

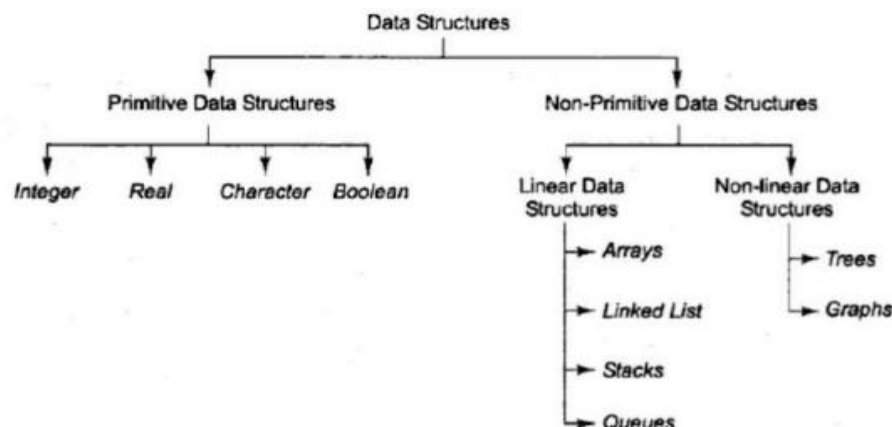
Module 1- Introduction

<p>Module 1 Syllabus</p>	<p>INTRODUCTION TO DATA STRUCTURES: Data Structures, Classifications (Primitive & Non-Primitive), Data structure Operations Review of pointers and dynamic Memory Allocation, ARRAYS and STRUCTURES: Arrays, Dynamic Allocated Arrays, Structures and Unions, Polynomials, Sparse Matrices, representation of Multidimensional Arrays, Strings STACKS: Stacks, Stacks Using Dynamic Arrays, Evaluation and conversion of Expressions</p>
---------------------------------	---

1.1 Data Structure

- **Data** is basically a fact or an entity and is used for calculation or manipulation.
- **Data structure** is a representation of logical relationship existing between individual elements of data.
- Data Structure is a way of collecting and organizing data in such a way that we can perform operations on these data in an effective way.
- A data structure is a storage that is used to store and organize data. It is a way of arranging data on a computer so that it can be accessed and updated efficiently.
- Ex: GPS, Escalators, QueueSystem, Feeds, Online ticket Booking .

1.2 Classification of data structures



Data structures are generally categorized into two classes:

- **Primitive data Structures**
- **Non-primitive data Structures**

Primitive Data Structures: Primitive data structures are the basic data structures that directly operate upon the machine instructions. They are the fundamental data types which are supported by a programming language. Some basic data types are integer, real, character, and Boolean. The terms ‘data type’, ‘basic data type’, and ‘primitive data type’ are often used interchangeably.

Non-Primitive Data Structures: Non-primitive data structures are those data structures which are created using primitive data structures. Examples of such data structures include linked lists, stacks, trees, and graphs.

Based on the structure and arrangement of data, non-primitive data structures is further classified into

- Linear Data Structure
- Non-linear Data Structure

Linear data Structures :If the elements of a data structure are stored in a linear or sequential order, then it is a linear data structure. Linear data structures can be represented in memory in two different ways.

- One way is to have to a linear relationship between elements by means of sequential memory locations.
- The other way is to have a linear relationship between elements by means of links.
- Examples include arrays, linked lists, stacks, and queues.

1. **Array:** Simplest type of data structure is linear array. It is the list of finite numbers n of similar data elements referenced respectively by a set of n consecutive numbers. Ex: array of students consisting of six students shown in fig.

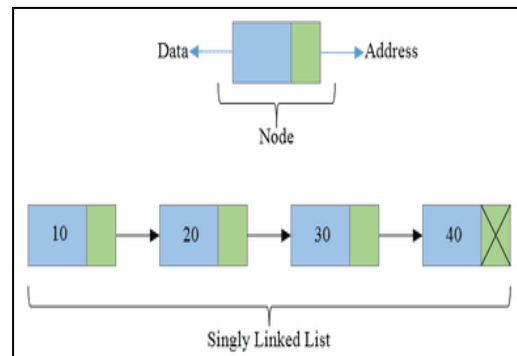


John Brown	Sandra Gold	Tom Jones	June Kelly	Mary Reed	Alan Smith
------------	-------------	-----------	------------	-----------	------------

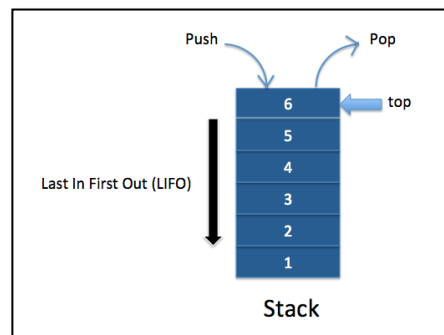
2. **A linked list** is a very flexible, dynamic data structure in which elements (called nodes) form a sequential list. A linked list, every node contains the following two types of data:

- The value of the node or any other data that corresponds to that node
- A pointer or link to the next node in the list

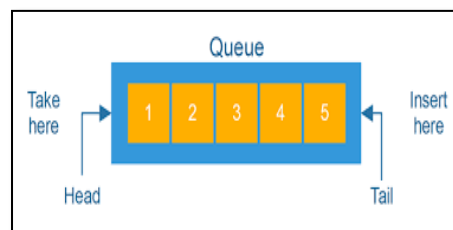
The last node in the list contains a NULL pointer to indicate that it is the end or tail of the list. Since the memory for a node is dynamically allocated when it is added to the list, the total number of nodes that may be added to a list is limited only by the amount of memory available. Figure shows a linked list of four nodes shown in fig.



3. **A stack** is a linear data structure in which insertion and deletion of elements are done at only one end, which is known as the top of the stack. Stack is called a last-in, first-out (LIFO) structure because the last element which is added to the stack is the first element which is deleted from the stack.

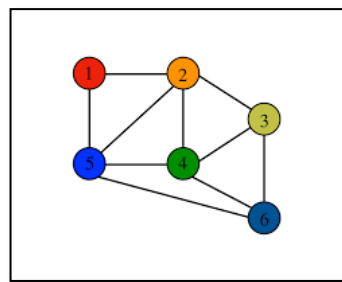


4. **A queue** is a first-in, first-out (FIFO) data structure in which the element that is inserted first is the first one to be taken out. The elements in a queue are added at one end called the rear and removed from the other end called the front.

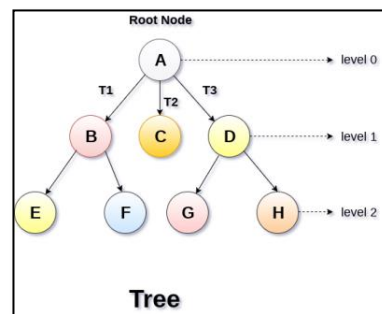


Non-linear data Structures If the elements of a data structure are not stored in a sequential order, then it is a non-linear data structure.

- The relationship of adjacency is not maintained between elements of a non-linear data structure.
 - This structure is mainly used to represent data containing a hierarchical relationship between elements.
 - **Examples include trees and graphs.**
1. **A graph** is a non-linear data structure which is a collection of vertices (also called nodes) and edges that connect these vertices. A graph is often viewed as a generalization of the tree structure, where instead of a purely parent-to-child relationship between tree nodes, any kind of complex relationships between the nodes can exist.



2. **A tree** is a non-linear data structure which consists of a collection of nodes arranged in a hierarchical order. One of the nodes is designated as the root node, and the remaining nodes can be partitioned into disjoint sets such that each set is a sub-tree of the root.



1.3 Data structure Operations

The data appearing in our data structures are processed by means of certain operations. Infact the particular data structure that one chooses for a given situation depends largely on the frequency with which specific operations are performed such as:

1. **Creating** - Creating a new record with some data.
2. **Traversing**- It is used to access each data item exactly once so that it can be processed. This accessing and processing sometimes called as visiting the record.
3. **Searching**- It is used to find out the location of the data item if it exists in the given collection of data items.

4. **Inserting**- It is used to add a new data item in the given collection of data items.
5. **Deleting**- It is used to delete an existing data item from the given collection of data items.
6. **Sorting**- It is used to arrange the data items in some order i.e. in ascending or descending order in case of numerical data and in dictionary order in case of alphanumeric data.
7. **Merging**- It is used to combine the data items of two sorted files into single file in the sorted form.

1.4 Review of Pointers

A pointer is a variable that contains the memory location of another variable. Therefore, a pointer is a variable that represents the location of a data item, such as a variable or an array element. Pointers are frequently used in C, as they have a number of useful applications. These applications include:

- Pointers are used to pass information back and forth between functions.
- Pointers enable the programmers to return multiple data items from a function via function arguments.
- Pointers provide an alternate way to access the individual elements of an array.
- Pointers are used to pass arrays and strings as function arguments. We will discuss this in subsequent chapters.
- Pointers are used to create complex data structures, such as trees, linked lists, linked stacks, linked queues, and graphs.
- Pointers are used for the dynamic memory allocation of a variable.

1.4.1 Declaring Pointer variables

The general syntax of declaring pointer variables can be given as below.

data_type *ptr_name;

Here, data_type is the data type of the value that the pointer will point to.

Eg code which shows the use of a pointer variable:

```
#include <stdio.h>

int main()
{
    int num, *pnum;
    pnum = &num;
    printf("\n Enter the number : ");
```

```
scanf("%d", &num);  
printf("\n The number that was entered is : %d", *pnum);  
return 0;  
}
```

Output

Enter the number : 10

The number that was entered is : 10

Note : The value of **num* is equivalent to simply writing *num*.

1.4.2 Pointer expressions and Pointer arithmetic

Pointer variables can be used in expressions. In C Programmer may:

- Add or subtract integers from pointers.
- Subtract two pointers of the same type.
- Use shorthand operators with pointer variables.
- Compare pointers using relational operators.
- Postfix unary increment (++) and decrement (--) operators have greater precedence than the dereference operator (*).
- Write (**ptr*)++ , to increment the value of the variable whose address is stored in *ptr*.

