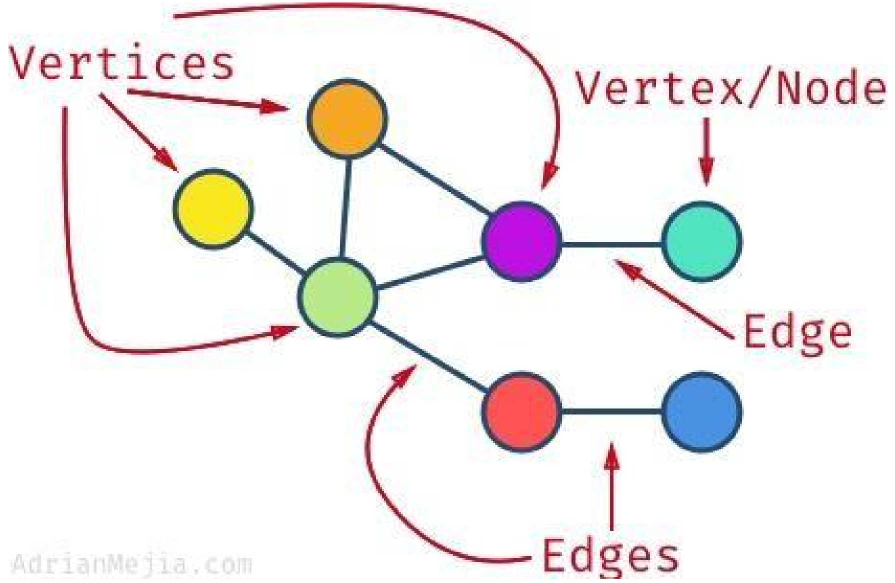




Graph

- *A Graph is a non-linear data structure consisting of nodes and edges. The nodes are sometimes also referred to as vertices and the edges are lines or arcs that connect any two nodes in the graph.*

- *A Graph consists of a finite set of vertices(or nodes) and set of Edges which connect a pair of nodes.*



- **Linear and Non- Linear Data Structure**

- In a linear data structure, the data items are arranged in a linear sequence. Example is array.
- n a non-Linear data structure, the data items are not in a sequence. Example is tree.
- homogeneous and Non- homogenous Data Structure In homogeneous Structure, all the elements are of same type. Example is arrays.
- In Non-homogeneous structure, the elements may or may not be of same type.
- Example is records.

*S*tatic and Dynamic Data Structure

- Static structures are ones whose sizes and structures, associated memory location are fixed at compile time.

- Dynamic structures are ones which expand or shrink as required during the program execution and there associated memory location change.

Data Structure Operations

Traversing: Traversing a Data Structure means to visit the element stored in it. This can be done with any type of DS.

Searching: Searching means to find a particular element in the given data-structure. It is considered as successful when the required element is found.

Insertion: It is the operation which we apply on all the data-structures. Insertion means to add an element in the given data structure.

Deletion: It is the operation which we apply on all the data-structures. Deletion means to delete an element in the given data structure.

Sorting: Sorting means arranging the data either in ascending or on decending.

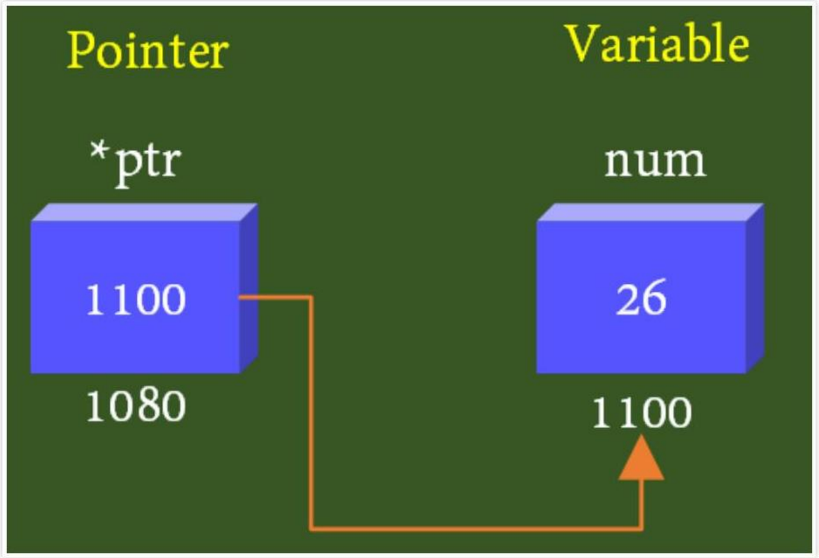
Pointers

->**Pointer is a variable which can hold the address of another variable**

->An alternative method to access the content of a memory location

1) Direct method

int num;
num=26;
printf("the content of the num is %d", num);
Printf("the address of the num is %x", &num); .



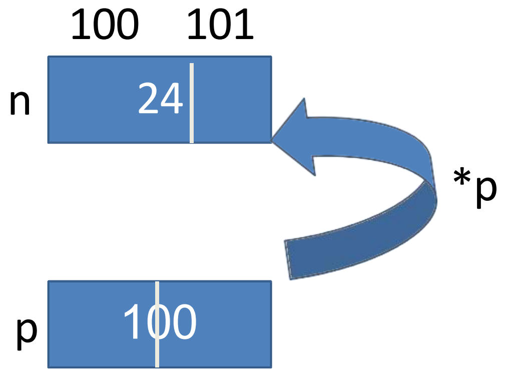
2. Indirect method (by using a pointer)

a. Creation of a pointer (declaration)

b. Assigning the created pointer the address

c. De-referencing the pointer access to the data

```
int n;
int *p;
n=24;
p=&n;
printf("%d", *p);
```



Output: 24

Syntax for declaring a pointer

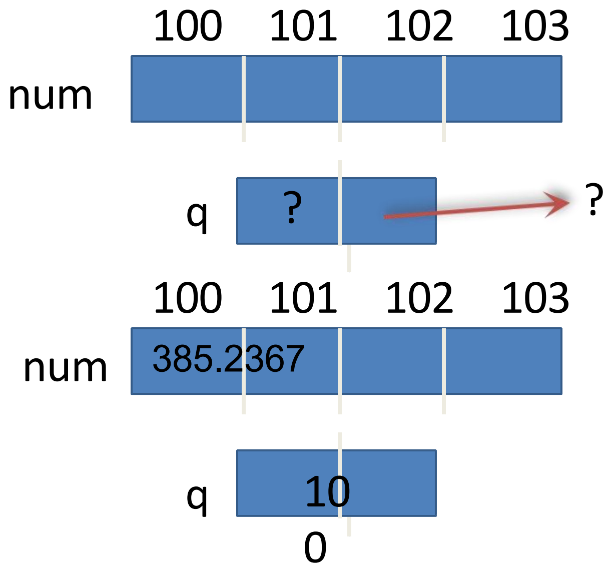
datatype * variable;

Ex: int *p;

float *q;

double *r;

```
float num;

float *q;

num=385.2367;

q=&num;
```



Dangling pointers: Created pointers are not pointing to any particular variable

Pointer can be declared and initialized in the same line

```
int a;                      int a=10;
a=10;
            or

int *p;                     int *p=&a;
p=&a;
```

In indirect method * is used for 2 purpose

1) To create a pointer

   int *p;

2) To de reference a pointer

   *p;

In C language * is used for 3 purpose

1) To create a pointer

   int *p;

2) To de reference a pointer

   *p;

3) To multiply 2 variables

   a * b;

1.

```
int n=5;
int p;
p=&n;
printf("%d", *p);
```

2.

```
int n=5;
int *p;
p=n;
printf("%d", *p);
```

3.

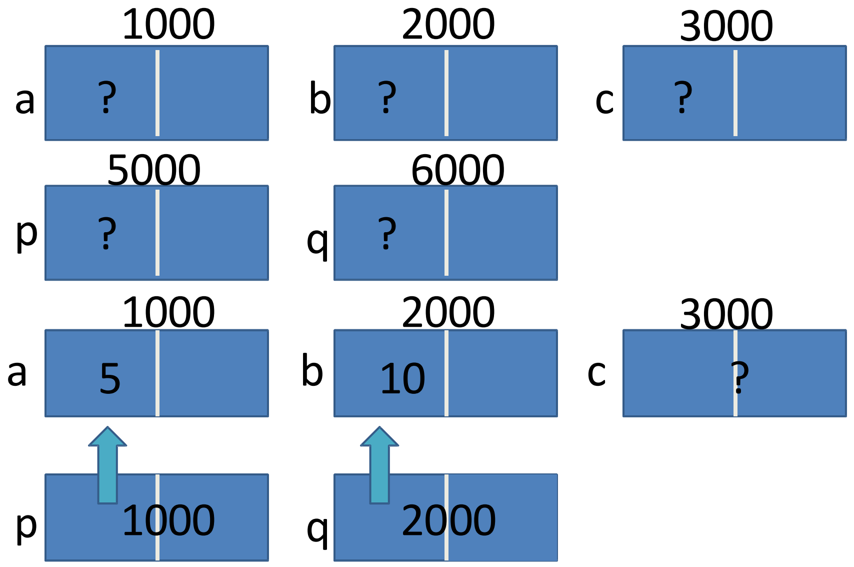
```
int n=5;
int *p;
p=&n;
printf("%d", p);
```

4.

```
int n=5;
float *p;
p=&n;
printf("%d", *p);
```

Accessing variables through pointer

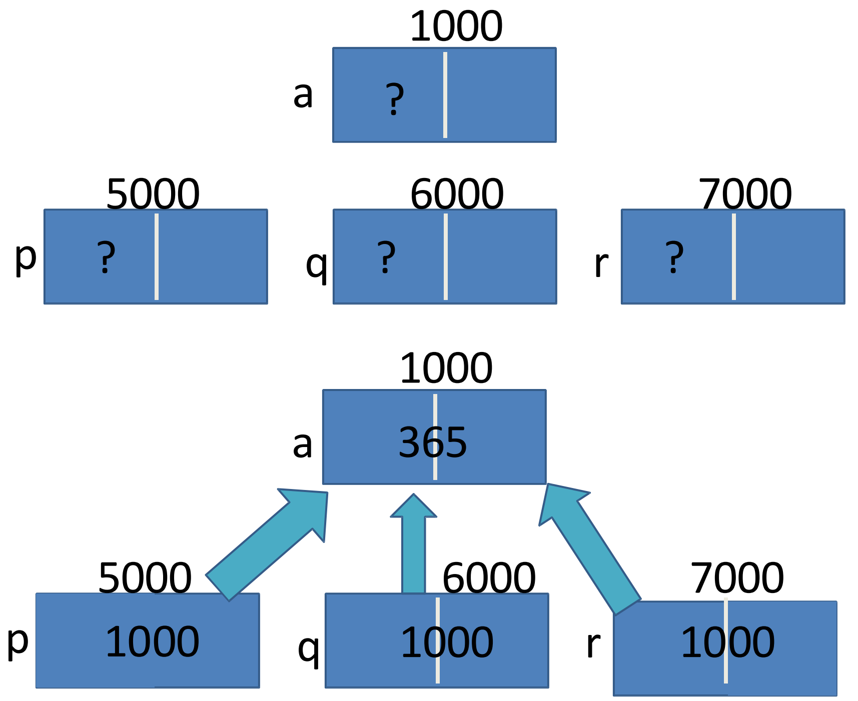
```
#include <stdio.h>
void main()
{
    int a,b,c;
    int *p, *q;
    a=5;
    b=10;

    p=&a;
    q=&b;
    c=*p+*q;
    printf("c is:%d",c);
}
```



Output:
c is: 15

Can there be more than one pointer to a variable?

```
#include <stdio.h>
void main()
{
    int a;
    int *p,*q,*r;

    a=365;

    p=&a;
    q=&a;
    r=&a;

    printf("the value of a is:%d\n", a);
    printf("the value of p is:%d\n",*p);
    printf("the value of q is:%d\n",*q);
    printf("the value of r is:%d\n",*r);
}
```

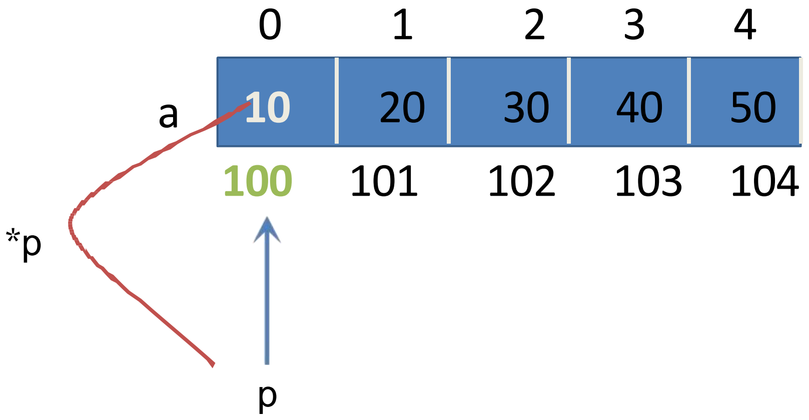


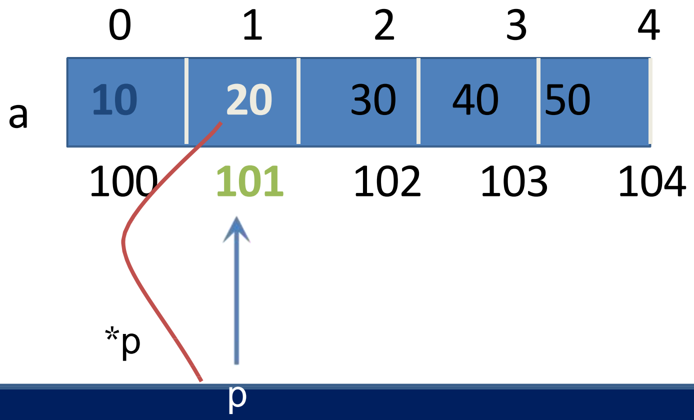
Difference between pointer variable and normal variable

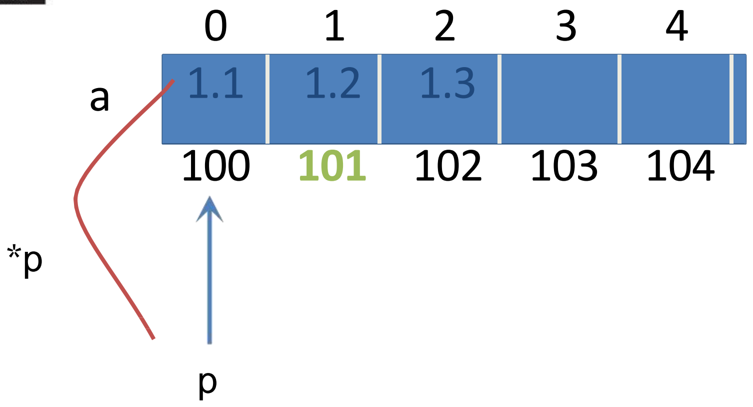
Pointer Variable	Normal Variable
1. Pointer Variable holds the address	1. A normal variable holds data
2. int *p;	2. int a;
3. We must de-reference a pointer to access data	3. There is no need to de-refernce a normal variable to access data

Pointers and Arrays

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


int *p;

p=&a[0]; or p=a;

for(int i=0; i<4; i++)
  {
      printf("%d\t", *p);
      p++;
  }
```

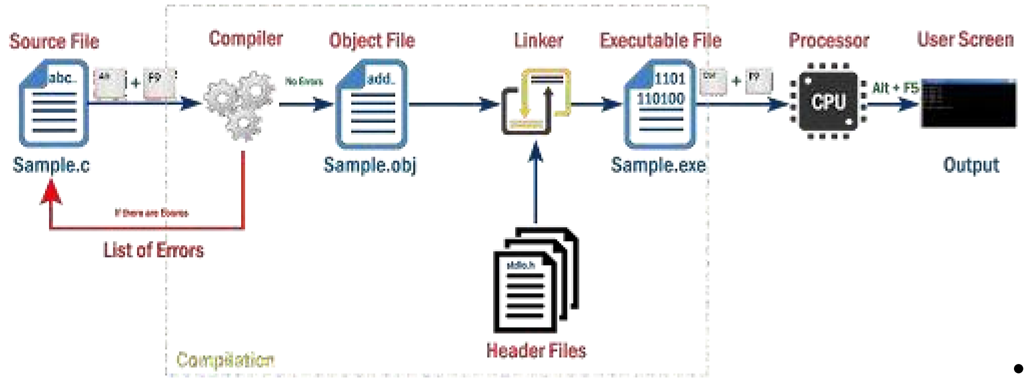






sum = sum + *(p+i)
Sum=0.0+1.1=1.1
Sum=1.1+1.2=2.3
sum=2.3+1.3=3.6

Dynamic memory allocation



- Conversion from HLL to MLL
- Decision to allocate memory to variable

Compilation

Execution

- Execution of machine level instructions
- Decision to allocate memory to variable

Static Memory Allocation

Dynamic Memory Allocation

- **Static Memory Allocation**
- Wastage of memory
- Reediting is a time consuming process

Dynamic Memory Allocation

- malloc()
- calloc()
- realloc()
- free()

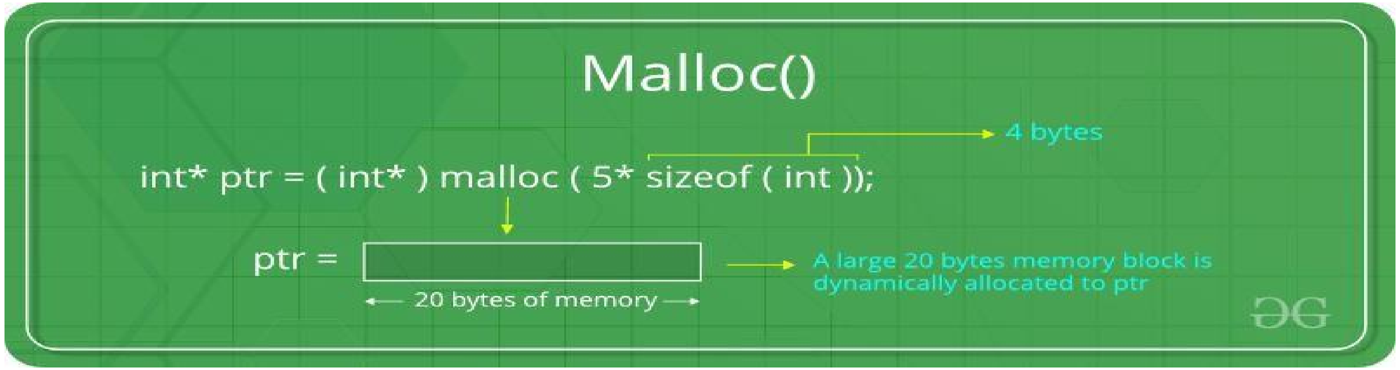
ATME College of Engineering

malloc()

- malloc stands for Memory allocation
- General form of memory allocation using malloc is,

datatype *ptr = (datatype *) malloc(RequiredAmountOfMemory * sizeof(datatype));

If malloc() is unable to find the required amount of memory, it returns **NULL**


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

Example

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

- A memory space equivalent to 100 times the size of an int bytes is reserved
- The address of the first byte of the allocated memory is assigned to the pointer p of type int
- On successful allocation, the function returns the address of first byte of allocated memory.
- Since address is returned, the return type is a void pointer. By type casting appropriately we can use it to store integer, float etc.

- cptr = (char *) malloc (20);

  - Allocates 20 bytes of space for the pointer cptr of type char

- sptr = (struct stud *) malloc(10*sizeof(struct stud));

- Allocates space for a structure array of 10 elements. sptr
  - points to a structure element of type struct stud

- **Always use sizeof operator to find number of bytes for a data type, as it can vary from machine to machine**