1.4.3 Null Pointers

- A null pointer is a special pointer value that does not point to any valid memory address.
- To declare a null pointer, use the predefined constant NULL, `int *ptr = NULL;`
- Check whether a pointer variable is null by writing `if(ptr == NULL).`
- Initialize a pointer to null using the constant 0, but it is better to use NULL to avoid ambiguity. `int *ptr; ptr = 0;`
- A function that returns pointer values can return a null pointer when it is unable to perform its task.

1.4.4 Generic Pointers

- A generic pointer is a pointer variable that has void as its data type.
- The void pointer, or the generic pointer, is a special type of pointer that can point to variables of any data type.
- It is declared like a normal pointer variable but using the void keyword as the pointer's data type.

For example:

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

```

int x=10;
char ch = 'A';
void *gp;
gp = &x;
printf("\n Generic pointer points to the integer value = %d", *(int*)gp);
gp = &ch;
printf("\n Generic pointer now points to the character= %c", *(char*)gp);
return 0;
}

```

Output

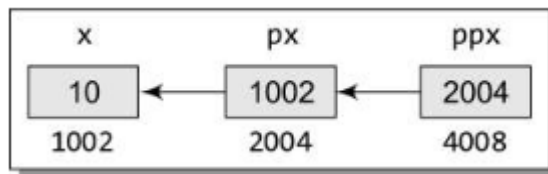
Generic pointer points to the integer value = 10

Generic pointer now points to the character = A

1.4.5 Pointer to Pointers

In C, we can also use pointers that point to pointers.

- The pointers in turn point to data or even to other pointers.
- To declare pointers to pointers, just add an asterisk * for each level of reference.



For example, consider the following code:

```

int x=10;
int *px, **ppx;
px = &x;
ppx = &px;
printf("\n %d", **ppx);

```

Then, it would print 10, the value of x.

1.4.6 Drawbacks of Pointers

- If pointers are used incorrectly, they can lead to bugs that are difficult to unearth.
- For example, if a pointer is used to read a memory location but that pointer is pointing to an incorrect location, then we may end up reading a wrong value.
- An erroneous input always leads to an erroneous output.

Example:

```

int x, *px;
x=10;
*px = 20;

```

Error: Un-initialized pointer. px is pointing to an unknown memory location. Hence it will overwrite that location's contents and store 20 in it.

```

int x, *px;
x=10;
px = x;

```

Error: It should be px = &x;

1.5 Dynamic Memory Allocation

- The process of allocating memory to the variables during execution of the program or at run time is known as dynamic memory allocation.
- C language provides a mechanism of dynamically allocating memory so that only the amount of memory that is actually required is reserved.
- We reserve space only at the run time for the variables that are actually required.
- Dynamic memory allocation gives best performance in situations in which we do not know memory requirements in advance.

C language has four library routines which allow this function.

Function	Task
malloc()	Allocates memory and returns a pointer to the first byte of allocated space
calloc()	Allocates space for an array of elements, initializes them to zero and returns a pointer to the memory
free()	Frees previously allocated memory
realloc()	Alters the size of previously allocated memory

Memory Allocations process

- The free memory region is called heap.
- The size of heap is not constant as it keeps changing when the program is executed.
- Some new variables are created and some variables cease to exist when the block in which they were declared is exited.
- When an overflow condition occurs, the memory allocation functions will return a null pointer.

Allocating a block of Memory

1. malloc()

- malloc is declared in <stdlib.h>
- The above header file is included in any program that calls malloc.
- The malloc function reserves a block of memory of specified size and returns a pointer of type void.

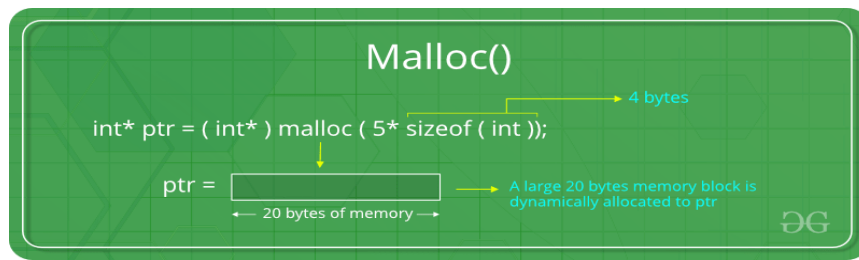
The general syntax of malloc() is : **ptr=(cast-type*)malloc(byte-size);**

where ptr is a pointer of type cast-type. malloc() returns a pointer (of cast type) to an area of memory with size byte-size.

Eg: **arr=(int*)malloc(10*sizeof(int));**

dynamically allocates memory equivalent to 10 times the area of int bytes.

space is reserved and the address of the first byte of memory allocated is assigned to the pointer arr of type int.



Example of malloc()- Program: Write a program to read and display values of an integer array. Allocate space dynamically for the array.

```
#include <stdio.h>

#include <stdlib.h>

int main()

{

int i, n;

int *arr;

printf("\n Enter the number of elements ");

scanf("%d", &n);

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

if(arr == NULL)

{

printf(" \n Memory Allocation Failed");

exit(0);

}

for(i = 0;i < n;i++)

{

printf("\n Enter the value %d of the array: ", i);

scanf("%d", &arr[i]);

}

printf("\n The array contains \n");

for(i = 0;i < n;i++)
```

```
printf("%d", arr[i]);

return 0;

}
```

2.calloc()

calloc() function is a function that reserves memory at the run time.

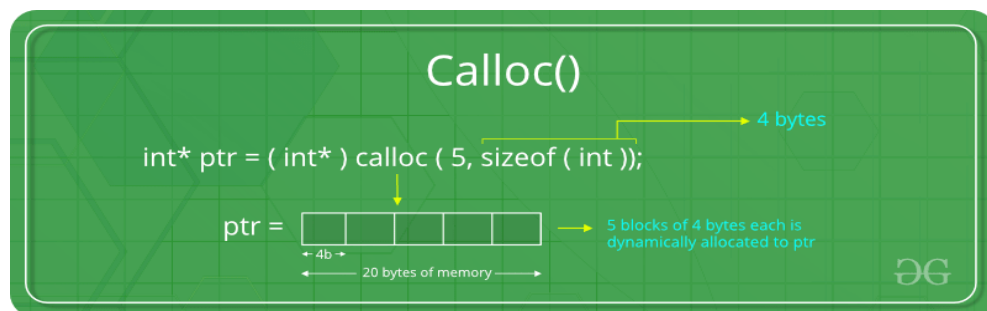
It is used to request multiple blocks of storage each of the same size and then sets all bytes to zero.

calloc() stands for contiguous memory allocation and is primarily used to allocate memory for arrays.

The syntax of calloc() can be given as:

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

The above statement allocates contiguous space for n blocks each of size elem-size bytes.



The difference between malloc() and calloc() is that when we use calloc(), all bytes are initialized to zero.

calloc() returns a pointer to the first byte of the allocated region.

When memory is allocated using malloc() or calloc(), a NULL pointer will be returned if there is not enough space in the system to allocate.

A NULL pointer, points nowhere.

Eg. A call to malloc, with an error check:

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

if(ip == NULL)

{

printf("\n Memory could not be allocated");
```

```
return;  
  
}
```

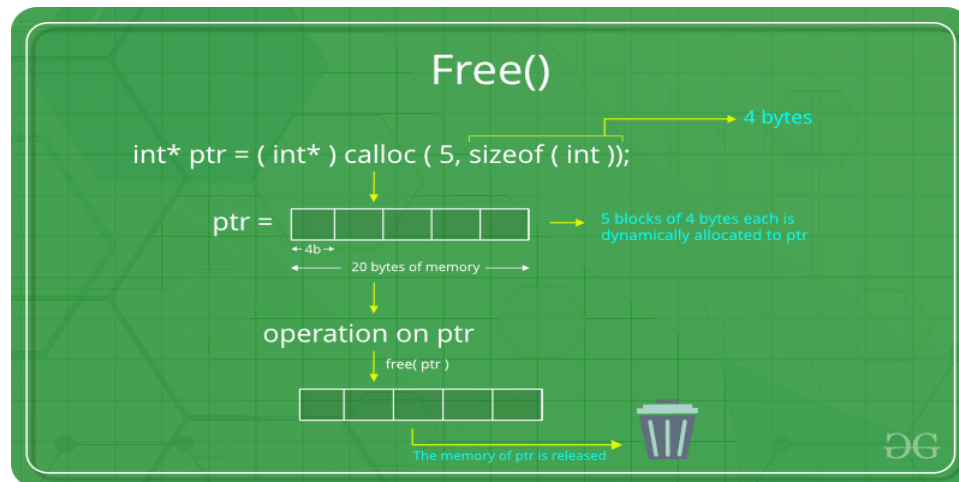
Example of calloc()- Program: Demonstrates the use of calloc() to dynamically allocate space for an integer array.

```
#include <stdio.h>  
  
#include <stdlib.h>  
  
int main ()  
{  
  
    int i,n;  
  
    int *arr;  
  
    printf ("\n Enter the number of elements: ");  
  
    scanf ("%d",&n);  
  
    arr = (int*) calloc(n,sizeof(int));  
  
    if (arr==NULL)  
  
        exit (1);  
  
    printf("\n Enter the %d values to be stored in the array", n);  
  
    for (i = 0; i < n; i++)  
  
        scanf ("%d",&arr[i]);  
  
    printf ("\n You have entered: ");  
  
    for(i = 0; i < n; i++)  
  
        printf ("%d",arr[i]);  
  
    free(arr);  
  
    return 0;  
  
}
```

3.free() - Releasing the used space

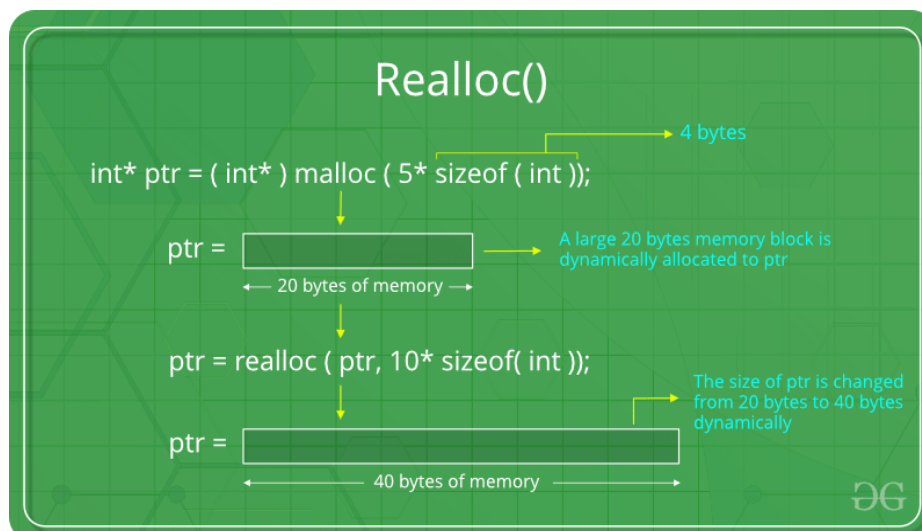
- Memory allocated during compile time for variables is automatically released by the system.

- Dynamically allocated memory must be manually released using the `free()` function.
- Releasing unused memory is crucial for efficient memory management, especially when storage space is limited.
- The `free()` function takes a pointer as an argument and returns the memory block pointed to by that pointer to the free list within the heap.
- The pointer passed to `free()` must have been previously allocated using `malloc()` or `calloc()`.



4.realloc()- To Alter the Size of Allocated Memory

- `Realloc()` is a function that can be used to change the size of memory that was previously allocated using `calloc()` or `malloc()`.
- The function takes two arguments: a pointer to the memory to be resized and the new size of the memory.
- `Realloc()` returns a pointer to the resized memory block, or `NULL` if the request fails.
- If `realloc()` was able to make the old block of memory bigger, it returns the same pointer.
- Otherwise, `realloc()` allocates a new block of memory and copies the old data to it.
- It is important to check if the pointer returned by `realloc()` is `NULL` before using it.



/*Example program for reallocation*/

```
#include <stdio.h>

#include <stdlib.h>

#define NULL 0

int main()

{

char *str;

str = (char *)malloc(10);

if(str==NULL)

{

printf("\n Memory could not be allocated");

exit(1);

}

strcpy(str,"Hi");

printf("\n STR = %s", str);

/*Reallocation*/

str = (char *)realloc(str,20);

if(str==NULL)

{

printf("\n Memory could not be reallocated");

exit(1);

}

printf("\n STR size modified\n");

printf("\n STR = %s\n", str);

strcpy(str,"Hi there");

printf("\n STR = %s", str);
```

```
/*freeing memory*/

free(str);

return 0;

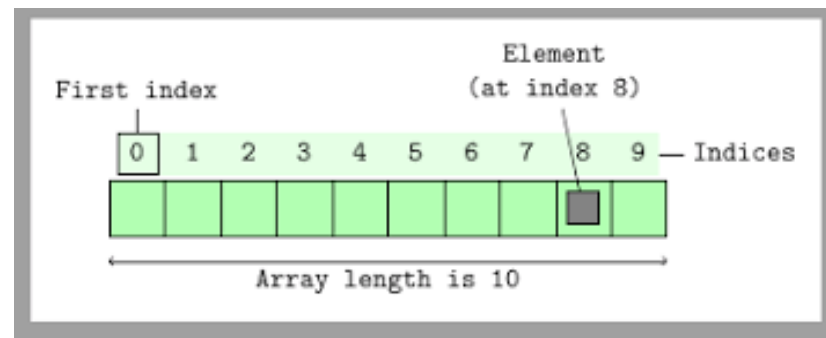
}
```

Difference between malloc() and calloc()

Malloc	Calloc
The name malloc stands for memory allocation.	The name calloc stands for contiguous allocation.
malloc () doesn't initialize the allocated memory. It contains garbage values	calloc () initializes the allocated memory to zero
Since no initialization, time efficiency is greater than calloc()	More expensive because of zero fillings. But convenient than malloc
Syntax variable_name=(datatype*)malloc(sizeof(datatype));	Syntax variable_name=(datatype*)calloc(n,size);
Required no. of bytes to be allocated is specified as arguments. i.e., size in bytes	Takes 2 arguments: n→no. of blocks to be allocated, size→no. of bytes in each block

1.6 Arrays

- An array data structure, or simply an array, is a data structure consisting of a collection of (mainly of similar data types) elements (values or variables), each identified by at least one array index or key. An array is stored so that the position of each element can be computed from its index tuple by a mathematical formula.
- An array is a collection of items stored at contiguous memory locations.



1.6.1 Linear arrays

- Linear arrays** are called **one dimensional arrays** which is referenced by one subscript. A 1-D array is a collection of similar data elements where each element is referenced by one subscripts.

- An array must be declared before being used. Declaring an array means specifying the following:
 1. **Data_type**—the kind of values it can store, for example, int, char, float, double.
 2. **Name**—to identify the array.
 3. **Size**—the maximum number of values that the array can hold.
- Arrays are declared using the following syntax:

Data_type Name[Size];

- The type can be either int, float, double, char, or any other valid data type.
- The number within brackets indicates the size of the array, i.e., the maximum number of elements that can be stored in the array.
- For example, if we write, int marks[10]; The above statement declares an array marks that contains 10 elements. In C, the array index starts from zero. This means that the array marks will contain 10 elements in all. The first element will be stored in marks[0], second element in marks[1], so on and so forth. Therefore, the last element, that is the 10th element, will be stored in marks[9].

1 st eleme nt	2 nd eleme nt	3 rd eleme nt	4 th eleme nt	5 th eleme nt	6 th eleme nt	7 th eleme nt	8 th eleme nt	9 th eleme nt	10 th eleme nt
Mark s[0]	Mark s[1]	Mark s[2]	Mark s[3]	Mark s[4]	Mark s[5]	Mark s[6]	Mark s[7]	Mark s[8]	Mark s[9]

Calculating the address of array Elements

- Since an array stores all its data elements in consecutive memory locations, storing just the base address, that is the address of the first element in the array, is sufficient. The address of other data elements can simply be calculated using the base address. The formula to perform this calculation is, Address of data element

$$A[k] = BA(A) + w(k - \text{lower_bound})$$

or

$$A[k] = \alpha + k * \text{sizeof}(\text{int}) \quad \text{where } \alpha = \text{base address}$$

- Here, A is the array, k is the index of the element of which we have to calculate the address, BA is the base address of the array A, and w is the size of one

element in memory, for example, size of int is 4 .Ex: Given an array int marks[] = {99,67,78,56,88,90,34,85}, calculate the address of marks[4] if the base address = 1000.

99	67	78	56	88	90	34	85
Marks[0]	Marks[1]	Marks[2]	Marks[3]	Marks[4]	Marks[5]	Marks[6]	Marks[7]
]]]]]]]]
1000	1002	1004	1006	1008	1010	1012	1014

$$\text{marks}[4] = 1000 + 4(4 - 0) = 1000 + 4(4) = 1016$$

or

$$\text{marks}[4] = 1000 + 4 * 4 = 1016$$

Calculating the length of array Elements

The length of an array is given by the number of elements stored in it. The general formula to calculate the length of an array is:

$$\text{Length} = \text{upper_bound} - \text{lower_bound} + 1$$

Where upper_bound is the index of the last element and lower_bound is the index of the first element in the array.Ex: in the above diagram,

$$\text{length of the array} = 7 - 0 + 1 = 8$$

Initialization of one-dimensional array

- Arrays can be initialized at declaration time as:

```
int age[5]={2,4,34,3,4};
```

- It is not necessary to define the size of arrays during initialization. Below instruction justifies it

```
int age[]={2,4,34,3,4};
```

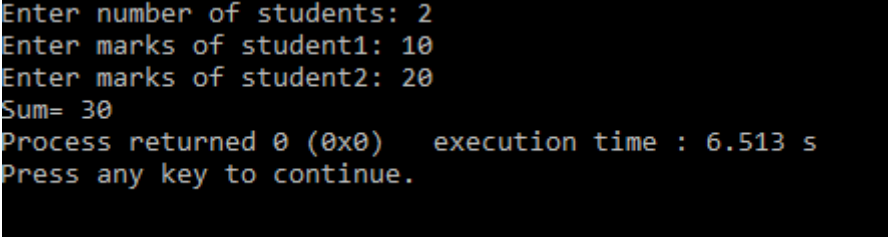
- In this case, the compiler determines the size of array by calculating the number of elements of an array.
- An array can be initialized by inputting values from the keyboard. In this method, a while/do-while or a for loop is executed to input the value for each element of the array.

```
int i, marks[10];
for(i=0;i<10;i++)
    scanf("%d", &marks[i]);
```


Example: C program calculates the sum of marks of n students using arrays.