```
void main()
{
    int n,i;
    printf("enter the number of
elements\n");
    scanf("%d",&n);

    int *p = (int *)malloc(n*sizeof(int));

    if(p == NULL)
    {
        printf("enough memory not
available");
        exit(0);
    }
 printf("enter array elements\n");
    for(i=0; i<n; i++)
    {
        scanf("%d", p+i);
    }


printf("array elements are\n");
    for(i=0; i<n; i++)
    {
        printf("%d\t", *(p+i));
    }
}
```

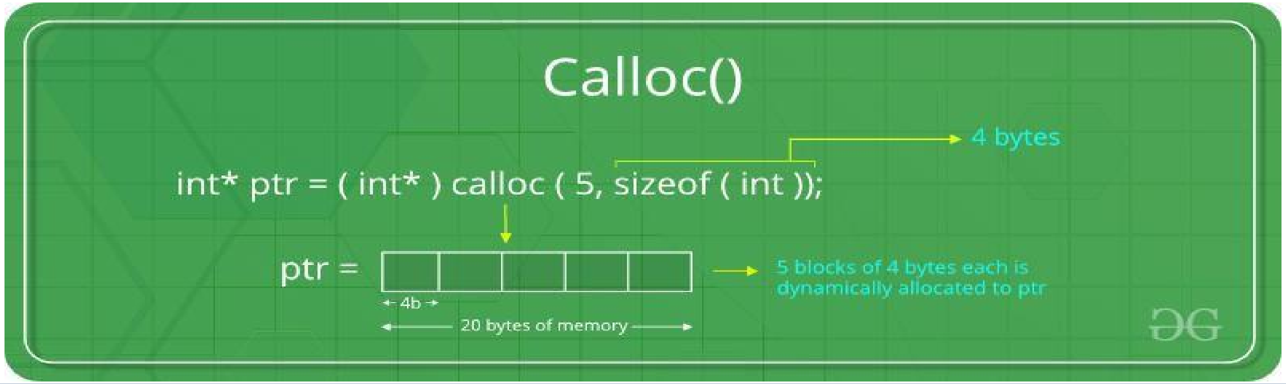
calloc()

- calloc stands for Contiguous allocation of multiple blocks.

- General form of memory allocation using calloc is,

datatype *ptr = (datatype *) calloc(n, sizeof(datatype));

If calloc() is unable to find the required amount of memory, it returns **NULL**



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

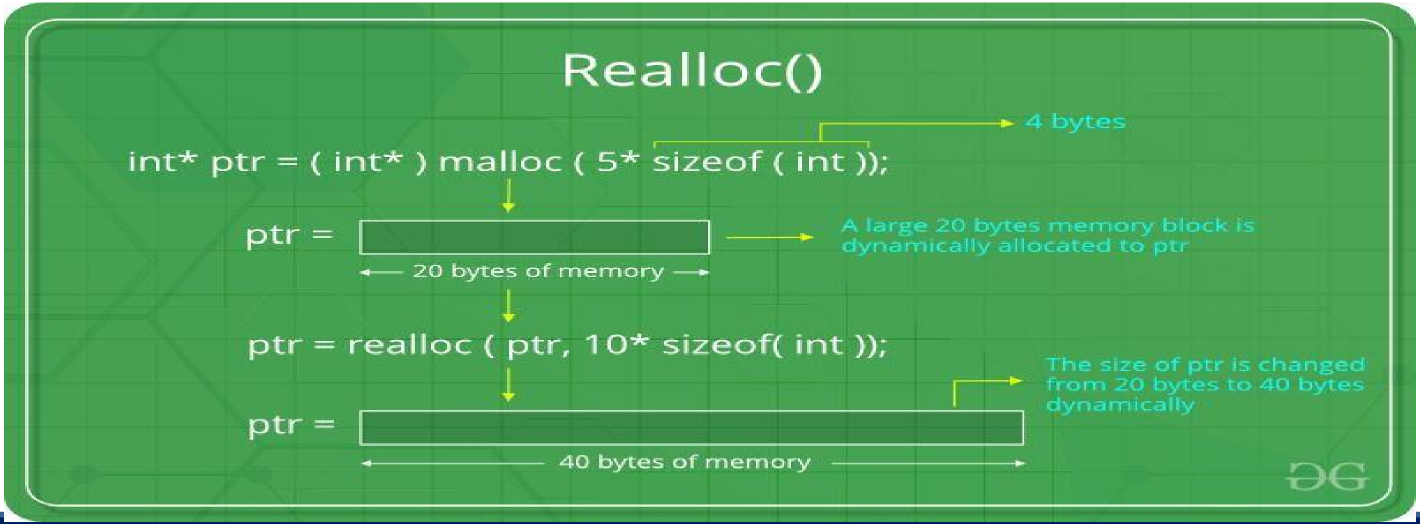
```
void main()
{
    int n,i;
    printf("enter the number of
elements\n");
    scanf("%d",&n);
    int *p = (int *)calloc(n,sizeof(int));

    if(p == NULL)
    {
        printf("enough memory not
available");
        exit(0);
    }
printf("enter array elements\n");
    for(i=0; i<n; i++)
    {
        scanf("%d", p+i);
    }
    printf("array elements are\n");
    for(i=0; i<n; i++)
    {
        printf("%d\t", *(p+i));
    }
}
```

realloc()

- realloc stands for re allocation

- General form of memory allocation using realloc is,

ptr = (datatype *) realloc(p, newsize*sizeof(datatype));


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

- realloc() changes the size of the block by extending or deleting the memory at
- the end of the block.
- If the existing memory can be extended, ptr value will not be changed
- If the memory cannot be extended, this function allocates a completely new
- block and copies the contents of existing memory block into new memory
- block and then deletes the old memory block.

```
void main()
{
    int n,i,new;
    printf("enter the number of elements\n");
    scanf("%d",&n);
    int *p = (int *)malloc(n*sizeof(int));
    if(p == NULL)
    {
        printf("enough memory not available");
        exit(0);
    }
    printf("enter array elements\n");
    for(i=0; i<n; i++)
    {
        scanf("%d", p+i);
    }
    printf("array elements are\n");
    for(i=0; i<n; i++)
    {
        printf("%d\t", *(p+i));
    }
  printf("\nenter the new number of elements\n");
    scanf("%d",&new);
    p=(int *)realloc(p,new*sizeof(int));
    printf("enter array elements\n");
    for(i=0; i<new; i++)
    {
        scanf("%d", p+i);
    }
    printf("array elements are\n");
    for(i=0; i<new; i++)
    {
        printf("%d\t", *(p+i)); }}
```

free()

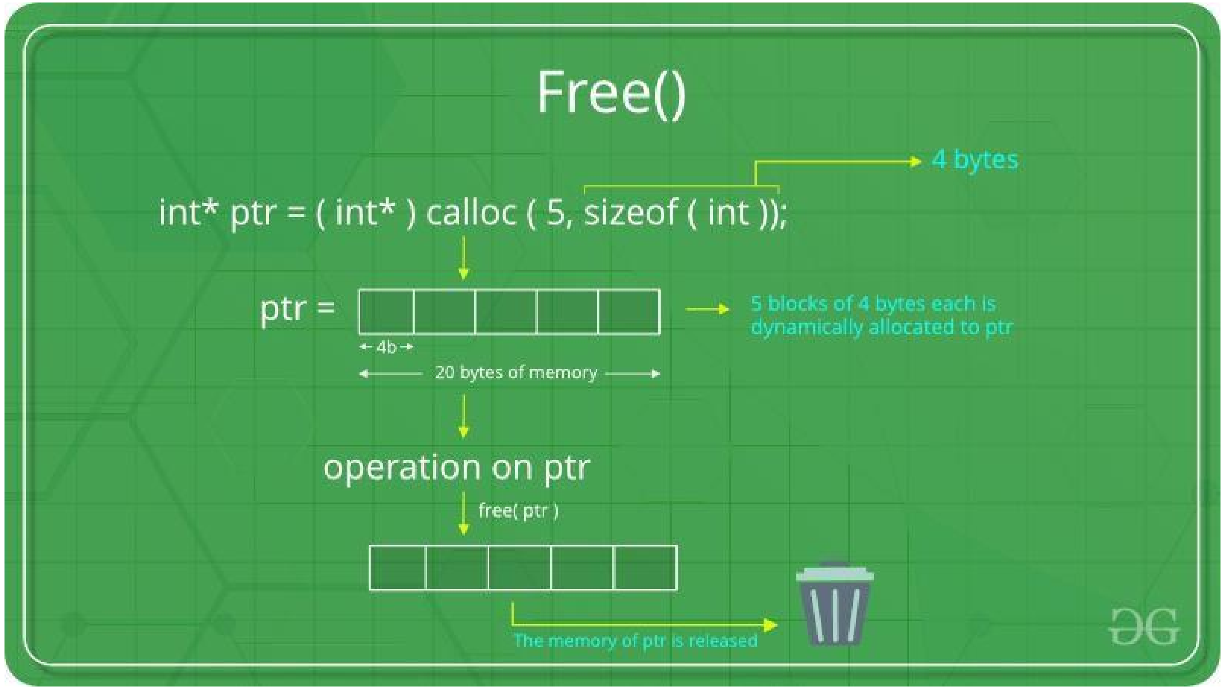
Dynamically allocated memory form calloc() or malloc() should be freed(released) using free().

General format:

void free(void *ptr);

Or

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

```
void main()
{
    int n,i;
    printf("enter the number of
elements\n");
    scanf("%d",&n);
    int *p = (int
*)malloc(n*sizeof(int));
    if(p == NULL)
    {
        printf("enough memory
not available");
        exit(0);
    }
    printf("enter array
elements\n");
for(i=0; i<n; i++)
    {
        scanf("%d", p+i);
    }
    printf("array elements are\n");
    for(i=0; i<n; i++)
    {
        printf("%d\t", *(p+i));
    }
    free(p);
}
```

Arrays

Tree

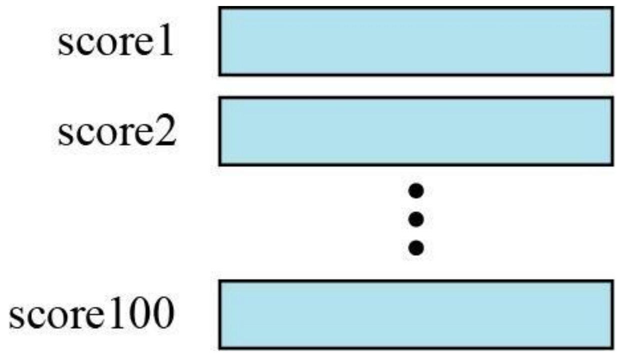
Array of trees

Student

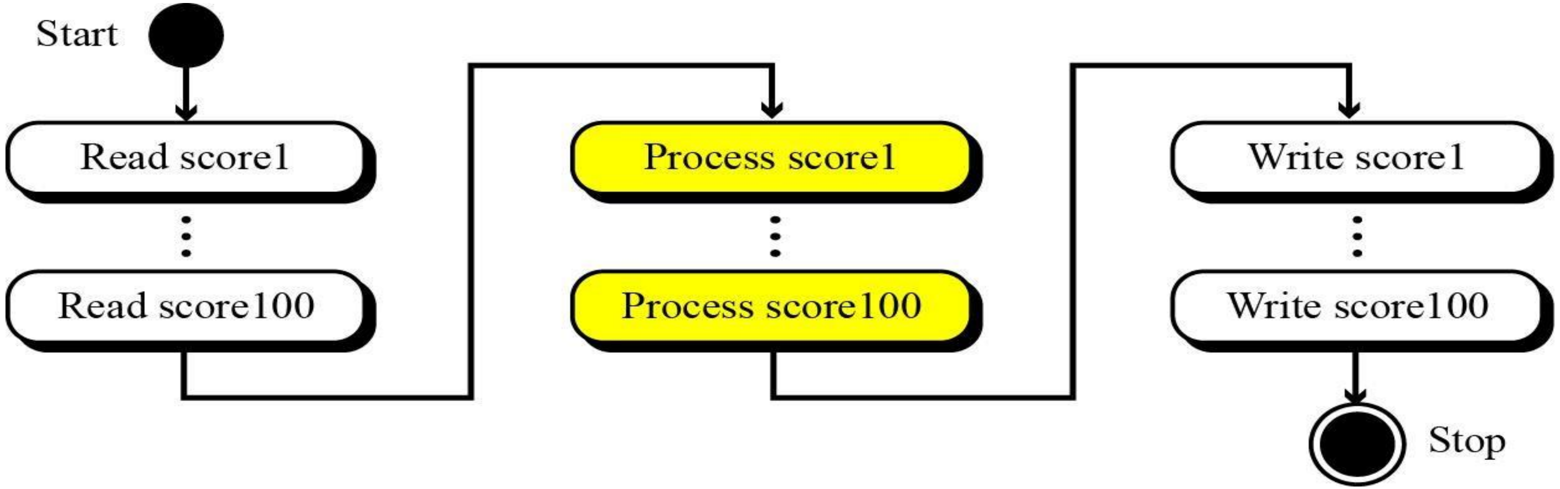
Array of students

- **Array** is a container which can hold a fix number of items and these items should be of the same type. Most of the data structures make use of arrays to implement their algorithms. Following are the important terms to understand the concept of Array.
- **Element** − Each item stored in an array is called an element.
- **Index** − Each location of an element in an array has a numerical index, which is used to identify the element.
- Array Representation - Arrays can be declared in various ways in different languages.

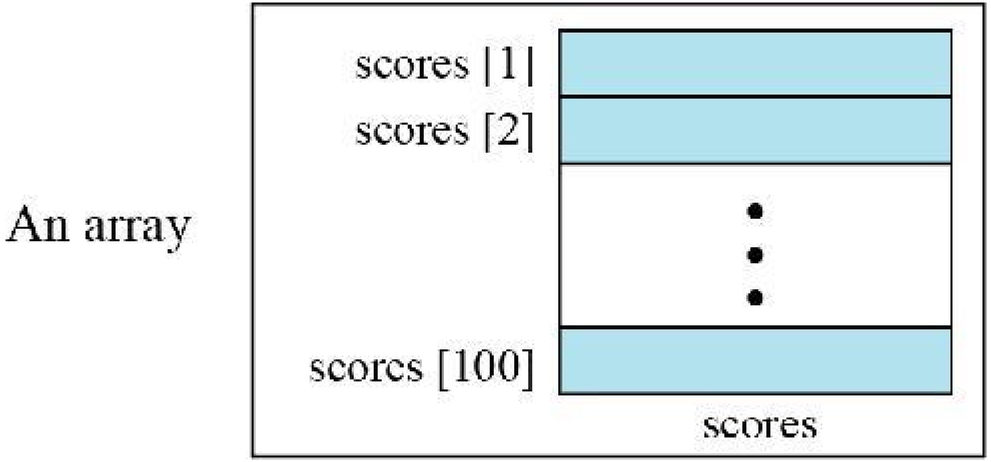
- Imagine that we have 100 scores. Weneed to read them, process them and print them. Wemust also keep these 100 scores in memory for the duration of the program. Wecan define a hundred variables, each with a different name.



But having 100 different names creates other problems. Weneed 100 references to read them, 100 references to process them and 100 references to write them.

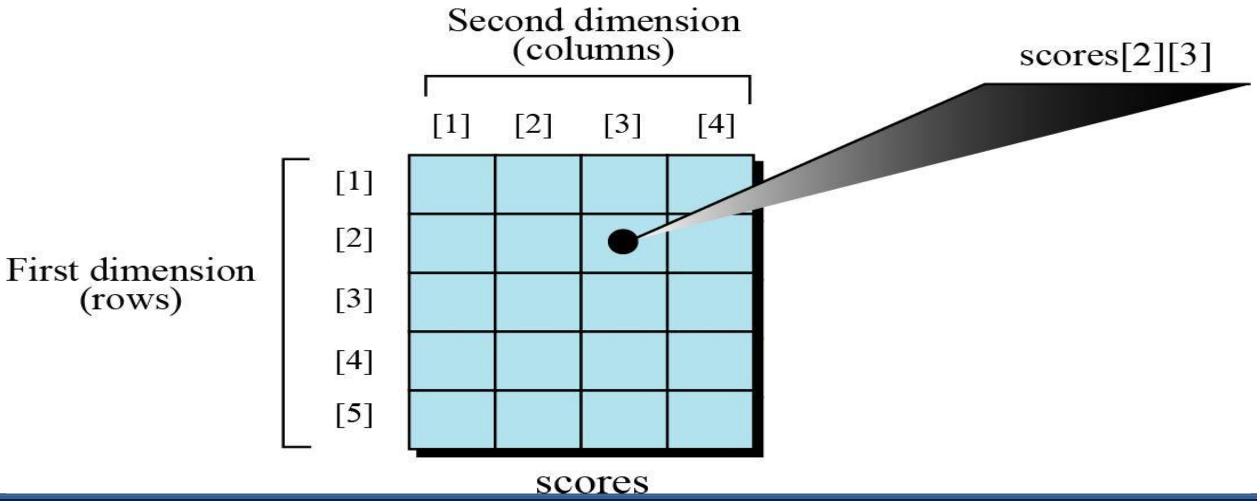


- An array is a sequenced collection of elements, normally of the same data type, although some programming languages accept arrays in which elements are of different types. Wecan refer to the elements in the array as the first element, the second element and so forth, until we get to the last element.



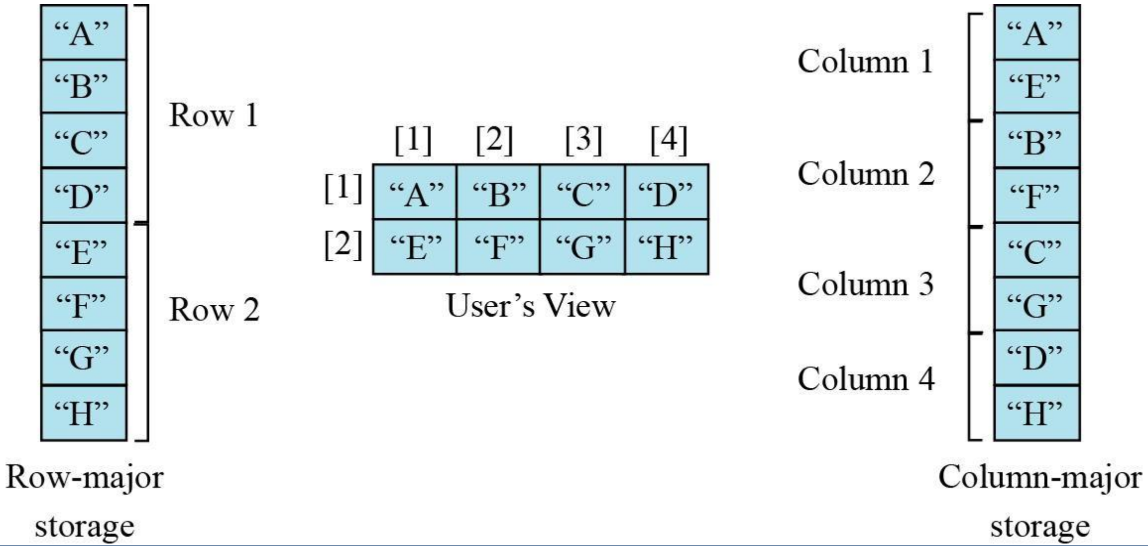
Multi-dimensional arrays

- The arrays discussed so far are known as one-dimensional arrays because the data is organized linearly in only one direction. Many applications require that data be stored in more than one dimension. Figure shows a table, which is commonly called a two-dimensional array



Memory Layout

- The indexes in a one-dimensional array directly define the relative positions of the element in actual memory. Figure shows a two-dimensional array and how it is stored in memory using row-major or column-major storage. Row- major storage is more common.



Dynamically allocated Arrays