```
#include <stdio.h>

int main(){
    int marks[10],i,n,sum=0;
    printf("Enter number of students: ");
    scanf("%d",&n);
    for(i=0;i<n;i++){
        printf("Enter marks of student%d: ",i+1);
        scanf("%d",&marks[i]);
        sum+=marks[i];
    }
    printf("Sum= %d",sum);
    return 0;
}
```



A screenshot of a terminal window showing the execution of the C program. The output is as follows:

```
Enter number of students: 2
Enter marks of student1: 10
Enter marks of student2: 20
Sum= 30
Process returned 0 (0x0)   execution time : 6.513 s
Press any key to continue.
```

1.7 Dynamically Allocated Array

If the array size is decided during run time, then it is called dynamically allocated array.

1.7.1 1-D Array:

- Consider # define max[100];
- Now 100 locations has been created. If we want more locations, we increase the size of max in definition and recompile the program. But if we set it very large, there occurs wastage of space, or if we make it small, sometimes our data may not fit in that. A good solution for this is to do run time allocation of array.
- **Example:**

```
int *list, n;

printf("enter the no. of elements to generate");
scanf("%d",&n);
```

```

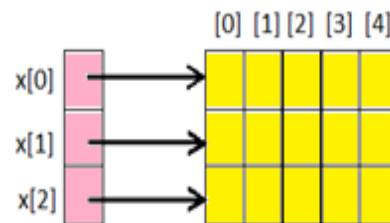
if(n<1)
{   printf("improper value of n");
    exit(0);
}
list=(int*)malloc(n*sizeof(int));

```

- Now if $n < 1$, it will come out of the program, else, it will create $n \times 4$ bytes during run time. i.e, if $n=3$, we need to generate 3 memory location for integer no's (12 bytes)

1.7.2 Two-D Array:

- C uses array of array representation to represent a multi-dimensional array. Normally, a pointer contains the address of a variable. A pointer-to-pointer contains address of another pointer. When we define a **pointer to a pointer**, the first pointer contains the address of the second pointer, which points to the location that contains the actual value as shown below.
 - `int a;` → allocates memory for variable 'a' where integer can be stored
 - `int *p1;` → allocates memory for variable 'p1' where address of an integer can be stored
 - `int **p2;` → allocates a memory for variable 'p2' where address of pointer can be stored
- A 2-D array represented as a 1-D array of pointers where each pointer contains address of 1-D array. ex: `int x[3][5]`; The figure shows array to array representation, where each of 3 pointers points to 1-D array consisting of 5 locations



- **Example:**

```

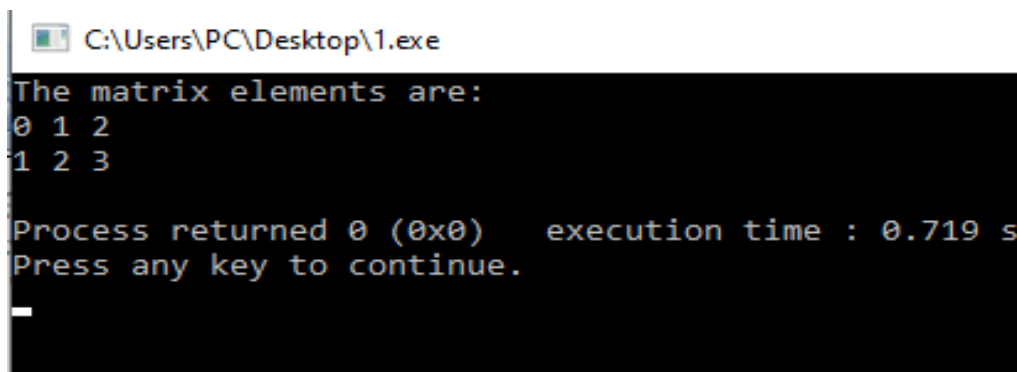
#include <stdio.h>
#include <stdlib.h>
int main() {

```

```

int row = 2, col = 3;
int *arr = (int *)malloc(row * col * sizeof(int));
int i, j;
for (i = 0; i < row; i++)
    for (j = 0; j < col; j++)
        *(arr + i * col + j) = i + j;
printf("The matrix elements are:\n");
for (i = 0; i < row; i++) {
    for (j = 0; j < col; j++)
    {
        printf("%d ", *(arr + i * col + j));
    }
    printf("\n");
}
free(arr);
return 0;
}

```



```

C:\Users\PC\Desktop\1.exe
The matrix elements are:
0 1 2
1 2 3

Process returned 0 (0x0)   execution time : 0.719 s
Press any key to continue.

```

1.8 Structures

- Structure is a user defined data type that can hold data items of different data types.
- The major difference between a structure and an array is that an array can store only information of same data type.

Syntax

```

struct{
    member1;

```

```
        member2;  
        .....  
        member n;  
}; structurename
```

Example

```
struct {  
    char name[10];  
    int age;  
    float salary;  
} person;
```

- The above example creates a structure and variable name is Person and that has three fields:

name = a name that is a character array

age = an integer value representing the age of the person

salary = a float value representing the salary of the individual

- **Assign values to fields**to assign values to the fields, use. (dot) as the structure member operator. This operator is used to select a particular member of the structure

Ex: strcpy(Person.name, "james");

Person.age = 10;

Person.salary = 35000;

1.8.1 Structure Declaration

- Structure definition with a tag name-**Tagged Structure**

Syntax:

```
Struct structure_name  
{  
    member1;  
    member2;  
    ...  
    member n;  
};
```

Example:

```
struct person  
{
```

```

char name[10];
int age;
float salary;
};

```

- **Example1: Program that uses a simple structure to store the student marks details**

```

#include <stdio.h>
#include <string.h>
struct student
{
char name[20], subject[20];
float percentage;
}s1;
int main()

{
strcpy(s1.name, "aditi");
strcpy(s1.subject, "Maths");
s1.percentage = 91.25;
printf(" Name      : %s \n", s1.name);
printf(" Subject   : %s \n", s1.subject);
printf(" Percentage : %f \n", s1.percentage);
return 0;
}

```

```

Name      : aditi
Subject   : Maths
Percentage : 91.250000

Process returned 0 (0x0)   execution time : 0.125 s
Press any key to continue.

```

- **Type Defined Structure:** We can create our own structure data type by using typedef statement as:

```

typedef struct
{
data_type member 1;
data_type member 2;

```

```
.....  
.....  
data_type member n;  
} TypeName;
```

- **typedef** is the keyword used at the beginning of the definition and by using typedef user defined data type can be obtained.
- **struct** is the keyword which tells structure is defined to the compiler
- The members are declared with their data_type
- **Type name** is not a variable; it is user defined data_type

Method-1:

```
typedef struct person{  
    char name[10];  
    int age;  
    float salary;  
};  
person person1, person2;
```

Method-2:

```
typedef struct {  
    char name[10];  
    int age;  
    float salary;  
} person;  
person person1, person2;
```

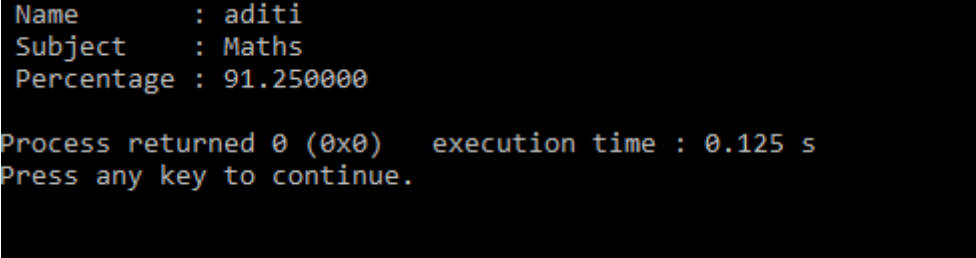
- In above example, **person** is the name of the type and it is a user defined data type. Declarations of structure variables: **person person1, person2;**
- This statement declares the variable **person1** and **person2** are of type **person**.
- **Example2:Program that uses a simple structure to store the student marks details using typedef structure**

```
#include <stdio.h>  
#include <string.h>  
typedef struct  
{  
    char name[20], subject[20];  
    float percentage;
```

```
}students;

int main()

{
students s1;
strcpy(s1.name, "aditi");
strcpy(s1.subject, "Maths");
s1.percentage = 91.25;
printf(" Name      : %s \n", s1.name);
printf(" Subject   : %s \n", s1.subject);
printf(" Percentage : %f \n", s1.percentage);
return 0;
}
```



```
Name      : aditi
Subject   : Maths
Percentage : 91.250000

Process returned 0 (0x0)   execution time : 0.125 s
Press any key to continue.
```

Example3:Program that uses a simple structure to store the student details.

```
#include <stdio.h>

struct student
{
    char name[30];
    int rollno;
    int t_marks;
};

void main()
{
    int num,i;

    struct student std[10]; //statement declares array of structure
    printf("enter the number of students:");
    scanf("%d",&num);
    for(i=0;i<num;i++)
    {
```

```

printf("\n enter the details of student %d",i+1);
printf("\n name:");
scanf("%s",std[i].name);
printf("\n Rollno:");
scanf("%d",&std[i].rollno);
printf("\n totalmarks:");
scanf("%d",&std[i].t_marks);
}
printf("\n the student details are:");
for(i=0;i<num;i++)
    printf("\n student %d \n name %s \n Rollno %d\n Total marks
%d\n",i+1,std[i].name,std[i].rollno,std[i].t_marks);
}

```

```

enter the number of students:2

enter the details of student 1
name:aditya

Rollno:01

totalmarks:99

enter the details of student 2
name:aditi

Rollno:02

totalmarks:89

the student details are:
student 1
name aditya
Rollno 1
Total marks 99

student 2
name aditi
Rollno 2
Total marks 89

Process returned 2 (0x2)   execution time : 26.570 s
Press any key to continue.

```

1.9 Unions

Union is a derived data type, like structure, i.e. collection of elements of different data types which are grouped together. Each element in a union is called member. Union

allocates one common storage space for all its members, or memory space is shared between its members. So only one field of union is active at any given time

Syntax:

Method-1:

```
union tagname
{
    Type1 member1;
    Type2 member2;
    ....
};
```

Method-2:

```
typedef union
{
    Type1 member1;
    Type2 member2;
    ....
};
```

- Similar to structures, a union is a collection of variables of different data types. The only difference between a structure and a union is that in case of unions, you can only store information in one field at any one time.
- To better understand a union, think of it as a chunk of memory that is used to store variables of different types. When a new value is assigned to a field, the existing data is replaced with the new data.
- Thus, unions are used to save memory. They are useful for applications that involve multiple members, where values need not be assigned to all the members at any one time. Table below shows the difference between structures and unions.

Basis of comparison	Structure	Union
Basic	The separate memory location is allotted to each member of the 'structure'.	All members of the 'union' share the same memory location.
Declaration	<pre>struct struct_name{ type element1; type element2; } variable1, variable2, ...;</pre>	<pre>union u_name{ type element1; type element2; } variable1, variable2, ...;</pre>
Keyword	'struct'	'union'

Size	Size of Structure= sum of size of all the data members.	Size of Union=size of the largest members.
Store Value	Stores distinct values for all the members.	Stores same value for all the members.
At a Time	A 'structure' stores multiple values, of the different members, of the 'structure'.	A 'union' stores a single value at a time for all members.
Way of Viewing	Provide single way to view each memory location.	Provide multiple way to view same memory location.

- **Example**

```
#include <stdio.h>
#include <string.h>
union student
{
char name[20], subject[20];
float percentage;
}s1;
int main()
{
strcpy(s1.name, "aditi");
strcpy(s1.subject, "Maths");
s1.percentage = 91.25;
printf(" Name      : %s \n", s1.name);
printf(" Subject   : %s \n", s1.subject);
printf(" Percentage : %f \n", s1.percentage);
return 0;
}
```

```
Name      :
Subject   :
Percentage : 91.250000

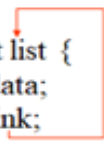
Process returned 0 (0x0)   execution time : 0.219 s
Press any key to continue.
```

1.10 Self- Referential Structure

- Self-referential structures are those structures that contain a reference to the data of its same type. That is, a self-referential structure, in addition to other data, contains a pointer to a data that is of the same type as that of the structure.

- For example, consider the structure node given below. Here, the structure node will contain two types of data: a character data and a pointer link. The value of link is either the address in the memory of an instance of a list or the null pointer.

```
typedef struct list {
    char data;
    list *link;
};
```



- Consider these statements which create 3 structures and assign values to their respective fields:



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

- Structures item1, item2 and item3 each contains the data item a, b and c and the null pointer. We can attach these structures together by replacing the null link field in item 2 with one that points to item 3 and by replacing the null link field in item 1 with one that points to item2.

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

1.11 Polynomials

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

For ex, $A(x)=3x^{20}+2x^5+4$ and $B(x)=x^4+10x^3+3x^2+1$. The largest (or leading) exponent of a polynomial is called its degree.

- Assume that we have 2 polynomials, $A(x)=\sum a_i x^i$ & $B(x)=\sum b_i x^i$, then $A(x)+B(x)=\sum (a_i + b_i) x^i$. i.e the above sum would be $S(x)=3x^{20}+2x^5+ x^4+10x^3+3x^2+5$
- A polynomial thus may be represented using arrays or linked lists. If any term is not present, then we assign 0's in the corresponding coefficient.
- A structure may be defined such that it contains two parts- one is the coefficient and second is the corresponding degree. Given below is the polynomial representation using structure.

```
#define max_degree 100
```

```
typedef struct
```

```
{
    float coef[max_degree];
    int degree;
}polynomial;
```

- Array of Structure:** Instead of using one array for each polynomial, we use one array to store all polynomials, which saves space

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

	<i>starta</i>	<i>finisha</i>	<i>startb</i>		<i>finishb</i>	<i>avail</i>
	↓	↓	↓		↓	↓
<i>coef</i>	2	1	1	10	3	1
<i>exp</i>	1000	0	4	3	2	0
	0	1	2	3	4	5

- Polynomial Addition:** Suppose **a** is one polynomial and **b** is another polynomial then $c=a+b$, where **c** is the addition of polynomial **a** and **b**. inorder to add 2 polynomials, there are different cases to be verified:

Case1: Power of Polynomial a is Equal to Power of Polynomial b

Ex: $a = 25x^6+10x^5+7x^2+9$ $b = 15x^6+5x^4+4x^3$
 lead exponent(a)=6 lead exponent(b)=6
 so, lead exponent(a)=lead exponent(b)
 sum is,

$$25x^6 + 10x^5 + 7x^2 + 9$$

$$\underline{15x^6 + 5x^4 + 4x^3}$$

$$c = \underline{40x^6 + \dots}$$

Case 2: Power of Polynomial a is Greater than Power of Polynomial b

Remaining polynomial is: $a = 10x^5 + 7x^2 + 9$ lead exponent(a)=5

$$b = 5x^4 + 4x^3 \quad \text{lead exponent(b)=4}$$

so, lead exponent(a) > lead exponent(b). so copy lead exponent(a) directly to c sum is,

$$10x^5 + 7x^2 + 9$$

$$\underline{5x^4 + 4x^3}$$

$$c = \underline{10x^5 + \dots}$$

Case 3: Power of Polynomial a is Less than Power of Polynomial b

Remaining polynomial is: $a = 7x^2 + 9$ lead exponent(a)=2

$$b = 5x^4 + 4x^3 \quad \text{lead exponent(b)=4}$$

so, lead exponent(a) < lead exponent(b). so copy lead exponent(b) directly to c sum is,

$$7x^2 + 9$$

$$\underline{5x^4 + 4x^3}$$

$$c = \underline{5x^4 + \dots}$$

so final polynomial is: **$40x^6 + 10x^5 + 5x^4 + 4x^3 + 7x^2 + 9$**

• **Coding:**

/ d = a + b, where a, b, and d are polynomials */*

d = Zero()

while (! IsZero(a) && ! IsZero(b)) do {

switch COMPARE (Lead_Exp(a), Lead_Exp(b)) {

case -1: d = Attach(d, Coef (b, Lead_Exp(b)), Lead_Exp(b));

b = Remove(b, Lead_Exp(b));

break;

case 0: sum = Coef (a, Lead_Exp (a)) + Coef (b, Lead_Exp(b));

if (sum) {

Attach (d, sum, Lead_Exp(a));

a = Remove(a , Lead_Exp(a));

b = Remove(b , Lead_Exp(b));

```

    }
    break;
case 1: d = Attach(d, Coef(a, Lead_Exp(a)), Lead_Exp(a));
        a = Remove(a, Lead_Exp(a));
    }
}
insert any remaining terms of a or b into d

```

1.12 Sparse Matrix

- A sparse matrix is a matrix in which most of the elements are zero. By contrast, if most of the elements are nonzero, then the matrix is considered dense. Sparse data is by nature more easily compressed and thus require significantly less storage. If we apply the operations using standard matrix structures and algorithms to sparse matrices, then the execution will slow down and the matrix will consume large amount of memory
- When a sparse matrix is represented as a 2-dimensional array, we waste space. For ex, if 100*100 matrixes contain only 100 non-zero entries then we waste 9900 out of 10000 memory spaces. Solution is to Store only the non-zero elements.
- A matrix is typically stored as a two-dimensional array. Each entry in the array represents an element a_{ij} of the matrix and is accessed by the two indices i and j . Conventionally, i is the row index, numbered from top to bottom, and j is the column index, numbered from left to right. For an $m \times n$ matrix, the amount of memory required to store the matrix in this format is proportional to $m \times n$. In the case of a sparse matrix, substantial memory requirement reductions can be realized by storing only the non-zero entries.
- **Sparse Matrix Representstion:**
A sparse matrix can be represented by using TWO representations:

1. Triplet Representation

2. Linked Representation

- **Triplet Representation**

```

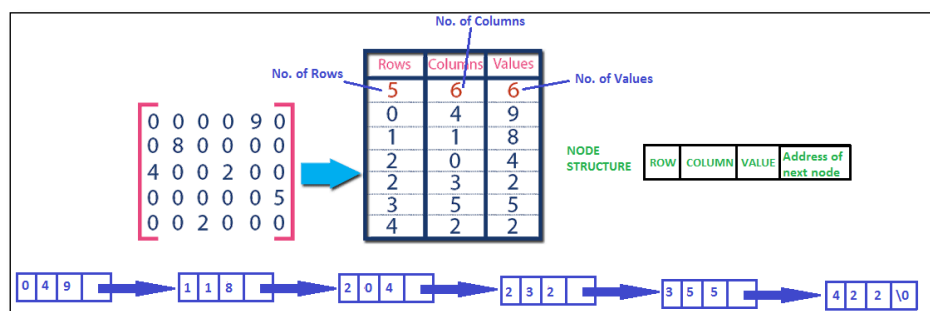
#define max 101
typedef struct
{
    int col, row, val;

```

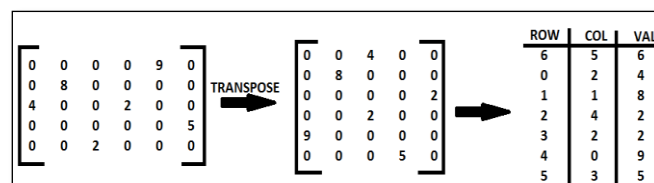
}term;

term a[max];

- In this representation, we consider only non-zero values along with their row and column index values. Each non zero value is a triplet of the form $\langle R, C, Value \rangle$ where R represents the row in which the value appears, C represents the column in which the value appears and Value represents the nonzero value itself. In this representation, the 0th row stores total rows, total columns and total non-zero values in the matrix. For example, consider a matrix of size 5 X 6 containing 6 number of non-zero values. This matrix can be represented as shown in the fig.



- In above example matrix, there are only 6 non-zero elements (those are 9, 8, 4, 2, 5 & 2) and matrix size is 5 X 6. We represent this matrix as shown in the above image. Here the first row in the right side table is filled with values 5, 6 & 6 which indicate that it is a sparse matrix with 5 rows, 6 columns & 6 non-zero values. Second row is filled with 0, 4, & 9 which indicates the value in the matrix at 0th row, 4th column is 9. In the same way the remaining non-zero values also follows the similar pattern.
- To transpose a matrix** we just interchange the rows and columns. This means that each element $a[i][j]$ in the original matrix becomes $b[j][i]$ in the transpose matrix (Fig 2.11).



We can write the algorithm as:

For each row i

Take element $\langle i, j, val \rangle$

transpose it as element <j, i, val>

- The product of 2 sparsematrixes may no longer be sparse. For example see the Fi

$$\begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

Algorithm to find transpose of sparse matrix:

```
void transpose(term a[], term b[])
/* b is set to the transpose of a */
{
    int n,i,j, currentb;
    n = a[0].value;          /* total number of elements */
    b[0].row = a[0].col; /* rows in b = columns in a */
    b[0].col = a[0].row; /* columns in b = rows in a */
    b[0].value = n;
    if (n > 0 ) { /* non zero matrix */
        currentb = 1;
        for (i = 0; i < a[0].col; i++)
            /* transpose by the columns in a */
            for (j = 1; j <= n; j++)
                /* find elements from the current column */
                if (a[j].col == i) {
                    /* element is in current column, add it to b */
                    b[currentb].row = a[j].col;
                    b[currentb].col = a[j].row;
                    b[currentb].value = a[j].value;
                    currentb++;
                }
    }
}
```

Fast Transpose:

Create a much better algorithm by using a little more storage.

Transpose of a matrix can be represented as a sequence of triples in $O(\text{columns} + \text{elements})$ time.

Fast-transpose proceeds by first determining the number of elements in each column of the original matrix.

This gives us the number of elements in each row of the transpose matrix. From this information, we can determine the starting position of each row in the transpose matrix. We now can move the elements in the original matrix one by one into their correct position in the transpose matrix.