```
#include <stdio.h>
#include <stdlib.h>
void main()
{
    int r = 3, c = 4, i; //Taking number of Rows and Columns
    int *ptr; //creating pointer
    ptr = (int *)malloc((r * c) * sizeof(int)); //Dynamically Allocating Memory (12*2=24)
    for(i = 0; i < r * c; i++)
    {
        ptr[i] = i + 1; //Giving value to the pointer and simultaneously printing it.
        printf("%d ", ptr[i]);
        if ((i + 1) % c == 0)
        {
            printf("\n");
        }
    }
    free(ptr);
}
```

35		23	45	20	100	70
35.5		11.5	34.3	15.9	90.6	46.7
m		a	t	u	w	r

Array means, a series of entities or a sequence of entities of the same type (homogeneous).
In C language entities can be char, int, float and double type data

Declaration of 1-D array

Syntax:

datatype arrayname[size];

Ex: int a[5];

float b[5];

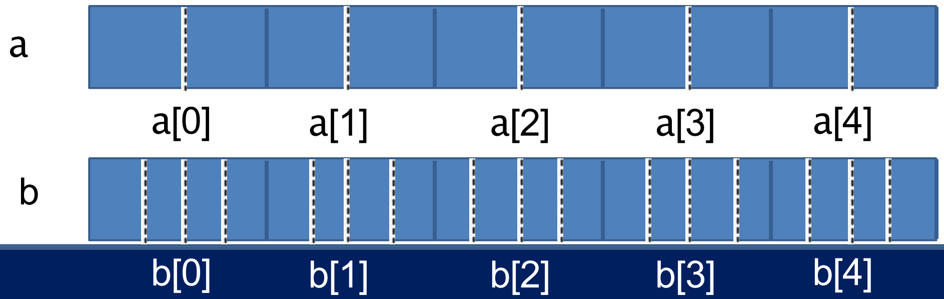
A declaration statement tells the compiler,

->data type of the array

->name of the array

->size of the array

Compiler then allocate memory depending upon the declaration.



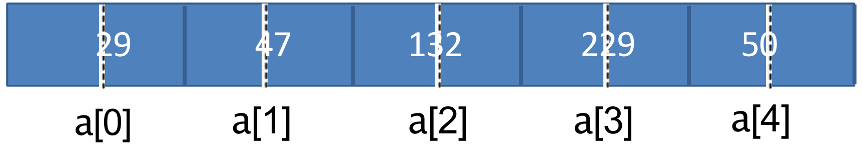
Initialization of 1-D array

1) Direct Initialization (Compile time initialization)

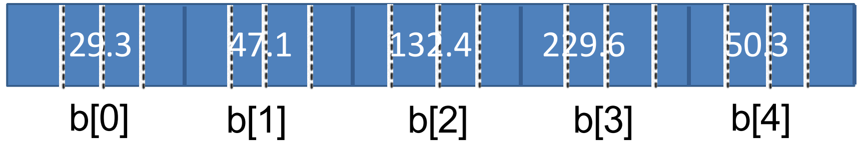
2) Initialization using a for loop (Run time initialization)

1) Direct Initialization : Mentioning array size is not compulsory.

Ex: int a[5] = {29, 47, 132, 229, 50 };



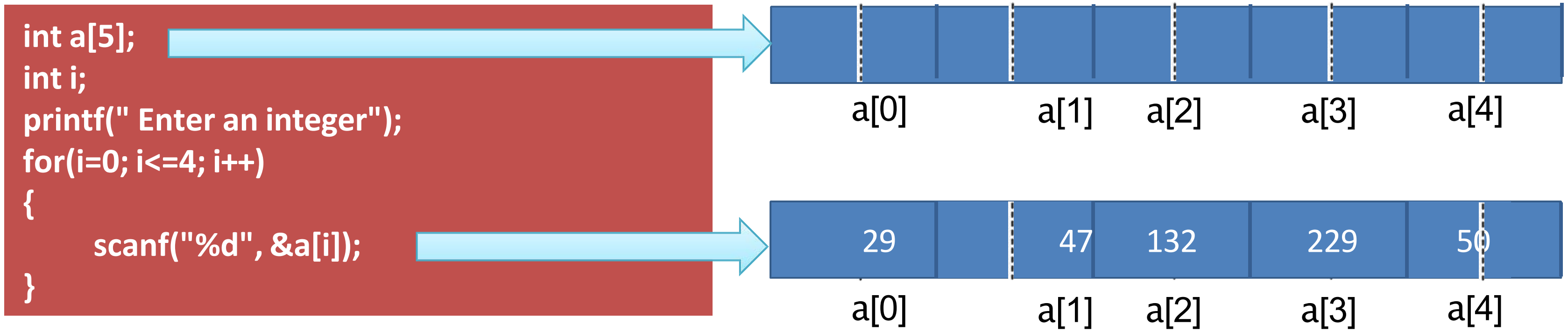
Ex: float b[5] = {29.3, 47.1, 132.4, 229.6, 50.3 };



2) Initialization using a for loop:

->Using a for loop to initialize the array blocks

->Mentioning array size is compulsory

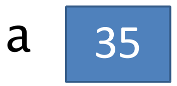


```
int a[5];
int i;
printf(" Enter an integer");
for(i=0; i<=4; i++)
{
       scanf("%d", &a[i]);
}
```

How do we store single integer in memory and how do we store array of integers in memory?

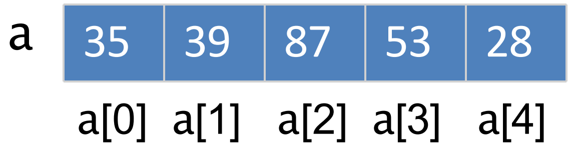
To store single integer,

int a = 35;



To store array of integers,

int a[5] = {35,39,87,53,28};



Note: Array is a Indexed Data Structure

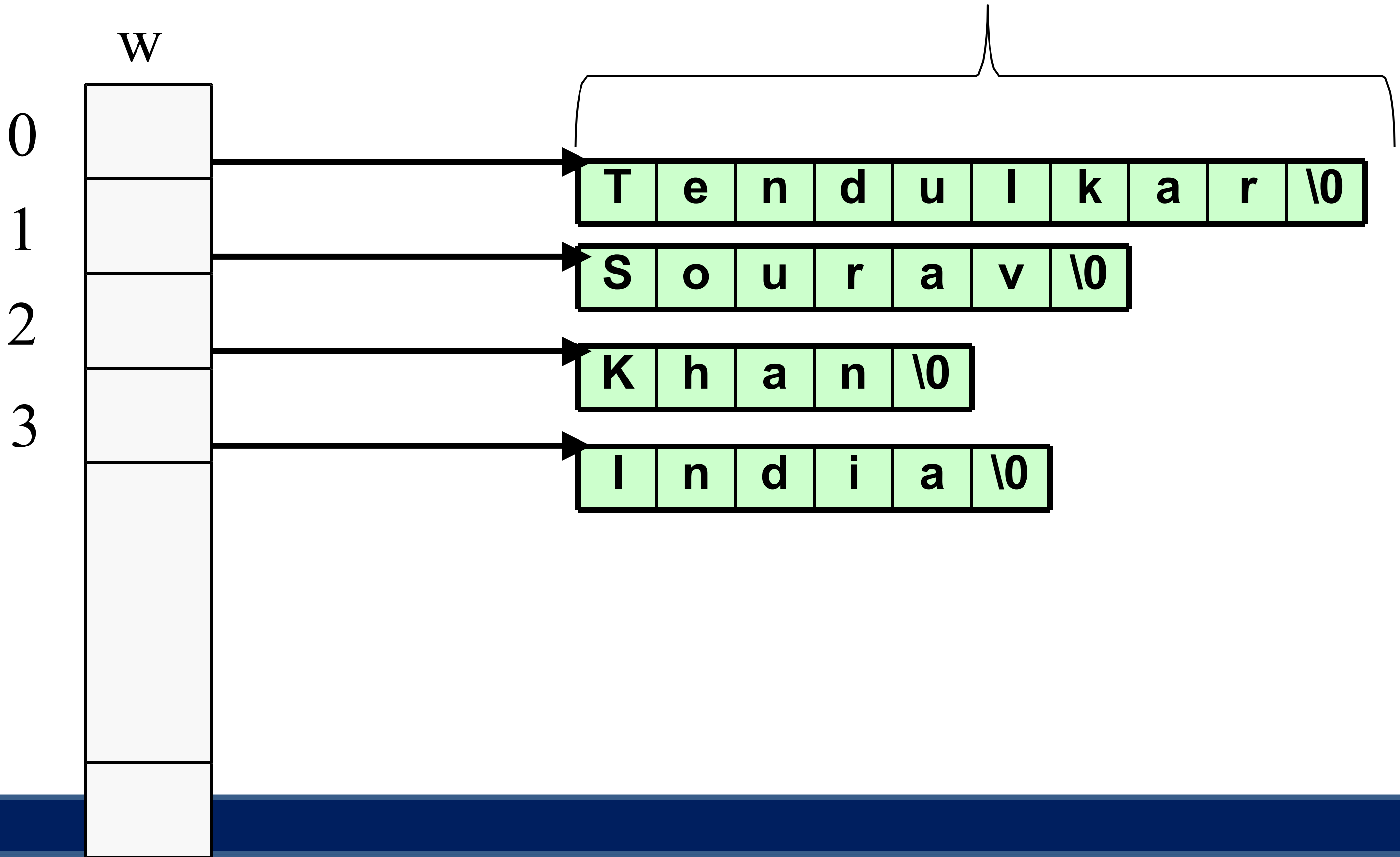
Total size of the array=size of the array * number of bytes per block

=5*2

=10 bytes

```
#define N20
#define M 10
int main()
{
char word[N], *w[M];
int i, n;
scanf("%d",&n);
for (i=0; i<n; ++i) {  scanf("%s", word);
w[i] = (char *) malloc
((strlen(word)+1)*sizeof(char));
strcpy (w[i], word) ;
}
for (i=0; i<n; i++)
{ printf("w[%d] = %s
\n",i,w[i]); return 0;
}
```

```
4
Tendulkar
Sourav  Khan
Khan
 India
w[0] = Tendulkar
w[1] = Sourav
w[2] = Khan
```


w
0
1
2
3
T e n d u l k a r \0
S o u r a v \0
K h a n \0
I n d i a \0

1. A car manufacturing company uses an array car to record number of cars sold each year starting from 1965 to 2015

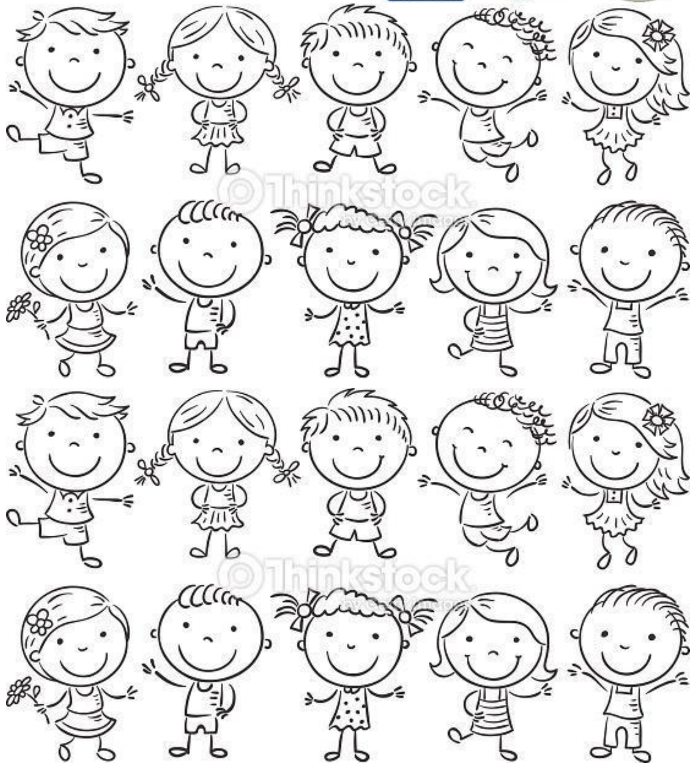
i) Find the total number of years(elements)

ii) Suppose base address = 500, word length (w) = 4, find address of car[1967], car[1969] and car[2015]

Two Dimensional Array

Student

1-D array of Students


2-D array of Student

Multidimensional Arrays

Two-dimensional arrays are called matrices in mathematics and tables in business applications. There is a standard way of drawing a two-dimensional m x n array A where the elements of a form a rectangular array with m rows and n columns and where the element A[J, K] appears in row J and column K.

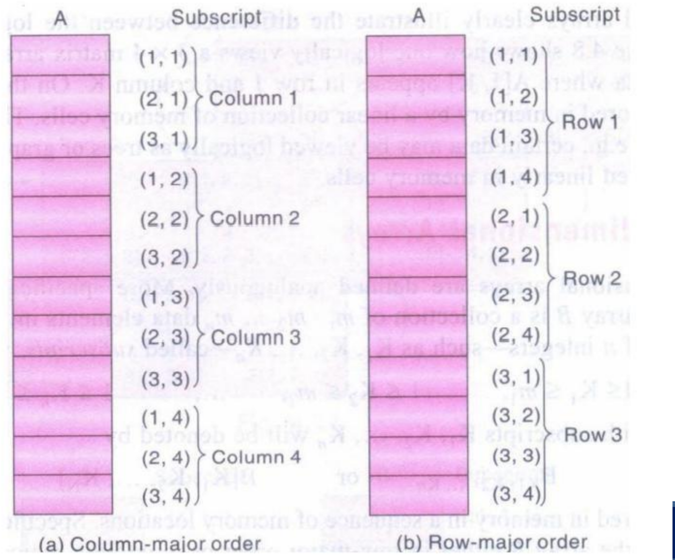
$$
\begin{array}{cc}
 & \text{Columns} \\
 & \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\
\text{Rows} \begin{array}{c} 1 \\ 2 \\ 3 \end{array} & \begin{bmatrix} A[1,1] & A[1,2] & A[1,3] & A[1,4] \\ A[2,1] & A[2,2] & A[2,3] & A[2,4] \\ A[3,1] & A[3,2] & A[3,3] & A[3,4] \end{bmatrix}
\end{array}
$$

Fig. *Two-Dimensional 3 × 4 Array A*

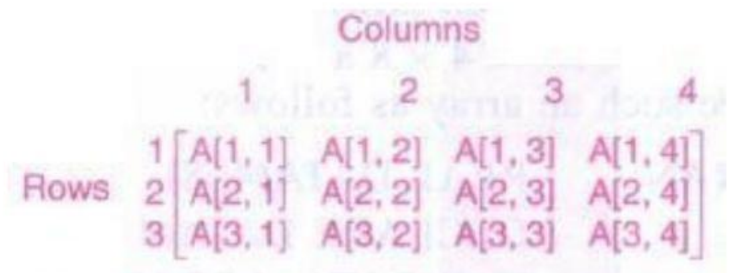
Representation of Two-Dimensional Arrays in Memory

Let A be a two-dimensional m x n array. Although A is pictured as a rectangular array of elements with m rows and n columns, the array will be represented in memory by a block of m . n sequential memory locations.

The programming language will store the array A either (1) column by column, is called **column-major order**, or (2) row by row, in **row-major order**.



(a) Column-major order (b) Row-major order



$$\begin{array}{c|c|cccc} & & & \text{Columns} & & \\ & & 1 & 2 & 3 & 4 \\ \hline & 1 & A[1,1] & A[1,2] & A[1,3] & A[1,4] \\ \text{Rows} & 2 & A[2,1] & A[2,2] & A[2,3] & A[2,4] \\ & 3 & A[3,1] & A[3,2] & A[3,3] & A[3,4] \end{array}$$

Fig. *Two-Dimensional 3 × 4 Array A*

(Column-major order) LOC(A[J, K]) = Base(A) + w[M(K - 1)+ (J - 1)]

(Row-major order) LOC(A[J, K]) = Base(A) + w[N(J - 1) + (K - 1)]

- A two dimensional array is represented as one-dimensional array of pointers where each pointer contains address of one-dimensional array.
- For example, consider the following declaration: int a[3][5];
- Recall that address of [i][j]-th element is found by first finding the address of first element of i-th row, then adding j to it.

- Now think of a 2-d array of dimension [M][N] as M 1-d arrays, each with N elements, such that the starting address of the M arrays are contiguous (so the starting address of k-th row can be found by adding 1 to the starting address of (k-1)-th row)
- This is done by allocating an array p of M pointers, the pointer p[k] to store the starting address of the k-th row

ATME College of Engineering

Structures

int a;

float b;

Student:
Name
Age
Marks
USN

Tree:
Name
No. of branches
height