```

void fast_transpose(term a[], term b[])
{
    /* the transpose of a is placed in b */
    int row_terms[MAX_COL], starting_pos[MAX_COL];
    int i, j, num_cols = a[0].col, num_terms = a[0].value;
    b[0].row = num_cols; b[0].col = a[0].row;
    b[0].value = num_terms;
    if (num_terms > 0) { /* nonzero matrix */
        for (i = 0; i < num_cols; i++)
            row_terms[i] = 0;
        for (i = 1; i <= num_terms; i++)
            row_terms[a[i].col]++;
        starting_pos[0] = 1;
        for (i = 1; i < num_cols; i++)
            starting_pos[i] =
                starting_pos[i-1] + row_terms[i-1];
        for (i = 1; i <= num_terms; i++) {
            j = starting_pos[a[i].col]++;
            b[j].row = a[i].col; b[j].col = a[i].row;
            b[j].value = a[i].value;
        }
    }
}

```

- **In linked representation**, we use linked list data structure to represent a sparse matrix. In this linked list, we use two different nodes namely header node and element node. Header node consists of three fields and element node consists of five fields as shown in the fig.

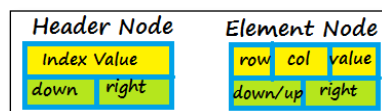
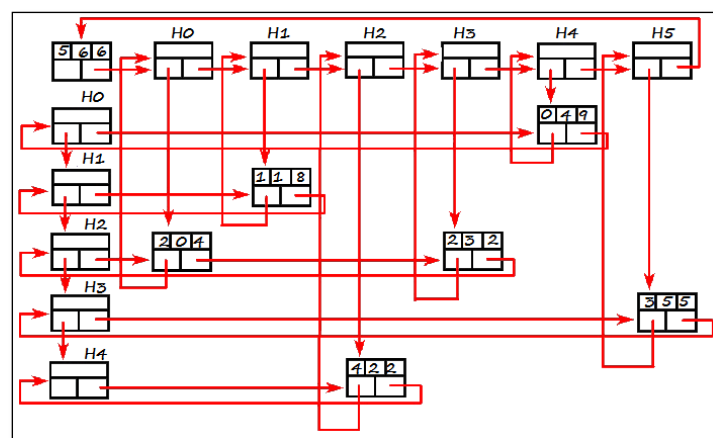


Fig.Header and element node representation



Linked list representation of sparse matrix

- Consider the sparse matrix used in the Triplet representation. This sparse matrix can be represented using linked representation as shown in the below image. In this representation, H0, H1..., H5 indicates the header nodes which are used to represent indexes. Remaining nodes are used to represent non-zero elements in the matrix, except the very first node which is used to represent abstract information of the sparse matrix (i.e., It is a matrix of 5 X 6 with 6 non-zero elements). In this representation, in each row and column, the last node right field points to its respective header node in the above figure.

1.13 Representation of Multi-Dimensional Arrays

- A multi-dimensional array can be termed as an array of arrays that stores homogeneous data in tabular form. Data in multidimensional arrays are stored in row-major order. A three-dimensional (3D) array is an array of arrays of arrays. In C programming an array can have two, three, or even ten or more dimensions. The maximum dimensions a C program can have depend on which compiler is being used. More dimensions in an array means more data is held, but also means greater difficulty in managing and understanding arrays.

Syntax: data_type array_name[size1][size2]....[sizeN];

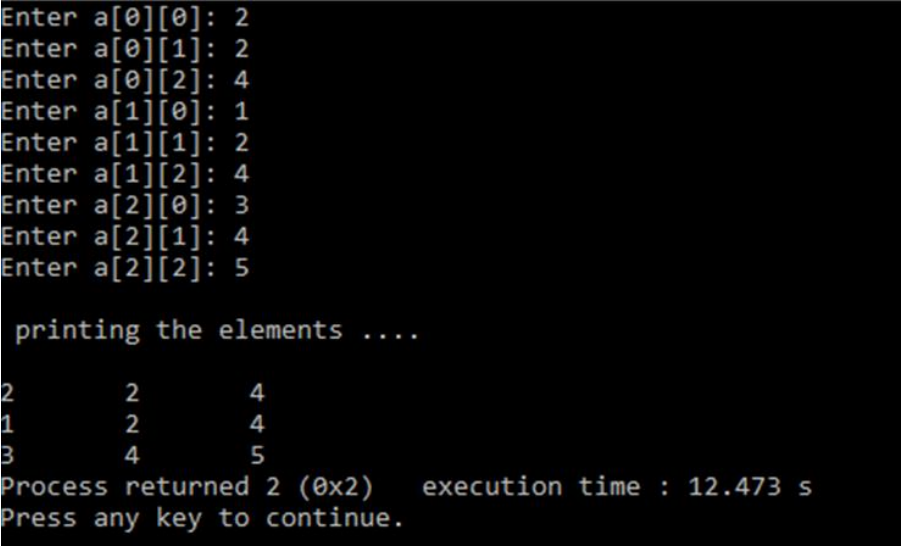
- Ex: Two dimensional array: int two_d[10][20];
- Three dimensional array: int three_d[10][20][30];

Example: 2-Dimensional Program to print the array contents

```
#include <stdio.h>

void main ()
{
    int arr[3][3],i,j;
    for (i=0;i<3;i++)
    {
        for (j=0;j<3;j++)
        {
            printf("Enter a[%d][%d]: ",i,j);
            scanf("%d",&arr[i][j]);
        }
    }
    printf("\n printing the elements ....\n");
    for(i=0;i<3;i++)
```

```
{  
    printf("\n");  
    for (j=0;j<3;j++)  
    {  
        printf("%d\t",arr[i][j]);  
    }  
}  
}
```



```
Enter a[0][0]: 2  
Enter a[0][1]: 2  
Enter a[0][2]: 4  
Enter a[1][0]: 1  
Enter a[1][1]: 2  
Enter a[1][2]: 4  
Enter a[2][0]: 3  
Enter a[2][1]: 4  
Enter a[2][2]: 5  
  
printing the elements ....  
  
2      2      4  
1      2      4  
3      4      5  
Process returned 2 (0x2)   execution time : 12.473 s  
Press any key to continue.
```

1.14 Strings

- A C string is a null-terminated character array.
- This means that a null character ('\0') is stored after the last character to signify the end of the string.
- For example, the string "HELLO" is stored in memory as HELLO'\0'.
- The name of the character array (or the string) is a pointer to the beginning of the string.
- If you declare a string as `char str[5] = "HELLO";`, the null character will not be appended automatically to the character array. This is because str can hold only 5 characters and the characters in HELLO have already filled the space allocated to it.

<pre>char str[] = "HELLO";</pre>	<pre>char ch = 'H';</pre> <p>Here H is a character not a string. The character H requires only one memory location.</p>
<pre>char str[] = "H";</pre> <p>Here H is a string not a character. The string H requires two memory locations. One to store the character H and another to store the null character.</p>	<pre>char str[] = "";</pre> <p>Although C permits empty string, it does not allow an empty character.</p>

Difference between character storage and string storage

Memory representation of character array

str[0]	1000	H
str[1]	1001	E
str[2]	1002	L
str[3]	1003	L
str[4]	1004	O
str[5]	1005	\0

- To access elements of a string, subscripts are used, starting from 0.
- Strings are stored in successive memory locations.
- To declare a constant string, initialize it with a value while declaring it.
- To declare a non-constant string, use the general form `char str[size];`.
- allocate enough space for the null character when allocating memory for a string.
- We can also initialize a string as an array of characters.
- If we declare a string with size much larger than the number of elements that are initialized, the compiler will automatically calculate the size based on the number of characters.
- The compiler creates an array of size 10; stores "HELLO" in it and finally terminates the string with a null character.

Pattern Matching:

Knuth, Morris, and Pratt have developed a pattern matching algorithm that has linear complexity. Using the example, suppose

pat = 'abcabcacab'

Let $s = s_0 s_2 \dots s_{m-1}$ be the string and assume that we are currently determining

whether or not there is a match beginning at s_i . If $s_i \neq a$ then, clearly, we may proceed by

comparing s_{i+1} and a. Similarly if $s_i = a$ and $s_{i+1} \neq b$ then we may proceed by compar

ing s_{i+1} and a . If $s_i s_{i+1} = ab$ and $S_{i+2} \neq c$ then we have the situation:

$s =$	'	-		a	b	?	?	?	?
$pat =$				a	b	c	a	b	c	a	c	a	b

The ? implies that we do not know what the character in s is.

The first ? in s represents S_{i+2} and $S_{i+2} \neq c$.

we may continue the search for a match by comparing the first character in pat with S_{i+2}

$s =$	'	-		a	b	c	a	?	?	.	.	.	?
$pat =$				a	b	c	a	b	c	a	c	a	b

We observe that the search for a match can proceed by comparing S_{i+4} and the second character in pat , b . This is the first place a partial match can occur by sliding the pattern pat towards the right. Thus, by knowing the characters in the pattern and the position in the pattern where a mismatch occurs with a character in s we can determine where in the pattern to continue the search for a match without moving backwards in s . To formalize this, we define a failure function for a pattern.

Definition: If $p = p_0 p_1 \dots p_{n-1}$ is a pattern, then its *failure function*, f , is defined as:

$$f(j) = \begin{cases} \text{largest } i < j \text{ such that } p_0 p_1 \dots p_i = p_{j-i} p_{j-i+1} \dots p_j & \text{if such an } i \geq 0 \text{ exists} \\ -1 & \text{otherwise} \end{cases}$$

For the example pattern, $pat = abcabcacab$, we have:

j	0	1	2	3	4	5	6	7	8	9
pat	a	b	c	a	b	c	a	c	a	b
f	-1	-1	-1	0	1	2	3	-1	0	1

KMP (Knuth Morris Pratt) Algorithm:

```
#include <stdio.h>
#include <string.h>
#define max_string_size 100
#define max_pattern_size 100
int pmatch();
void fail();
int failure[max_pattern_size];
char string[max_string_size];
char pat[max_pattern_size];



---


int pmatch(char *string, char *pat)
{
    /* Knuth, Morris, Pratt string matching algorithm */
    int i = 0, j = 0;
    int lens = strlen(string);
    int lenp = strlen(pat);
    while ( i < lens && j < lenp ) {
        if (string[i] == pat[j]) {
            i++; j++;
        }
        else if (j == 0) i++;
        else j = failure[j-1]+1;
    }
    return ( (j == lenp) ? (i-lenp) : -1);
}



---


```

Computing Failure function:

```


---


void fail(char *pat)
{
    /* compute the pattern's failure function */
    int n = strlen(pat);
    failure[0] = -1;
    for (j=1; j < n; j++) {
        i = failure[j-1];
        while ((pat[j] != pat[i+1]) && (i >= 0))
            i = failure[i];
        if (pat[j] == pat[i+1])
            failure[j] = i+1;
        else failure[j] = -1;
    }
}



---


```

1.15 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 is an ordered list in which all insertions and deletions are made at one end, called the top. Hence, a stack is called a LIFO (Last-In-First-Out) data structure, as the element that was inserted last is the first one to be taken out.
- For example, a pile of plates where one plate is placed on top of another (Fig 1). Now, when you want to remove a plate, you remove the

topmost plate first. Hence, you can add and remove an element (i.e., a plate) only at/from one position which is the topmost position.

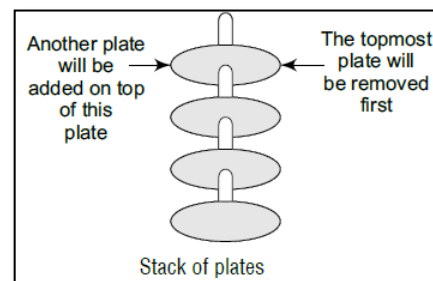


Fig. 1: Example for Stack of Plate

Array Representation of Stacks

- Every stack has a variable called TOP associated with it, which is used to store the address of the topmost element of the stack. It is this position where the element will be added to or deleted from.
- There is another variable called MAX, which is used to store the maximum number of elements that the stack can hold.
- If $TOP = NULL$, then it indicates that the stack is empty and if $TOP = MAX - 1$, then the stack is full. (Since array indices start from 0, it is $MAX - 1$, instead of MAX).

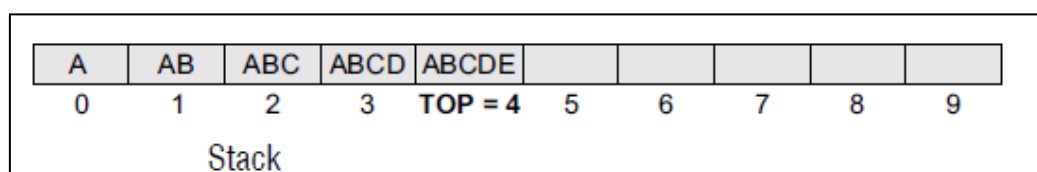


Fig. 2: Array Representation of Stack

Stack Operations

Various operations in stack are:

1. **Push:** Element is inserted into the top of stack using push operation.
2. **Pop:** Element is deleted from the top of stack using pop operation.
3. **Overflow:** checks if the stack is full or not.
4. **Underflow:** checks if the stack is empty or not.

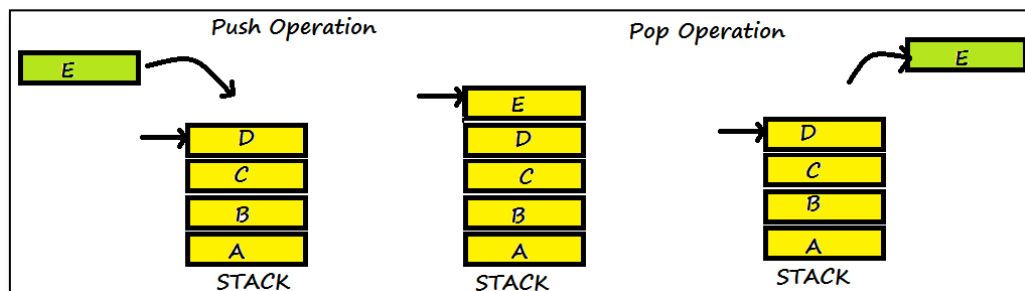


Fig. 3: Stack operations

a. Stack Create

It creates stack `stack[]` with size MAX (here defined as 5)

```
#define MAX 5
```

Stack can be declared as:

```
int stack[max];
```

`top` is a variable which points to stack.

Initially it is pointing to -1 (Fig 4).

```
int top=-1;
```

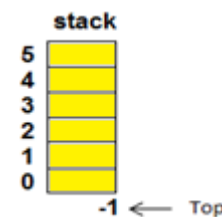


Fig. 4: Empty Stack

b. Push and Stack Overflow

- Inserting an element to stack is called push operation.
- The element can be inserted only from one end of the stack called top of stack. Initially as we know that `top` will be pointing to -1. This is called empty stack.
- Suppose we want to insert 1, 2, 3, 4 and 5 into a stack. Element 1 needs to be inserted to `stack[0]`. So we need to increment `stack[top]` by 1.

```
top = top+1;
```

Now insert the item 10 into that position.

```
stack[top]=item;
```


- To insert next element, top needs to be incremented once again and insert the element.
- We can repeat the same operation until stack is full. Now if we try to push one more element say 6, it is not possible, since stack is full.
- This situation is called stack overflow. i.e. when the stack is full and if we try to insert one more new element into stack, then it becomes stack overflow (fig.5).

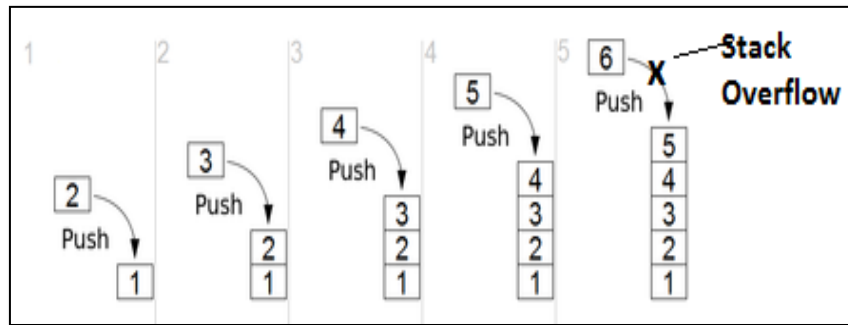


Fig. 5: Push and Overflow

Algorithm for push operation

1. Start
2. Initialize $top = -1$, set max as maximum no. of elements that can be inserted into stack,
3. If $top = \text{max} - 1$, Display Stack overflow
4. Else (perform push operation)
 - Read the element to be pushed (item)
 - Increment top by 1.
 - Copy num to $\text{stack}[top]$.
5. To push more elements, go to step 3 else go to step 6
6. Stop

Program code for push operation

```
void push(int stack[], int item)
{
    if (top == (MAX-1))
        printf("\n\nStack is Overflow");
    }
    else
    {
        top=top+1;
        stack[top] = item;
    }
}
```

c. **Pop and Stack Underflow**

- Deleting an element from the stack is called pop operation. Only one element can be deleted at a time and it is from the top of the stack.
- Element 5 is at top most position and needs to be deleted first. Then the pointer top needs to be pointed to previous position. When the top points to -1, which means there is no element in the stack. When we try to delete element when stack is empty, underflow occurs(Fig 6).



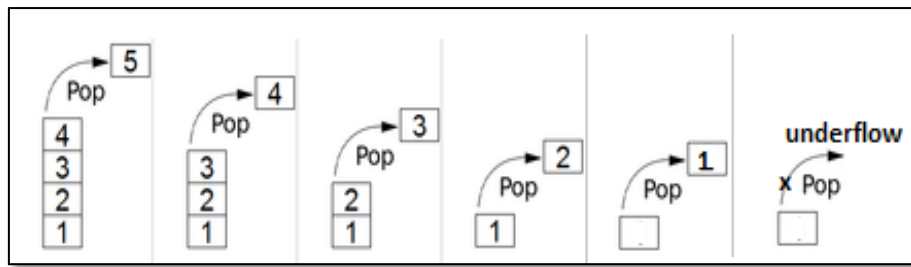


Fig. 6: Pop and Underflow

Algorithm for pop operation:

1. Start
2. if $\text{top} = -1$, Display Stack underflow
3. else (perform pop operation)
 - Copy $\text{stack}[\text{top}]$ to num
 - Decrement top by 1
4. To pop more elements, go to step 2 else go to step 5
5. Stop

Program code for pop operation