```
struct student
{
        char name[5];
        int age;
        int marks;
        char USN;
};
struct tree
{
        char name[5];
        int No.of branches;
        float height;
};
```

Structures

Structure provides a mechanism for the programmer to create his/her own data type called "**user defined date type**"

Keyword for creating structure

Name of the structure

```
struct student
{
        char name[20];
        int age;
        int marks;
        float height;
};
```

Structure members/data

Syntax

```
struct structure_name
{
        datatype  member variable 1;
        datatype  member variable 2;
        datatype  member variable 3;
        datatype  member variable 4;
};
```

Ex. 1

```
struct student
{
        char name[20];
        int age;
        int marks;
        float height;
};
```

Ex. 2

```
struct tree
{
        char name[5];
        int No.of
        branches;
        float height;
};
```

Structures

```
struct student
{
        char name[5];
        int age;
        int marks;
        float height;
};
```

```
struct tree
{
        char name[5];
        int No.of branches;
        float height;
};
```

To access members of a structure, a structure variable has to be created

```
struct student s1;
```

```
struct tree t1;
struct student
{
        char name[5];
        int age;
        int marks;
        float height;
};


struct student s1;
```

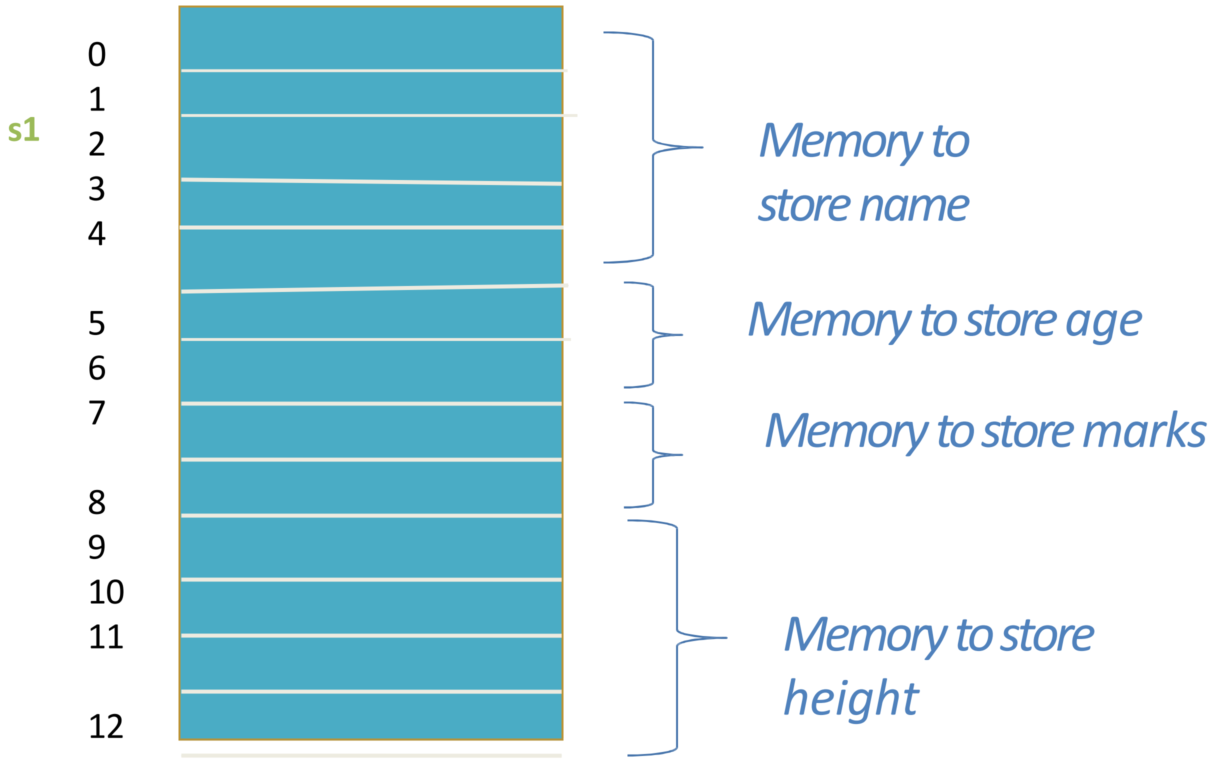
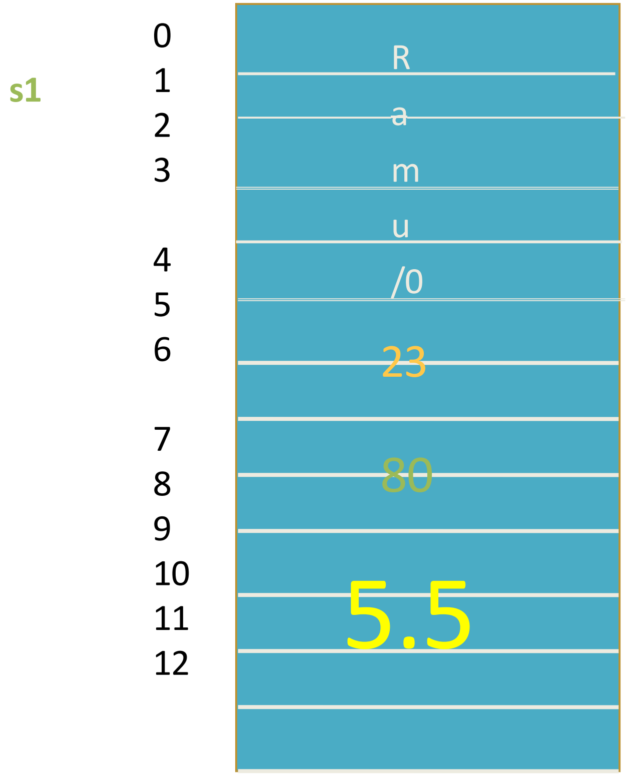


Fig: Memory for structure member

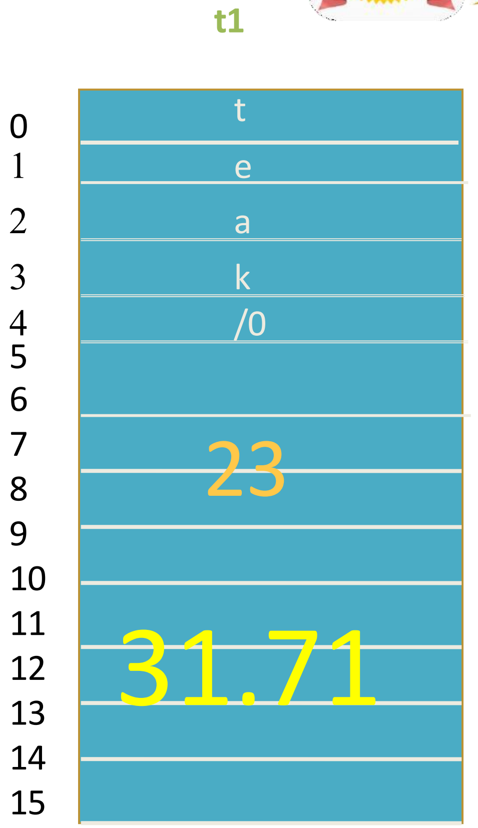
```
struct student
{
        char name[5];
        int age;
        int marks;
        float height;
};
struct student s1;

  s1.name="Ramu";
  s1.age=23;
  s1.marks=80;
  s1.height=5.5;
```

.(dot) -> Member access operator



```
struct tree
{
        char name[5];
        int noofbranches;
        float height;
};

struct tree t1;

  t1.name="teak";
  t1.noofbranches=23;
  t1.height=31.71;
```

.(dot) -> Member access operator



```
//program to read and display student details
using structures
#include <stdio.h>
struct student
{
   char name[20];
   int age;
   int marks;
   float height;
};
void main()
{
   struct student s1;
printf("Enter Student Details\n");
   printf("enter the name of the student\n");
   scanf("%s", s1.name);
printf("enter student age\n");
   scanf("%d", &s1.age);
   printf("enter student marks\n");
   scanf("%d", &s1.marks);
   printf("enter student height\n");
   scanf("%f", &s1.height);
   printf("Student details you entered:\n");
   printf("Student name is: %s\n",s1.name);
   printf("Student age is: %d\n",s1.age);
   printf("Student marks is: %d\n", s1.marks);
   printf("student height is: %.2f\n",s1.height);
}
```

//program to read and display tree details using structures

```
#include <stdio.h>
struct tree
{
   char name[20];
   float height;
   int noofbranches;
};
void main()
{
   struct tree t1;
   printf("enter the name of the tree\n");
scanf("%s", t1.name);
  printf("enter tree height\n");
  scanf("%f", &t1.height);

printf("enter number of noofbranches\n");
 scanf("%d", &t1.noofbranches);
 printf("the tree name is: %s\n",t1.name);
 printf("the tree height is: %f\n",t1.height);
 printf("noofbranches in the tree are:
%d\n",t1.noofbranches);
   }
```

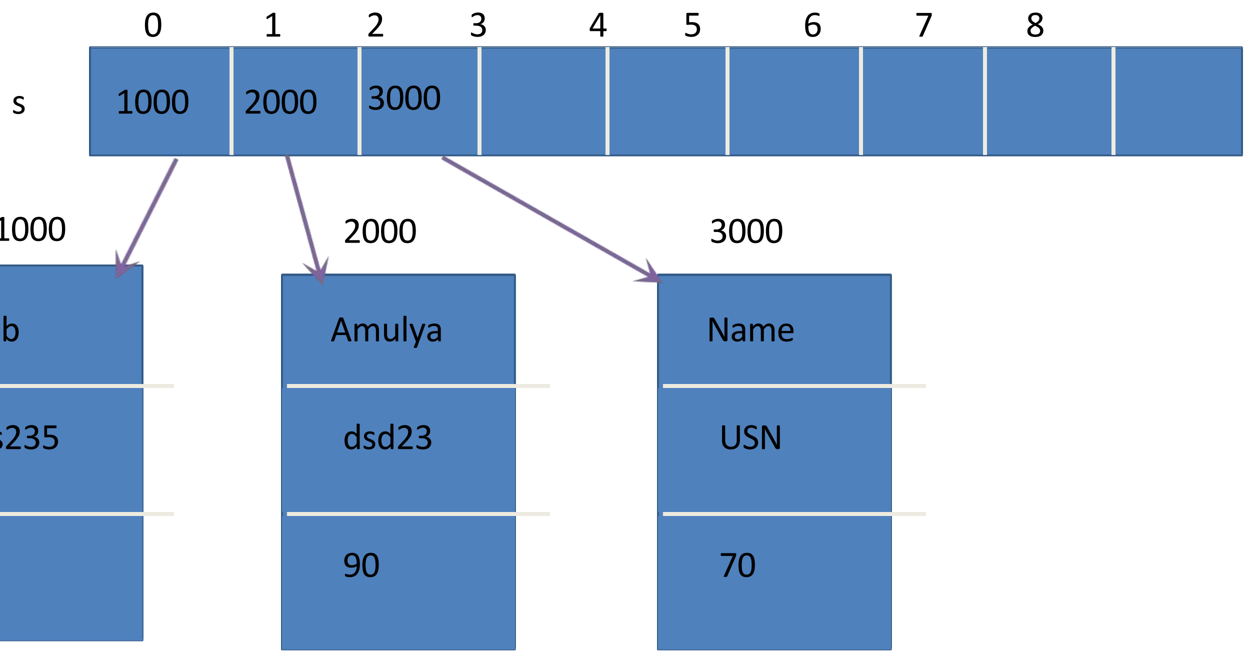
Array	Structure
Array is a collection of related data elements of same data type. (homogeneous data)	Structure is collection of logically related data elements of different data types. (heterogeneous data)
Array data are accessed using index	Structure data are accessed using structure name and dot operator
No key word is used to create array	Struct keyword is used create structure
Each element will be of same size	Size of the elements can be different

Structures and Functions

```
#include <stdio.h>
struct tree
{
   char name[20];
   float height;
   int noofbranches;
};
void disp(struct tree tr)
{
   printf("the tree name is: %s\n",tr.name);
   printf("the tree height is: %f\n",tr.height);
   printf("noofbranches in the tree are:
%d\n",tr.noofbranches);
 }
```

```
void main()
{
   struct tree t1;

   printf("enter the name of the tree\n");
   scanf("%s", t1.name);
   printf("enter tree height\n");
   scanf("%f", &t1.height);
   printf("enter number of noofbranches\n");
   scanf("%d", &t1.noofbranches);
   disp(t1);
}
```

Array of Structures



Array of Structures

```
#include <stdio.h>
struct tree
{
    char name[20];
    float height;
    int noofbranches;
};
void main()
{
    struct tree t1[2];

    for(int i=0; i<=1; i++)
    {
    printf("enter the name of the tree\n");
    scanf("%s", t1[i].name);
    printf("enter tree height\n");
    scanf("%f", &t1[i].height);
    printf("enter number of noofbranches\n");
    scanf("%d", &t1[i].noofbranches);
    }

    for(int i=0; i<=1; i++)
    {
    printf("the tree name is: %s\n",t1[i].name);
    printf("the tree height is: %f\n",t1[i].height);
    printf("noofbranches in the tree are:
%d\n",t1[i].noofbranches);
    }
}
```

Typedefing a Structures

- *The typedef is a keyword that is used to provide existing data types with a new name.*
- *The C typedef keyword is used to redefine the name of already existing data types.*

```
#include<stdio.h>
typedef struct
{
   char name[10];
   int usn;
}student;
void main()
{
   student s1;
   printf("enter student name:\n");
   scanf("%s", s1.name);
   printf("enter student usn:\n");
   scanf("%d", &s1.usn);
   printf("student name is %s",s1.name);
   printf("student usn is %d", s1.usn);
}
```

Different ways of writing a program using structure

```
#include<stdio.h>
struct student
{
   char name[10];
   int usn;
};
void main()
{
   struct student s1;
   printf("enter student name:\n");
   scanf("%s", s1.name);
   printf("enter student usn:\n");
   scanf("%d", &s1.usn);
   printf("student name is
%s\n",s1.name);
  printf("student usn is %d", s1.usn);
}
```

```
#include<stdio.h>
struct student
{
   char name[10];
   int usn;
}s1;
void main()
{
   printf("enter student name:\n");
   scanf("%s", s1.name);
   printf("enter student usn:\n");
   scanf("%d", &s1.usn);
   printf("student name is
%s\n",s1.name);
  printf("student usn is %d", s1.usn);
 }
```

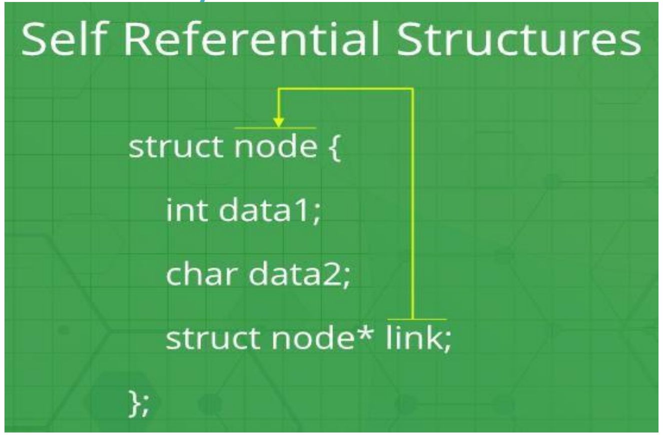
```
#include<stdio.h>
typedef struct
{
   char name[10];
   int usn;
}student;

void main()
{
   student s1;
   printf("enter student name:\n");
   scanf("%s", s1.name);
   printf("enter student usn:\n");
   scanf("%d", &s1.usn);
   printf("student name is
%s\n",s1.name);
  printf("student usn is %d", s1.usn);
}
```

```
#include<stdio.h>
struct address
{
    char city[20];
    int pin;
    char phone[14];
};
struct employee
{
    char name[20];
    struct address add;
};
void main ()
{
   struct employee emp;
   printf("Enter employee information?\n");
   scanf("%s %s %d %s",emp.name,emp.add.city, &emp.add.pin, emp.add.phone);
   printf("Printing the employee information....\n");
 printf("name:%s\nCity:%s\nPincode:%d\nPhone:%s",emp.name,emp.add.city,emp.add.pin,emp.add.phone)
 }
```

Self-Referential Structures

A self-referential structure is one in which one or more of its components is a pointer to itself. Self referential structures usually require dynamic storage management routines (malloc and free) to explicitly obtain and release memory.

```
typedef struct {

char data;
struct list *link ;

} list;
```


Self Referential Structures

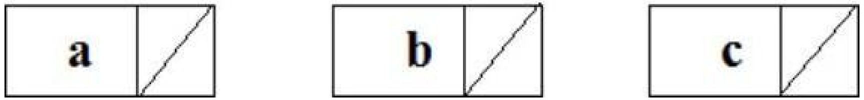
```
struct node {
    int data1;
    char data2;
    struct node* link;
};
```

Each instance of the structure list will have two components data and link.

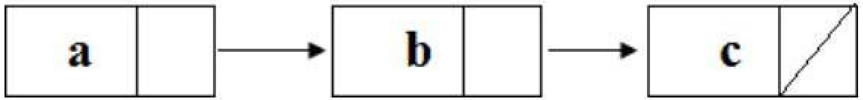
- Data: is a single character,
- Link: link is a pointer to a list structure. The value of link is either the address in memory of an instance of list or the null pointer.

Self-Referential Structures

```
list item1, item2, item3;
item1.data = 'a';
item2.data = 'b';
item3.data = 'c';
item1.link = item2.1ink = item3.link = NULL;
```



```
item1.link = &item2;
item2.1ink = &item3;
```



Union

A union is similar to a structure, it is collection of data similar data type or dissimilar.

Syntax:

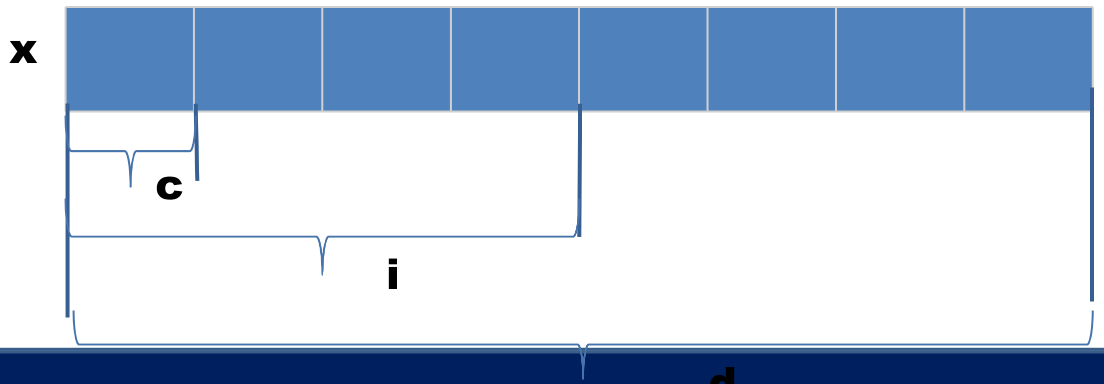
```
union tag_name{
          data_type member 1;
          data_type member 2;
          ...........................
          ...........................
          data_type member n;
}variable_name;
```

Union

```
typedef union
{
        int i;
        double d;
        char c;
}item;
```

```
union item
{
        int i;
        double d;
        char c;
};
```

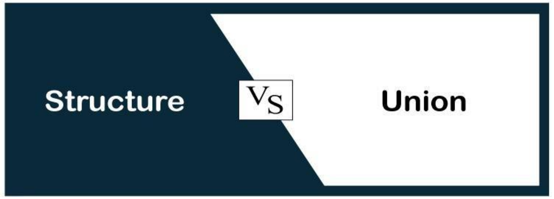
item x;

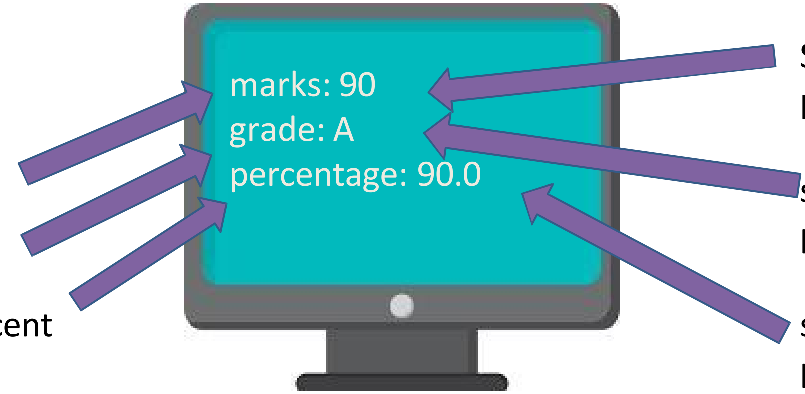


Difference between Structure and Union

	STRUCTURE	UNION
Keyword	The keyword **struct** is used to define a structure	The keyword **union** is used to define a union.
Size	When a variable is associated with a structure, the compiler allocates the memory for each member. The size of structure is **greater than or equal to the sum of sizes of its members.**	when a variable is associated with a union, the compiler allocates the memory by considering the size of the largest memory. So, size of **union is equal to the size of largest member.**
Memory	Each member within a structure is assigned unique storage area of location.	Memory allocated is shared by individual members of union.
Value Altering	Altering the value of a member will not affect other members of the structure.	Altering the value of any of the member will alter other member values.
Accessing members	Individual member can be accessed at a time.	Only one member can be accessed at a time.
Initialization of Members	Several members of a structure can initialize at once.	Only the first member of a union can be initialized.

```
void main()
{
   typedef struct
   {
        int marks;
        char grade;
        float percentage;
   }student;

student s;

s.marks=90;
s.grade='A';
s.percentage=90.0;

printf("marks:%d\n",s.marks);
printf("Grade:%c\n",s.grade);
printf("percentage:%f\n",s.percent
age);
}
```





```
marks: 90
grade: A
percentage: 90.0
```

```
void main()
{
   typedef union
   {
        int marks;
        char grade;
        float percentage;
   }student;

student s;

S.marks=90;
printf("marks:%d\n",s.marks);

s.grade='A';
printf("Grade:%c\n",s.grade);

s.percentage=90.0;
printf("percentage:%f\n",s.percentage);
}
```

Polynomials

- A polynomial is a sum of terms, where each term has a form ax^e, where x is the variable, a is the coefficient and e is the exponent

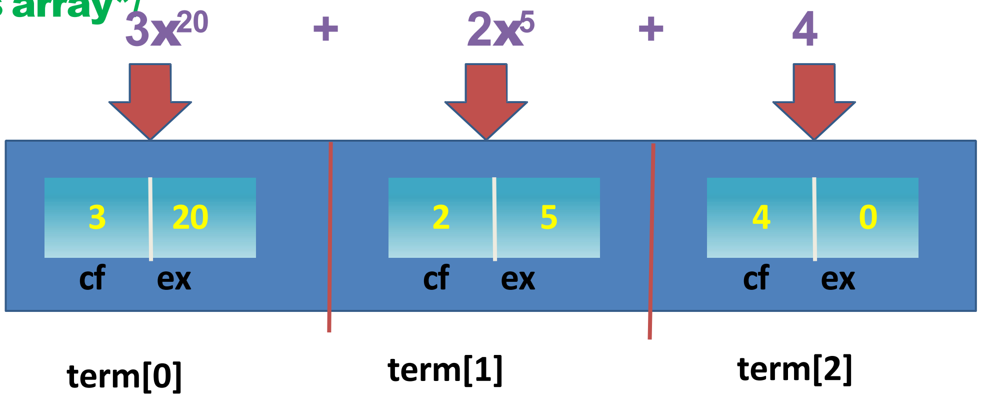
$A(x) = 3x^{20} + 2x^5 + 4$
$B(x) = x^4 + 10x^3 + 3x^2 + 1$

The largest (or leading) exponent of a polynomial is called its degree. Coefficients that are zero are not displayed. The term with exponent equal to zero does not show the variable since x raised to a power of zero is 1.

Polynomial Representation

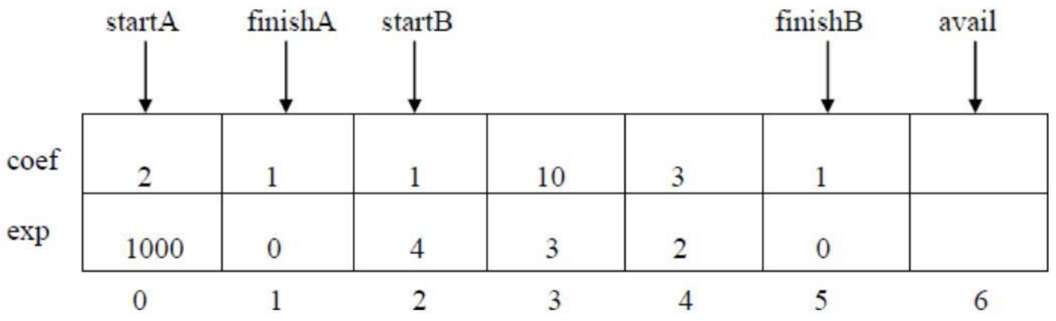
One way to represent polynomials in C is to use typedef to create the type polynomial as below:

```
#define MAX_TERMS 100 /*size of terms array*/
typedef struct
{
        float coef;
        int expon;
} polynomial;

polynomial  terms[MAX-TERMS];
int avail = 0;
```



Consider the two polynomials

$A(x) = 2x^{1000} + 1$
$B(x) = x^4 + 10x^3 + 3x^2 + 1$



	startA	finishA	startB			finishB	avail
coef	2	1	1	10	3	1	
exp	1000	0	4	3	2	0	
	0	1	2	3	4	5	6

The above figure shows how these polynomials are stored in the array terms. The index of the first term of A and B is given by startA and startB, while finishA and finishB give the index of the last term of A and B.

- The index of the next free location in the array is given by avail.
- For above example, startA=0, finishA=1, startB=2, finishB=5, & avail=6.

Program to read and display Polynomial

```
#include<stdio.h>
typedef struct
{
    int cf;  //used to hold coefficient
    int px;  //used to hold power of x
}poly;
//function to read a polynomial with n terms
void read_poly(poly p[], int n)
{
    int i,cf,px;
    for(i=0; i<n; i++)
    {
        printf("enter Coefficient and exponent:");
        scanf("%d%d", &p[i].cf, &p[i].px););
    }
}
//function to display a polynomial with n terms
void print_poly(poly p[], int n)
{
    int i;
            for(i=0; i<n; i++)
    {
            if(p[i].cf < 0)
          printf("%d",p[i].cf);

      if(p[i].px != 0)
      printf("x^%d",p[i].px); }
    printf("\n");}
void main()
{
    int n;
    poly p[10];
    printf("enter number of terms:\n");
    scanf("%d", &n);
    read_poly(p, n);
    print_poly(p,n);
}
```

A T M E College of Engineering

SPARSE MATRICES

What is Sparse Matrix?

A matrix which contains many zero entries or very few non-zero entries is called as **Sparse matrix.** In the figure B contains only 8 of 36 elements are nonzero and that is sparse.

$$\begin{array}{c|ccc|}
 & \text{col0} & \text{col1} & \text{col2} \\
\text{row 0} & -27 & 3 & 4 \\
\text{row 1} & 6 & 82 & -2 \\
\text{row 2} & 109 & -64 & 11 \\
\text{row 3} & 12 & 8 & 9 \\
\text{row 4} & 48 & 27 & 47
\end{array}$$

Figure A

$$\begin{array}{c|cccccc|}
 & \text{col0} & \text{col1} & \text{col2} & \text{col3} & \text{col4} & \text{col 5} \\
\text{row0} & 15 & 0 & 0 & 22 & 0 & -15 \\
\text{row1} & 0 & 11 & 3 & 0 & 0 & 0 \\
\text{row2} & 0 & 0 & 0 & -6 & 0 & 0 \\
\text{row3} & 0 & 0 & 0 & 0 & 0 & 0 \\
\text{row4} & 91 & 0 & 0 & 0 & 0 & 0 \\
\text{row5} & 0 & 0 & 28 & 0 & 0 & 0
\end{array}$$

Figure B

A sparse matrix can be represented in 1-Dimension, 2- Dimension and 3- Dimensional array. When a sparse matrix is represented as a two-dimensional array as shown in Figure B, more space is wasted.

Sparse Matrix Representation

- An element within a matrix can characterize by using the triple <row, col, val> This means that, an array of triples is used to represent a sparse matrix.
- Organize the triples so that the row indices are in ascending order.
- The operations should terminate, so we must know the number of rows and columns, and the number of nonzero elements in the matrix.

```
#define MAX_TERMS 100 /* maximum number of terms */
typedef struct
{
        int col;
        int row;
        int value;
} TERM;

//1- dimensional array representing array of triples<row,col,val>
TERM a[MAX_TERMS];
```

The below figure shows the representation of matrix in the array "a" a[0].row contains the number of rows, a[0].col contains the number of columns and a[0].value contains the total number of nonzero entries.

	col0	col1	col2	col3	col4	col 5
row0	15	0	0	22	0	-15
row1	0	11	3	0	0	0
row2	0	0	0	-6	0	0
row3	0	0	0	0	0	0
row4	91	0	0	0	0	0
row5	0	0	28	0	0	0

Figure B

	Row	Col	Val	
a[0]	6	6	8	6X6 is the size and 8 non zero values in given matrix
[1]	0	0	15	Row 0
[2]	0	3	22	Row 0
[3]	0	5	-15	Row 0
[4]	1	1	11	Row 1
[5]	1	2	3	Row 1
[6]	2	3	-6	Row 2
[7]	4	0	91	Row 4
[8]	5	2	28	Row 5

Fig (a): Sparse matrix stored as triple

- The various information can be accessed using as shown below:
- The size of the matrix using : a[0].row, a[0].col
- The number of non-zero elements using : a[0].val
- The row index of a non-zero element : a[j].row
- The column index of a non-zero element : a[j].col
- The index of non-zero element : a[j].val

$\left.\begin{array}{l} \\ \\ \\ \end{array}\right]$ for j = 1 to a[0].col

//Function to read the sparse matrix as a triple

```
void read_sparse_matrix(TERM a[], int m, int n)
{
      int i,j,k,item;
      a[0].row=m, a[0].col=n, k=1;
      for(i=0; i<m; i++)
      {
            for(j=0; j<n; j++)
            {
                  scanf("%d",&item);
                  if(item==0)
                        continue;
                  a[k].row=i, a[k].col=j, a[k].val=item;
                  k++;
            }
      }
      a[0].val=k-1;
}
```

	col0	col1	col2	col3	col4	col 5
row0	15	0	0	22	0	-15
row1	0	11	3	0	0	0
row2	0	0	0	-6	0	0
row3	0	0	0	0	0	0
row4	91	0	0	0	0	0
row5	0	0	28	0	0	0

Figure B

	Row	Col	Val
a[0]	6	6	8
[1]	0	0	15
[2]	0	3	22
[3]	0	5	-15
[4]	1	1	11
[5]	1	2	3
[6]	2	3	-6
[7]	4	0	91
[8]	5	2	28

6X6 is the size and 8 non zero values in given matrix

Row 0

Row 1

Row 2

Row 4

Row 5

Fig (a): Sparse matrix stored as triple

Transpose of a matrix

$$\begin{array}{c|cccccc|}
 & col0 & col1 & col2 & col3 & col4 & col\ 5 \\
row0 & 15 & 0 & 0 & 22 & 0 & -15 \\
row1 & 0 & 11 & 3 & 0 & 0 & 0 \\
row2 & 0 & 0 & 0 & -6 & 0 & 0 \\
row3 & 0 & 0 & 0 & 0 & 0 & 0 \\
row4 & 91 & 0 & 0 & 0 & 0 & 0 \\
row5 & 0 & 0 & 28 & 0 & 0 & 0
\end{array}$$

Figure B

	Row	Col	Val
a[0]	6	6	8
[1]	0	0	15
[2]	0	3	22
[3]	0	5	-15
[4]	1	1	11
[5]	1	2	3
[6]	2	3	-6
[7]	4	0	91
[8]	5	2	28

Fig (a): Sparse matrix stored as triple

	Row	Col	Val
b[0]	6	6	8
[1]	0	0	15
[2]	0	4	91
[3]	1	1	11
[4]	2	1	3
[5]	2	5	28
[6]	3	0	22
[7]	3	2	-6
[8]	5	0	-15

Fig (b): Transpose matrix stored as triple

6X6 is the size and 8 non zero values in given matrix

Col 0

Col 1

Col 2

Col 3

Col 5

//Function to find the transpose of a given sparse matrix matrix

```
void transpose(TERM a[], TERM b[])
{
      int i,j,k;
      b[0].row=a[0].col;
      b[0].col=a[0].row;
      b[0].val=a[0].val;
      k=1;
      for(i=0; i<a[0].col; i++)
      {
            for(j=1; j<a[0].val; j++)
            {
                  if(a[j].col==i)
                  {
                        b[k].row=a[j].col;
                        b[k].col=a[j].row;
                        b[k].val=a[j].val;
                        k++;
                  }
            }
```

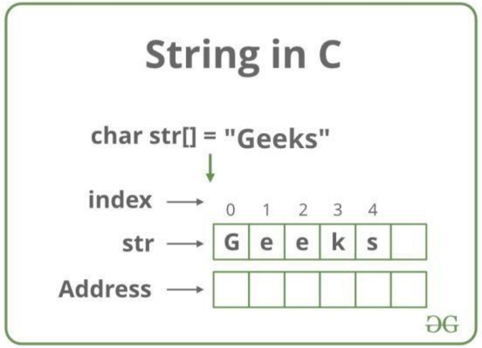
	Row	Col	Val
a[0]	6	6	8
[1]	0	0	15
[2]	0	3	22
[3]	0	5	-15
[4]	1	1	11
[5]	1	2	3
[6]	2	3	-6
[7]	4	0	91
[8]	5	2	28

Fig (a): Sparse matrix stored as triple

	Row	Col	Val
b[0]	6	6	8
[1]	0	0	15
[2]	0	4	91
[3]	1	1	11
[4]	2	1	3
[5]	2	5	28
[6]	3	0	22
[7]	3	2	-6
[8]	5	0	-15

Fig (b): Transpose matrix stored as triple

Character array - Strings



Array which has character in it is called as **String**

Strings end with special character called null character(\0).

Declaration:

char a[8];

	0	1	2	3	4	5	6	7
a								

Initialization:

char a[8] = {'H','e','l','l','o','\0'};

	0	1	2	3	4	5	6	7
a	H	e	l	l	o	\0		

Or

char a[]="HelloHi";

	0	1	2	3	4	5	6	7
a	H	e	l	l	o	H	i	\0

BASIC TERMINOLOGY:

String: A finite sequence S of zero or more Characters is called string.

Length: The number of characters in a string is called length of string.

Empty or Null String: The string with zero characters.

Concatenation: Let S1 and S2 be the strings. The string consisting of the characters of S1 followed by the character S2 is called Concatenation of S1 and S2.

Ex: 'THE' // 'END' = 'THEEND'

'THE' // ' ' // 'END' = 'THE END'

Substring: A string Y is called substring of a string S if there exist string X and Z such that

S = X // Y // Z

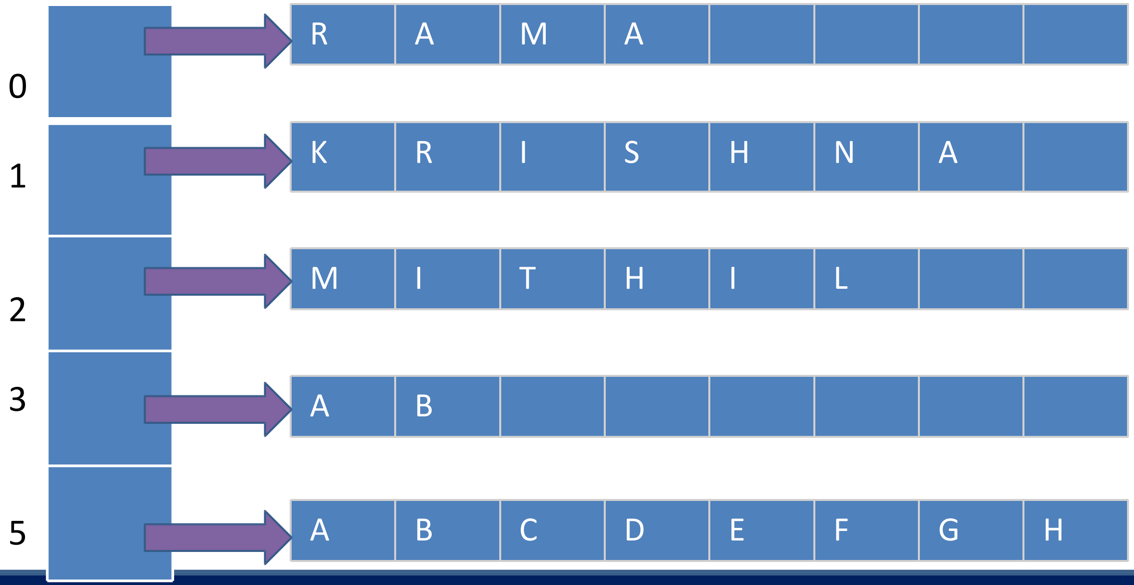
If X is an empty string, then Y is called an Initial substring of S, and Z is an empty string then Y is called a terminal substring of S.

Ex: 'BE OR NOT' is a substring of 'TO BE OR NOT TO BE'

'THE' is an initial substring of 'THE END'

1. Fixed length storage structures

In this storage structure, each line of text to be manipulated is viewed as a record where all records have same length.

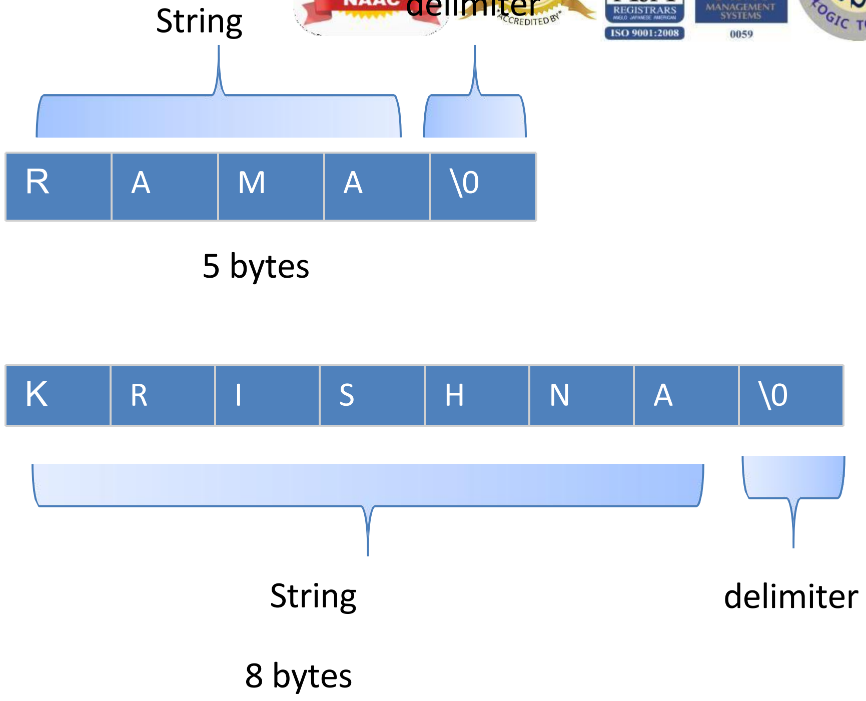


Disadvantages:

1. Time wasted in reading entire record if more spaces are present
2. Certain records may require more space than available to store a string
3. If the length reserved for string is too small, it is not possible to store larger data
4. If the length reserved for string is too large, too much memory is wasted
5. Once the string is defined, the length of the string can't be changed

2. Variable length structures

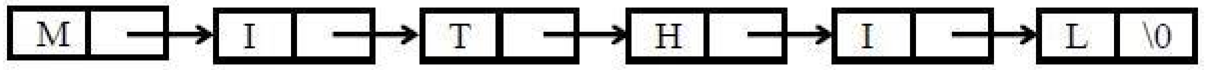
- variable-length strings don't have a pre-defined length. In this string, neither the precise length nor
- maximum length is known at creation time.
- The storage structure for a string can expand or shrink to accommodate any size of data.
- ->In C language strings end with a special character called NULL (denoted by \0)



```
char a[ ]="RAMA";

char a[ ]="KRISHNA";
```

3. Linked storage structures

- In most of the word processing applications, the strings are represented using linked lists.
- Using linked lists inserting/deleting a character/word is much easier.
- The string “MITHIL” can be represented using linked list as shown below:



String Operations

- Substring
- Indexing
- Concatenation
- Length

Substring

Substring is a string obtained by extracting a part of a given string, given the position and length of the substring.

Ex: SUBSTRING ('TO BE OR NOT TO BE', 4, 7) = 'BE OR N'

SUBSTRING ('THE END', 4, 4) = ' END'

Indexing

The process of finding the position of pattern string in a given text t is called indexing.

It is also called pattern matching.

If the paatern string is present in text t, the position of the first occurrence of the pattern string is returned otherwise, 0 is returned.

Let, text t = "RAMA IS THE KING OF AYODHYA", the pattern string pattern is "KING".

Then, INDEX(t, pattern) = 13

Concatenation

- The process of appending the second string to the end of first string is called concatenation.
- We can denote the concatenation symbol by “+”.

For example, let first string is s1=”SEETA” and the second string s2 =” RAMA”.

Then, s1+ s2 = “SEETA RAMA”

Length

The number of characters in a string is called length of the string. For example,

Length(“RAMA”) = 4

Printing and Reading Strings

Formatted - printf (for output)
- scanf (for input)

Un formatted - puts (for output)
- gets (for input)

Ex. for reading string using formatted input function:

```
char c[20];
scanf("%s", c);
```

```
#include<stdio.h>
Void main()
{
   char name[20];
   Printf("enter your name:");
   Scanf("%s", name);
   Printf("your name is %s:", name);
}
```

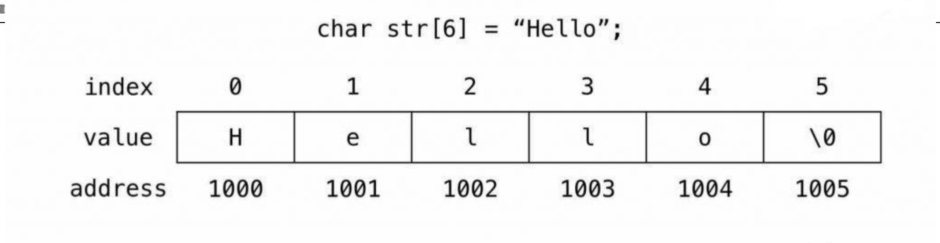
O/p:

```
enter your name: Sachin Tendulkar
your name is: Sachin
```

STRING HANDLING FUNCTIONS IN C

String functions	Description of each function
strlen(str)	Returns length of the string str
strcpy(dest, src)	Copies the source string src to destination string dest
strcat(str1,str2)	Append string str2 to string str1
strcmp(str1,str2)	Compare two strings str1 and str2
strrev(str)	Reverse the contents of string stored in str

String Length



char str[6] = “Hello”;

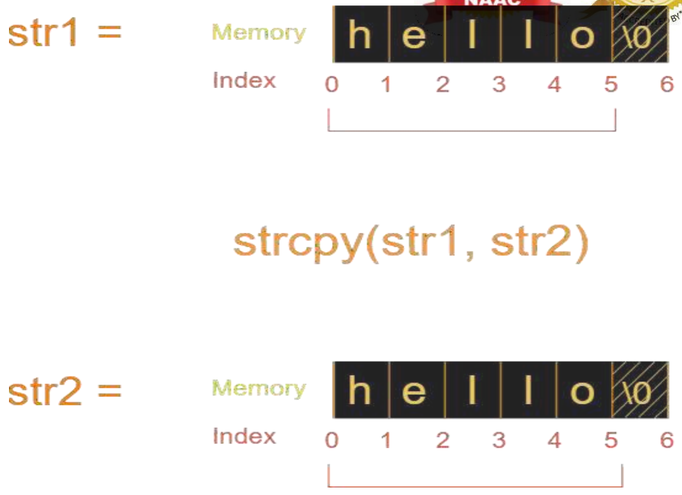
index	0	1	2	3	4	5
value	H	e	l	l	o	\0
address	1000	1001	1002	1003	1004	1005

```
#include<stdio.h>
void main()
{
   char str[6]="Hello";
   int len = my_strlen(str);
   printf("length of str is %d",len);
}
int my_strlen(char str[])
{
        int i=0;
        while(str[i] != '\0')
        {
              i++;
        }
        return i;
}
```

```
#include<stdio.h>
#include<string.h>
void main()
{
   char str[6]="Hello";
   int len = strlen(str);
   printf("length of str is %d",len);
}
```

String copy

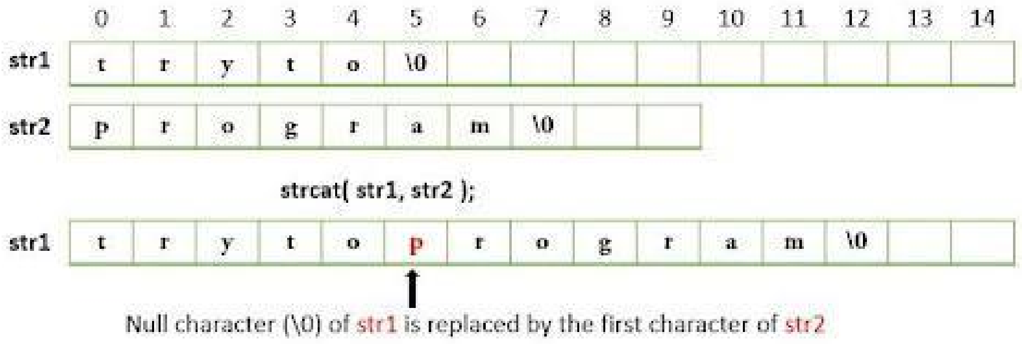
```
#include<stdio.h>
void main()
{
   char str1[6]="HELLO";
   char str2[6];
   My_strcpy(str1, str2);

}
void my_strcpy(char str1[], char str2[])
{
        int i=0;
        while(str1[i] != '\0')
        {
              str2[i]=str1[i];
              i++;
        }
        str2[i]='\0';
}
```



```
# i n c l u d e < s t d i o . h >
#include<string.h>
void main()
{
   char str1[6]="HELLO";
   char str2[6];
   strcpy(str1,str2);
   printf("string 2 is %s",str2);
}
```

String concatenation

```
#include<stdio.h>
void main()
{
   char str1[6]="HELLO";
   char str2[6]="WORLD";
   My_strcat(str1, str2);
}
void my_strcat(char str1[], char str2[])
{
        int i,j;
        i=0,j=0;
        while(str1[i] != '\0')
        {
              i++;
        }
        while(str2[j]!='\0')
        {
              str1[i++]=str2[j++];
        }
        str1[i++] = '\0';
}
```

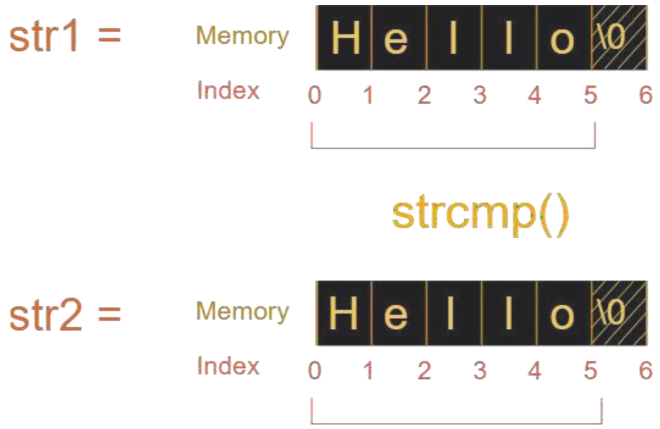


0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
str1
t r y t o \0
str2
p r o g r a m \0
strcat(str1, str2);
str1
t r y t o p r o g r a m \0
Null character (\0) of str1 is replaced by the first character of str2

```
#include<stdio.h>
#include<string.h>
void main()
{
   char str1[6]="HELLO";
   char str2[6]="WORLD";
   strcat(str1,str2);
   printf("str1 is %s",str1);
}
```

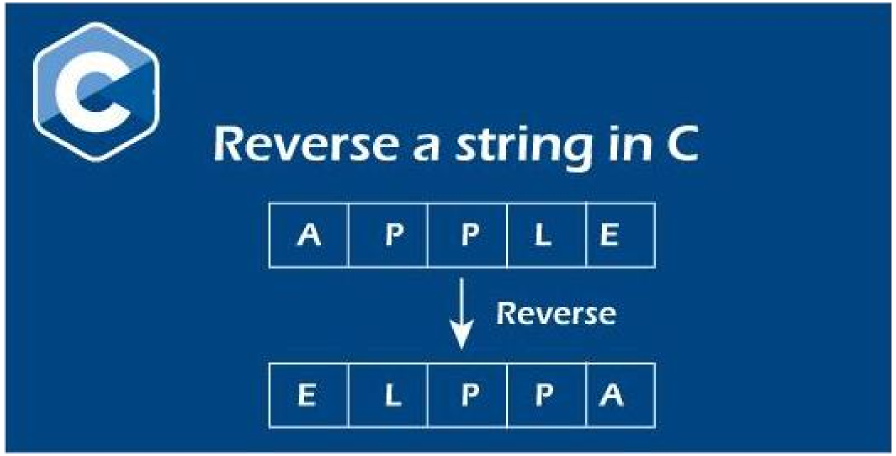
String compare

```
#include<stdio.h>
void main()
{
   char str1[6]="Hello";
   char str2[6]="Hello";
   int res = my_strcmp(str1, str2);
   if(res==0) { printf("strings are equal\n"); }
   else if(res < 0) { printf("string 1 is smaller than string 2");
   else { printf("string 1 is greater than string 2"); }
}
Int my_strcmp(char str1[], char str2[])
{
        int i;
        i=0;
        while(str1[i] == str2[i])
        {
                if(str1[i] == '\0')
                      break;
                i++;
        }
        return str1[i] - str2[i];
}
```



String Reverse

```
#include<stdio.h>
#include<string.h>
void main()
{
   char str1[6]="Hello";
   char str2[6];
   my_strrev(str1, str2);

}
void my_strrev(char str1[], char str2[])
{
        int i,n;
        n = strlen(str1)
        for(i=0; i<n; i++)
        {
                  str2[n-1-i] = str1[i];
        }
        str2[n] = '\0';
}
```



PATTERN MATCHING ALGORITHM

->Pattern matching is the problem of deciding whether or not a given string pattern P appears in a string text T. The length of P does not exceed the length of T.

Brute Force Algorithm

The first pattern matching algorithm is one in which comparison is done by a given pattern P with each of the substrings of T, moving from left to right, until a match is found.

WK = SUBSTRING (T, K, LENGTH (P))

- Where, WK denote the substring of T having the same length as P and beginning with the Kth character of T.

- First compare P, character by character, with the first substring, W1. If all the characters are the same, then P = W1 and so P appears in T and INDEX (T, P) = 1.
- Suppose it is found that some character of P is not the same as the corresponding character of W1. Then P ≠ W1
- Immediately move on to the next substring, W2 That is, compare P with W2. If P ≠ W2 then compare P with W3 and so on.
- The process stops, When P is matched with some substring WK and so P appears in T and INDEX(T,P) = K or When all the WK'S with no match and hence P does not appear in T.
- The maximum value MAX of the subscript K is equal to LENGTH(T) -LENGTH(P) +1.

Algorithm: (Pattern Matching_Brute Force)

P and T are strings with lengths R and S, and are stored as arrays with one character per element. This algorithm finds the INDEX of P in T.

```
1. [Initialize.] Set K: = 1 and MAX: = S - R + 1
2. Repeat Steps 3 to 5 while K ≤ MAX
3. Repeat for L = 1 to R: [Tests each character of P]
        If P[L] ≠ T[K + L – l], then: Go to Step 5
        [End of inner loop.]
4.  [Success.] Set INDEX = K, and Exit
5. Set K := K + 1
   [End of Step 2 outer loop]
6. [Failure.] Set INDEX = O
7. Exit
```

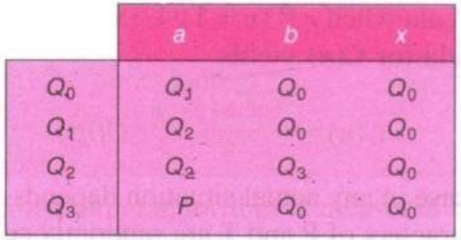
ATME College of Engineering

PATTERN MATCHING ALGORITHM - KMP algorithm(Knuth Morris Pratt)

```
Algm PatternMatching_KMP(s, p)        return int
{
    k=1, s1=$q_0$, n=length(s)
    While(k<=n && $s_k$!=p)
    {
        Read $t_K$
        $s_{k+1}$ = F($s_k$,$t_k$)
        k=k+1
    }
    If(k>n)
    {
        index=0
    }
    else
    {
        Index=k-length(p)
    }
    return index
}
```

	1	2	3	4	5	6	7	8	9	10	11
String s:	a	a	a	a	a	a	a	a	a	a	b

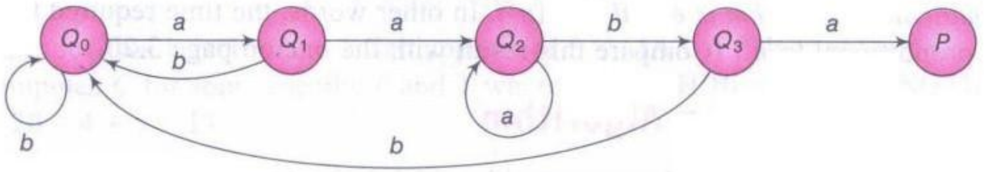
Pattern p: a a a b



	a	b	x
Q_0	Q_1	Q_0	Q_0
Q_1	Q_2	Q_0	Q_0
Q_2	Q_2	Q_3	Q_0
Q_3	P	Q_0	Q_0

(a) Pattern matching table

Pattern p: a a b a



$$\begin{tikzcd} Q_0 \arrow[r, "a"] \arrow[loop left, "b"] & Q_1 \arrow[r, "a"] \arrow[l, bend left, "b"] & Q_2 \arrow[r, "b"] \arrow[loop below, "a"] & Q_3 \arrow[r, "a"] \arrow[lll, bend left, "b"] & P \end{tikzcd}$$

b Pattern matching graph

Stacks

Stacks: Definition, Stack Operations, Array Representation of Stacks, Stacks using Dynamic Arrays, Stack Applications: Polish notation, Infix to postfix conversion, evaluation of postfix Expression

Recursion - Factorial, GCD, Fibonacci Sequence, Tower of Hanoi, Ackerman's Function.

Queues: Definition, Array Representation, Queue Operations, Circular Queues,

Circular queues using Dynamic arrays, Dequeues, Priority Queues, A Mazing Problem. Multiple Stacks and Queues. Programming Examples.

STACK

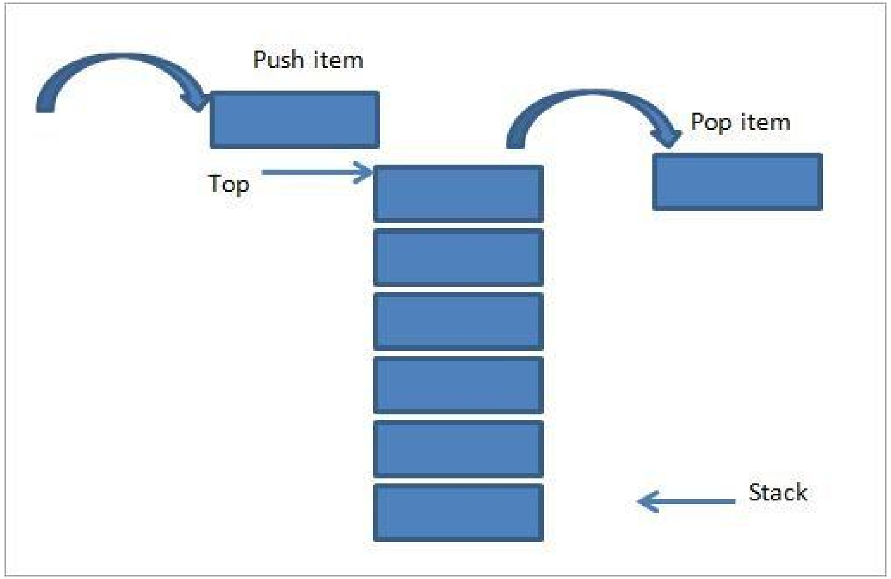
-> Stack is a linear data structure which follows a particular order in which the operations are performed.

->The order may be LIFO(Last In First Out) or FILO(First In Last Out).

-> A stack can be implemented by means of Array, Structure, Pointer, and Linked List.

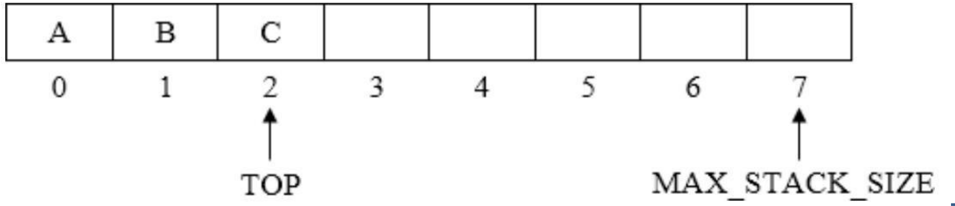
-> Here, we are going to implement stack using arrays, which makes it a fixed size stack implementation.

->At any given time, we can only access the top element of a stack.



ARRAY REPRESENTATION OF STACKS

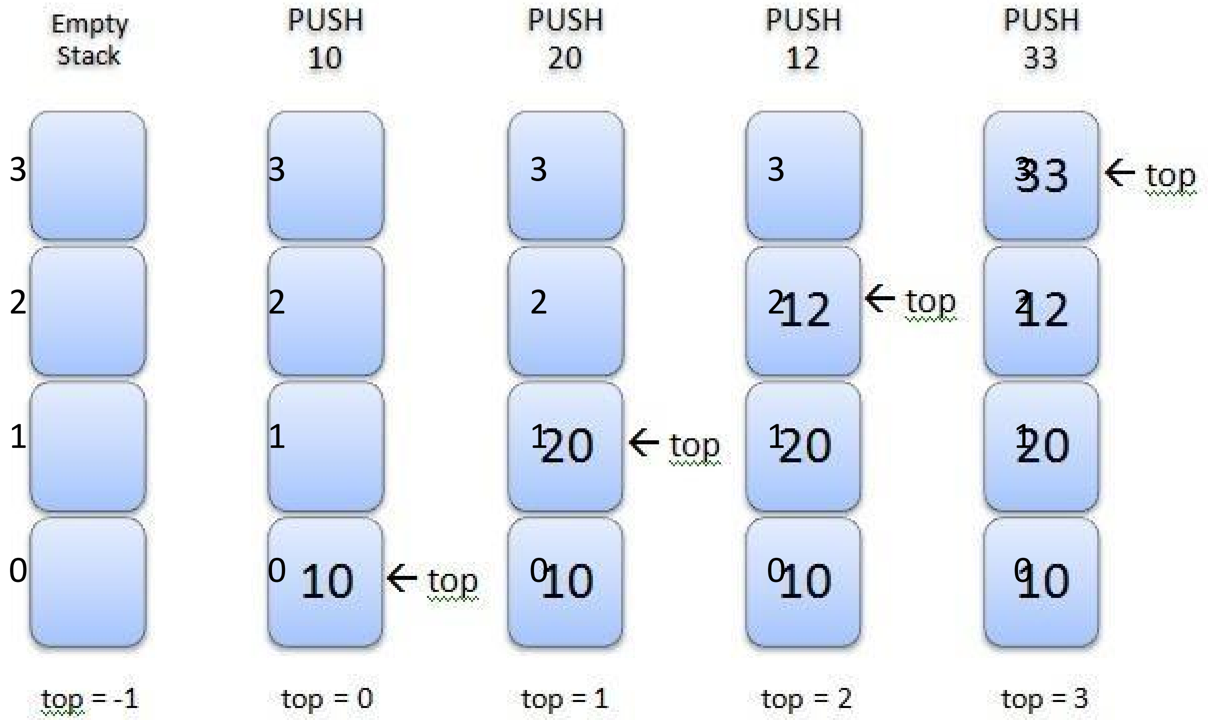
- Stacks may be represented in the computer in various ways such as one-way linked list
  (Singly linked list) or linear array.
  • Stacks are maintained by the two variables such as TOP and MAX_STACK_SIZE.
  • TOP which contains the location of the top element in the stack. If TOP= -1, then it indicates stack is empty.
  • MAX_STACK_SIZE which gives maximum number of elements that can be stored in stack.
  Stack can represented using linear array as shown below



STACK OPERATIONS

push() – Pushing (storing) an element on the stack.

->Check for overflow condition i.e top = Max_Size-1
->Increment top by 1
->Push an element



```
#define Max_Size 4
void push()
{
    if (top >= Max_Size-1)
      Stack overflow
    Else
       top++
      stack[top] = item;
}
```

Function to push an integer item (using global variables)

```
void push()
{
      If(top == Max_Size - 1)
      {
            printf("Stack Overflow");
            exit(0);
      }
      top++;
      s[top] = item;
}
```

Function to push an integer item (by passing parameter)

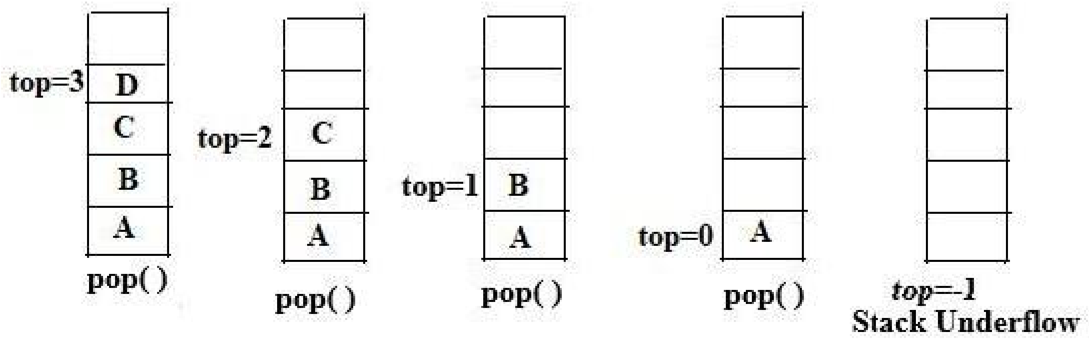
```
void push(int item, int top, int s[])
{
      If(top == Max_Size - 1)
      {
            Printf("Stack Overflow");
            Return;
      }
      top++;
      s[top] = item;
}
```

pop()

Deleting an element from the stack is called pop **operation. The element is deleted only from the top of the stack and only one element is deleted at a time.**

Removing an element from the stack.

- **Check underflow condition i.e top=-1**
- **Item=s[top]**
- **Decrement top by 1**



Function to pop an integer item (using global variables)

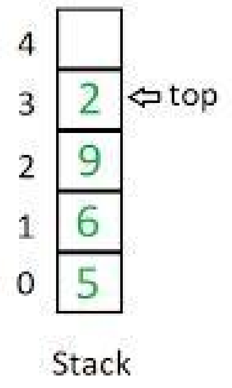
```
int pop()
{
    int item_deleted;

    If(top==-1)
       Return 0;

    item_deleted=s[top--];
        Return item_deleted;
}
```

Function to pop an integer item (by passing parameter)

```
int pop(int top, int s[])
{
    int item_deleted;

    If(top==-1)
       Return 0;

    item_deleted=s[(top)--];
        Return item_deleted;
}
```

Display()

->Display the elements of stack
->Check for underflow condition
->Display using for loop



Display Stack content using global variables

```
void display()
{
    Int i;
    If(top==-1)
    {
        Printf("stack is empty\n");
        Return;
    }
    Printf("Contents of the stack are:\n");
    For(i=0; i<=top; i++)
    {
         Printf("%d\n",s[i]);
    }
}
```

Display Stack content by pass by parameter

```
void display(int top, int s[])
{
    Int i;
    If(top==-1)
    {
        Printf("stack is empty\n");
        Return;
    }
    Printf("Contents of the stack are:\n");
    For(i=0; i<=top; i++)
    {
         Printf("%d\n",s[i]);
    }
}
```

Applications/Advantages of Stack

- Stacks can be used for expression evaluation.
- Stacks can be used to check parenthesis matching in an expression
- Stacks can be used for Conversion from one form of expression to another.
- Stacks can be used for Memory Management

Expression can be represented in in different format such as

- Prefix Expression or Polish notation
- Infix Expression
- Postfix Expression or Reverse Polish notation

Infix Expression: In this expression, the binary operator is placed in-between the operand. The expression can be parenthesized or un- parenthesized.
Example: A + B
Here, A & B are operands and + is operator

Prefix or Polish Expression: In this expression, the operator appears before its operand.
Example: + A B
Here, A & B are operands and + is operator

Postfix or Reverse Polish Expression: In this expression, the operator appears after its operand.
Example: A B +   Here, A & B are operands and + is operator

Steps to evaluate postfix expression

Step 1: If operand - push
Step2: If operator -

- **Pop the top of the stack and make it operand 2**
- **Pop the next top of the stack and make operand 1**
- **Perform operation**
- **Push the result**

Infix -> 4 + 2 * 3 postfix -> 4 2 3 * +

Infix -> (8 + 5) * (6 / 3) postfix -> 8 5 + 6 3 / *

Infix -> 4 $ 2 *3 -3 + 8 / 4 / (1 + 1) postfix -> 4 2 $ 3 * 3 - 8 4 / 1 1 + / +

Steps to convert infix to postfix expression

Step 1: If operand - send it to postfix expression

Step2: If operator -

Check the Priority of current operator (PCO) VS Priority of top of stack(POTS)

If(PCO > POTS) then PUSH

If(PCO = POTS) then POP

If(PC0 < POTS) then POP

Step3: Parenthesis

If (PUSH

If) PUSH

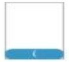
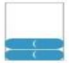
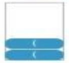
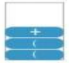
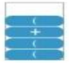
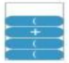
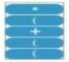
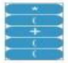
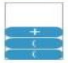
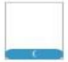
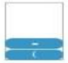
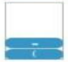
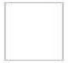
Permitted to push any symbol above the brackets

POP all the symbols between (and)

Priorities:

1. $, ^
2. *, /
3. +, -

Example : ((4 + (8 * 2)) - 10)

S.No	Scanned element	Input Expression	Operator Stack	Output Expression	Description
1	(	(4 + (8 * 2)) - 10)			Push '(' to stack
2	(	4 + (8 * 2)) - 10)			Push '(' to stack
3	4	+ (8 * 2)) - 10)		4	Output value
4	+	(8 * 2)) - 10)		4	Push '+' to stack
5	(	8 * 2)) - 10)		4	Push '(' to stack
6	8	* 2)) - 10)		4 8	Output value
7	*	2)) - 10)		4 8	Push '*' to stack
8	2	)) - 10)		4 8 2	Output value
9	)	) - 10)		4 8 2 *	Pop till '(' is found
10	)	- 10)		4 8 2 * +	Pop till '(' is found
11	-	10)		4 8 2 * +	Push '-' to stack
12	10	)		4 8 2 * + 10	Output value
13	)			4 8 2 * + 10 -	Pop till '(' is found is_empty()

/* Function to convert infix to postfix */

```
void main()
{
char infix[20];
char postfix[20];
Input and output
printf("Enter a valid infix expression\n");
scanf("%s",infix);
/* Convert infix to postfix expression */
infix_postfix(infix, postfix);
printf("The postfix expression is\n");
printf("%s\n",postfix);
}
```

```
  void infix_postfix(char infix[], char postfix[])
{
int top;          /* points to top of the stack */
int j;            /* Index for postfix expression */
int i;             /* Index to access infix expression*/
char s[30];          /* Acts as storage for stack elements */
char symbol       /* Holds scanned char from infix exprn */
top = -1;                   /* Stack is empty */
s[++top] = '#';             /* Initialize stack to # */
j = 0;                     /* Output is empty. So, j = 0 */
```

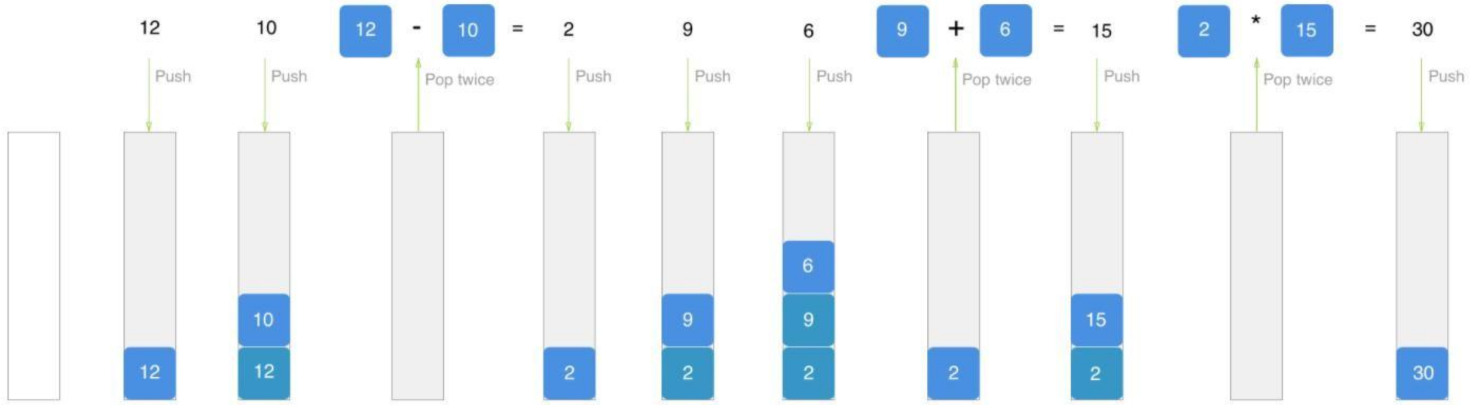
```
for( i = 0; i < strlen(infix); i++)
{
symbol = infix[i];                              /* Scan the next symbol */
while ( F(s[top]) > G(symbol) )                 /* if stack precedence is greater */
postfix[j++] = s[top--];                        /* Pop and place into postfix*/
if ( F(s[top]) != G(symbol) )
s[++top] = symbol;                              /* push the input symbol */
else
top--;                                          /* discard '(' from stack */
}
while ( s[top] != '#')
postfix [j++] = s[top--];                       /*pop and place in postfix */
postfix[j] = '\0';                              /* NULL terminate */
}
```

/* Function to evaluate postfix expression */

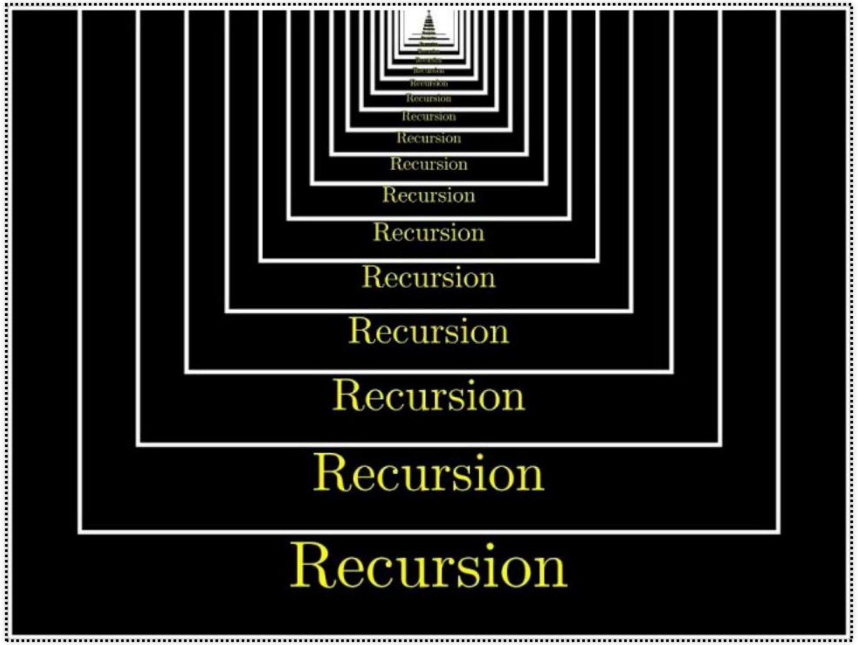
```
int eval(void)
{
        precedence token;
        char symbol;
        int opl,op2, n=0;
        int top= -1;
        token = getToken(&symbol, &n);
        while(token! = eos)
        {
                if (token == operand)
                push(); /* stack insert */
                else
                {
                        op2 = pop(); /* stack delete */
                        opl = pop();
                        switch(token)
                        {
                                case plus: push(opl+op2);
                                break;
                                case minus: push(opl-op2);
                                break;
                                case times: push(opl*op2);
                                break;
                                case divide: push(opl/op2);
                                break;
                                case mod: push(opl%op2);
                                break;
                        }
                }
                token = getToken(&symbol, &n);
        }
        return pop(); /* return result */
}
```

Postfix Evaluation Example

Example : 12 10 - 9 6 + *

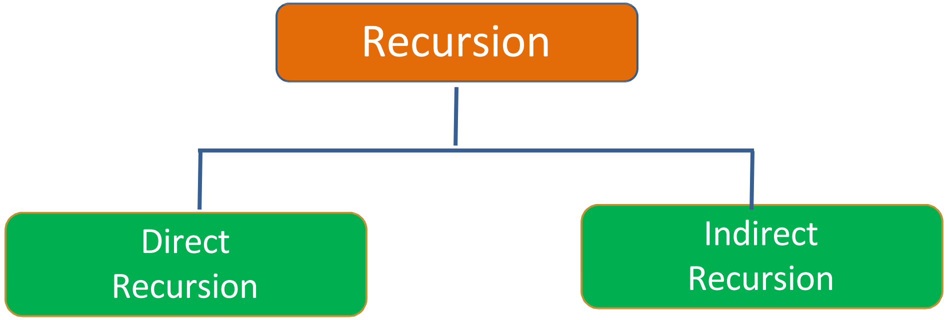


Recursion



A process of a function calling itself is what is called Recursion

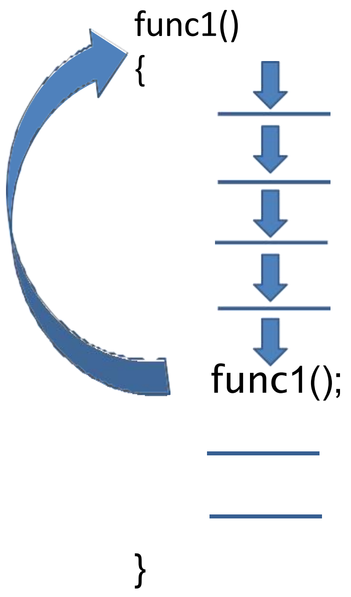
Recursion refers to the process where a function calls itself directly or indirectly

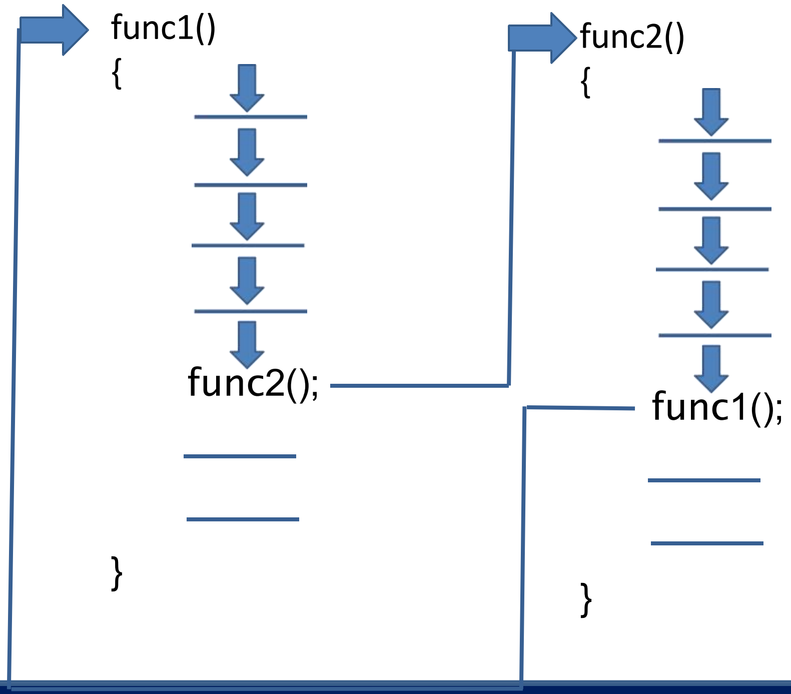


Direct Recursion

Indirect Recursion

Function calls itself directly





Direct Recursion

Example:

```
#include<stdio.h>
void main()
{
    printf("I LOVE MY COUNTRY\n");
    main();
}
```

Output:

```
I LOVE MY COUNTRY
I LOVE MY COUNTRY
…
…
…
(infinte number of times I LOVE MY
COUNTRY will printed )
```

Indirect Recursion

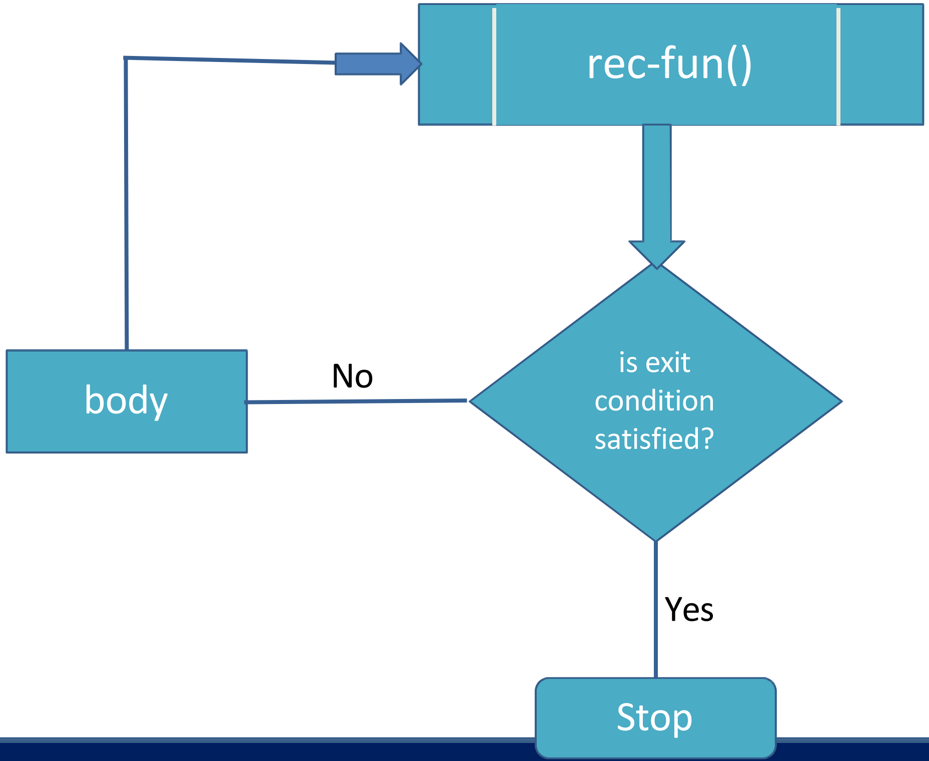
Example:

```
#include<stdio.h>
void fun(void);
void main()
{
    printf("I LOVE MY COUNTRY\n");
    fun();
}
fun()
{
    main();
}
```

Output:

```
I LOVE MY COUNTRY
I LOVE MY COUNTRY
…
(infinte number of times I LOVE MY
COUNTRY will printed )
```

General format of recursive function:



factorial of a given number

Factorial of a number 5!=5*4*3*2*1=120

```
#inlcude<stdio.h>
void main()
{
      fact=1;
     Printf("enter the no\n");
    Scanf("%d",&n);
    For(i=1;i<=n;i++)
     {
          fact=fact*i;
     }
  printf("factorial of a number is %d", fact);
}
```

Example

```
int sum(int k);
int main() {
  int result = sum(10);
  printf("%d", result);
  return 0;
}
int sum(int k)
  { if (k > 0) {
    return k + sum(k - 1);
  } else
    { return
    0;
  }}
```

Write a program to compute the factorial of a given number n using recursion.

0! = 1
1! = 1*(1-1)! = 1*1 =1
2! = 2*(2-1)! = 2*1 = 2
3! = 3*(3-1)! = 3*2 = 6
4! = 4*(4-1)! = 4*6 = 24
.
.
.
n! = n*(n-1)!

$$\text{fact}(n) = \begin{cases} 1 & \text{if } n==0 \\ n*\text{fact}(n-1) & \text{if } n>0 \end{cases}$$

```
#include<stdio.h>
int fact(int);
void main()
{
   int n,ans;
   printf("enter the value of n\n");
   scanf("%d",&n);
   ans=fact(n);
   printf("answer is %d",ans);
}
int fact(int n)
{
   if(n==0)
   {
      return 1;
   }
   else
   {    return  n*fact(n-1);} }
```

Fibonacci Sequence

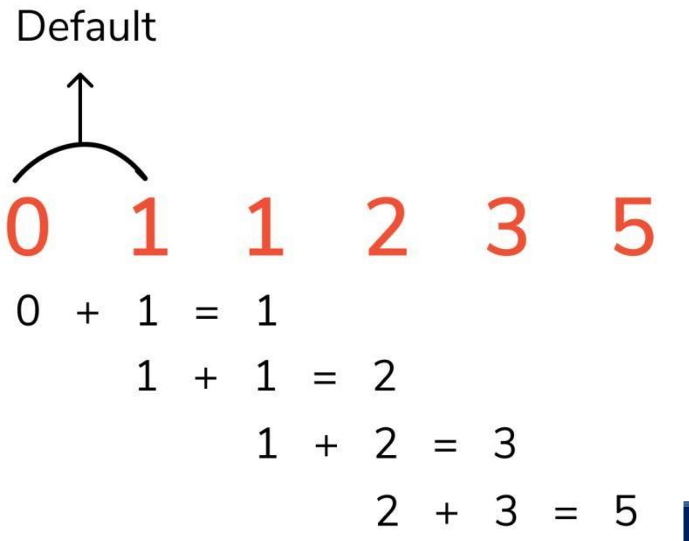
A series of numbers in which each number (*Fibonacci number*) is the sum of the two preceding numbers.

n	1	2	3	4	5	6	7	8	9	10
fib(n)	0	1	1	2	3	5	8	13	21	34

$$F_0 = 0, \quad F_1 = 1,$$

$$F_n = F_{n-1} + F_{n-2}$$

$$0,\ 1,\ 1,\ 2,\ 3,\ 5,\ 8,\ 13,\ 21,\ 34,\ 55,\ 89,\ 144,\ \ldots$$



$$\text{fib}(n) = \begin{cases} 0 & \text{if } n==1 \\ 1 & \text{if } n==2 \\ \text{fib}(n-1)+\text{fib}(n-2) & \text{if } n>2 \end{cases}$$

```
#include<stdio.h>
int fib(int);
void main()
{
    int n,ans;
    printf("enter the value of n\n");
    scanf("%d",&n);
    ans=fib(n);
    printf("answer is %d",ans);
}
```

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

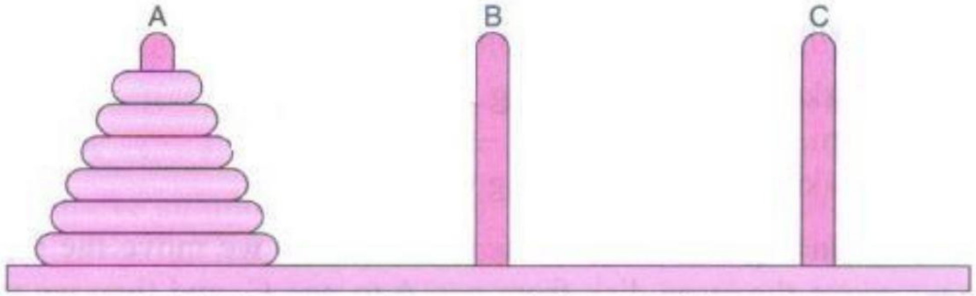
Tower of Hanoi

Suppose three pegs, labeled A, Band C, are given, and suppose on peg A a finite number n of disks with decreasing size are placed.

The objective of the game is to move the disks from peg A to peg C using peg B as an auxiliary.

The rules of the game are as follows:

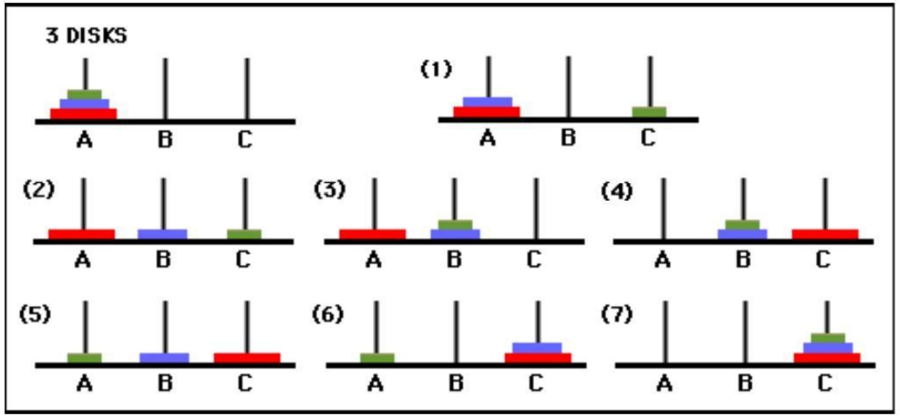
1. Only one disk may be moved at a time. Only the top disk on any peg may be moved to any other peg.
2. At no time can a larger disk be placed on a smaller disk



Initial Setup of Towers of Hanoi with n = 6

Example: Towers of Hanoi problem for n = 3.

Solution: Observe that it consists of the following seven moves



1. Move top disk from peg A to peg C.
2. Move top disk from peg A to peg B.
3. Move top disk from peg C to peg B.
4. Move top disk from peg A to peg C.
5. Move top disk from peg B to peg A.
6. Move top disk from peg B to peg C.
7. Move top disk from peg A to peg C.

In other words,

n=3: A→C, A→B, C→B, A→C, B→A, B→C, A→C

the solution to the Towers of Hanoi problem for n = 1 and n = 2

n=l: A→C

n=2: A→B, A→C, B→C

The Towers of Hanoi problem for n > 1 disks may be reduced to the following sub-problems:

(1) Move the top n - 1 disks from peg A to peg B

(2) Move the top disk from peg A to peg C:A→C.

(3) Move the top n - 1 disks from peg B to peg C

The general notation

• **TOWER (N, BEG, AUX, END) to denote a procedure which moves the top n disks from the initial peg BEG to the final peg END using the peg AUX as an auxiliary.**

• **When n = 1, the solution:**

TOWER (1, BEG, AUX, END) consists of the single instruction BEG→END

• **When n > 1, the solution may be reduced to the solution of the following three sub problems:**

(a) TOWER (N - I, BEG, END, AUX)

(b) TOWER (l, BEG, AUX, END) or BEG → END

(c) TOWER (N - I, AUX, BEG, END)

Procedure: TOWER (N, BEG, AUX, END)

This procedure gives a recursive solution to the Towers of Hanoi problem for N disks.

```
1. If N=l,then:
       (a) Write: BEG →END.
       (b) Return.
   [End of If structure.]
2. [Move N - 1 disks from peg BEG to peg AUX.]
   Call TOWER (N - 1, BEG, END, AUX).
3. Write: BEG →END.
4. [Move N - 1 disks from peg AUX to peg END.]
   Call TOWER (N - 1, AUX, BEG, END).
5. Return.
```

```
#include<stdio.h>
void tower(int n,char frompeg,char
topeg,char auxpeg);
int n;
void main()
{
        printf("Enter the no. of discs: \n");
        scanf("%d",&n);
        printf("the number of moves in tower
of henoi problem\n");
        tower(n,'A','C','B');
}
```

```
void tower(int n,char frompeg,char topeg,char
auxpeg)
{
        if(n==1)
        {
 printf("move disk1  from %C to %C\n",
                                        frompeg,topeg);
                return;
        }
tower(n-1,frompeg,auxpeg,topeg);
 printf("move disk%d from %C to%C\n",n,
                                        frompeg,topeg);
   tower(n-1,auxpeg,topeg,frompeg);
}
```