```
int pop(int stack[])
{
    int num;
    if(top == -1)
        printf("\n\nStack is Underflow");
    else
    {
        num = stack[top];
        top--;
        printf("\nPopped element is %d", num);
    }
    return num;
}
```

d. Stack Display (Traverse)

Algorithm for display operation:

1. Start
2. If $\text{top} = -1$, stack is empty
3. Else display the Elements of the $\text{stack}[]$ from top to 0.
4. Stop

Program code for display operation

```
void display()
```

```
{
    int i;
    if(top==-1)
        printf("\n Sorry Empty Stack");
    else
    {
        printf("\nThe elements of the stack are\n");
        for(i=top;i>=0;i--)
            printf("stack[%d] = %d\n",i, stack[i]);
    }
} }
```

1.15.1 Stacks using Dynamic Arrays.

- The array is used to implement stack, but the bound (MAX_STACK_ SIZE) should be known during compile time.
- The size of bound is impossible to alter during compilation hence this can be overcome by using dynamically allocated array for the elements and then increasing the size of array as needed.

a. Stack Operations using dynamic array

1. Stack CreateS()::= typedef struct

```
{
    int key; /* other fields */
} element;
element *stack;
MALLOC(stack, sizeof(*stack));
int capacity= 1;
int top= -1;
```

2. Boolean IsEmpty(Stack)::= top < 0;
3. Boolean IsFull(Stack)::= top >= capacity-1;
4. push() :Here the MAX_STACK_SIZE is replaced with capacity

```
void push(element item)
{
    /* add an item to the global stack */
    if (top >= capacity-1)
        stackFull();
    stack[++top] = item;
```

}

5. pop(): In this function, no changes are made.

```

element pop ( ){    /* delete and return the top element from the stack */
    if (top == -1)
        return stackEmpty(); /* returns an error key */
    return stack[top--];
}

```

6. stackFull(): The new code shown below, attempts to increase the capacity of the array stack so that new element can be added into the stack. Before increasing the capacity of an array, decide what the new capacity should be. In array doubling, array capacity is doubled whenever it becomes necessary to increase the capacity of an array.

```

void stackFull()
{
    REALLOC (stack, 2*capacity*sizeof(*stack));
    capacity *= 2;
}

```

1.15.3 Evaluation and conversion of Expressions

- An algebraic expression is a legal combination of operators and operands. Operand is the quantity on which a mathematical operation is performed. Operand may be a variable like x,y, z or a constant like 5, 4, 6 etc.
- Operator is a symbol which signifies a mathematical or logical operation between the operands. Examples of familiar operators include +, -, *, /, ^ etc. An algebraic expression can be represented using three different notations. They are

1. **Infix Expression:** In this expression, the binary operator is placed in-between the operand. Example is shown in Fig 6(a).
2. **Prefix notations (Polish notation):** In this expression, the operator appears before its operand. Example is shown in Fig 6(b).
3. **Postfix (Reverse Polish notation):** In this expression, the operator appears after its operand. Example is shown in Fig 6(c).

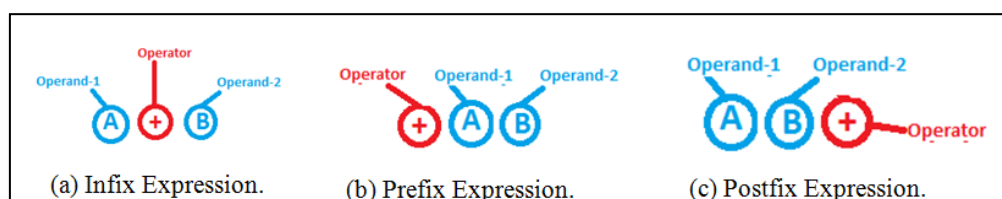


Fig. 6: Different Types of Expression.

4. Precedence and Associativity: In C, there is a precedence hierarchy that determines the order in which operators are evaluated. The table 4.1 contains the precedence hierarchy for C. The operators are arranged from highest precedence to lowest. Operators with highest precedence are evaluated first. The associativity column indicates how to evaluate operators with the same precedence. For example, the multiplicative operators have left-to-right associativity. This means that the expression $a * b / c \% d / e$ is equivalent to $(((a * b) / c) \% d) / e$. Parentheses are used to override precedence, and expressions are always evaluated from the innermost parenthesized expression first.

b. Infix to Postfix Conversion

- To evaluate infix expression we need to concentrate on precedence and associativity. But not for postfix or prefix expression. Hence it is easy and compiler converts infix to postfix or prefix and evaluates the expression. The algorithm given in fig.7 transforms an infix expression into postfix expression.

Table 4.1: Precedence and Associativity of Operators

Operator	Description	Precedence	Associativity
() [] → .	Parenthesis Square bracket Pointer Dot operator	18	L→R
^ \$	Power Dollar	17	R→L
--, ++	Post decrement and post increment	16	L→R
++, -- ! ~ +, - & * sizeof()	Pre decrement and pre increment Logical not One's complement Unary plus and minus Address Pointer Size in bytes	15	R→L
(type)	Type cast	14	R→L
*, /, %	Multiplicative operators	13	L→R
+, -	Binary add and subtract	12	L→R
<<, >>	Right and left shift	11	L→R
<, <=, >, >=	Relational operators	10	L→R
==, !=	Equality operators	9	L→R
&	Bitwise AND	8	L→R
^	Bitwise XOR	7	L→R
	Bitwise OR	6	L→R
&&	Logical AND	5	L→R
	Logical OR	4	L→R
?:	Conditional operator	3	R→L
=, +=, - =, *=,	Assignment operator	2	R→L

/=, %=			
,	Comma operator	1	$L \rightarrow R$

Algorithm:

```

Step 1: Fully Parenthesize the infix expression
Step 2: Push "(" on to the stack
Step 3: Repeat until each character in the infix notation is scanned
    IF a "(" is encountered, push it on the stack
    IF an operand (whether a digit or a character) is encountered, add it to the postfix expression.
    IF a ")" is encountered, then
        a. Repeatedly pop from stack and add it to the postfix expression until a "(" is encountered.
        b. Discard the "(". That is, remove the "(" from stack and do not add it to the postfix expression
    IF an operator 0 is encountered, then
        a. Repeatedly pop from stack and add each operator (popped from the stack) to the postfix expression which has the same precedence or a higher precedence than 0
        b. Push the operator 0 to the stack
    [END OF IF]
Step 4: Repeatedly pop from the stack and add it to the postfix expression until the stack is empty
Step 5: EXIT

```

Fig. 7: Algorithm which Converts Infix to Postfix.

Steps to implement the above algorithm:

1. Scan the infix expression **from left to right**.
2. If the scanned character is an operand, put it in the postfix expression.
3. Otherwise, do the following
 - If the precedence and associativity of the scanned operator are greater than the precedence and associativity of the operator in the stack, then push it in the stack.
 - Check for a condition when the operator at the top of the stack and the scanned operator both are '^'. In this condition, the precedence of the scanned operator is higher due to its right associativity. So it will be pushed into the operator stack.
 - In all the other cases when the top of the operator stack is the same as the scanned operator, then pop the operator from the stack because of left associativity due to which the scanned operator has less precedence.
 - Else, Pop all the operators from the stack which are greater than or equal to in precedence than that of the scanned operator.
4. After doing that, Push the scanned operator to the stack.
5. If the scanned character is a '(', push it to the stack.

5. If the scanned character is a ')', pop the stack and output it until a '(' is encountered, and discard both the parenthesis.
6. Repeat steps 2-5 until the infix expression is scanned.
7. Once the scanning is over, Pop the stack and add the operators in the postfix expression until it is not empty.
8. Finally, print the postfix expression.

Example:

Consider the infix expression **exp = a+b*c+d**

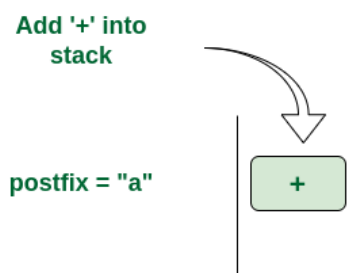
and the infix expression is scanned using the iterator **i**, which is initialized as **i = 0**.

1st Step: Here $i = 0$ and $\text{exp}[i] = 'a'$ i.e., an operand. So add this in the postfix expression. Therefore, **postfix = "a"**.



'a' is an operand. Add it in postfix expression

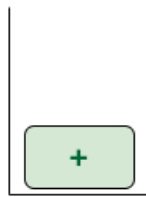
2nd Step: Here $i = 1$ and $\text{exp}[i] = '+'$ i.e., an operator. Push this into the stack. **postfix = "a"** and **stack = {+}**.



Stack is empty. Push '+' into stack

Push '+' in the stack

3rd Step: Now $i = 2$ and $\text{exp}[i] = 'b'$ i.e., an operand. So add this in the postfix expression. **postfix = "ab"** and **stack = {+}**.



postfix = "ab"

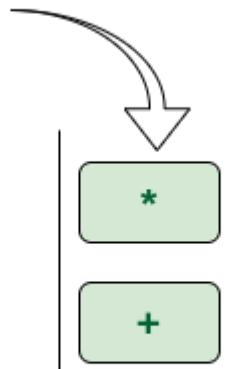
'b' is an operand. Add it in postfix expression

Add 'b' in the postfix

4th Step: Now $i = 3$ and $\text{exp}[i] = '*'$ i.e., an operator. Push this into the stack. **postfix = "ab"** and **stack = {+, *}**.

Add '*' into stack

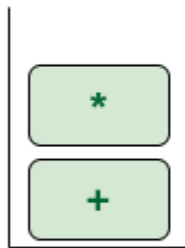
postfix = "ab"



*** has higher precedence. Push it into stack**

Push '*' in the stack

5th Step: Now $i = 4$ and $\text{exp}[i] = 'c'$ i.e., an operand. Add this in the postfix expression. **postfix = "abc"** and **stack = {+, *}**.

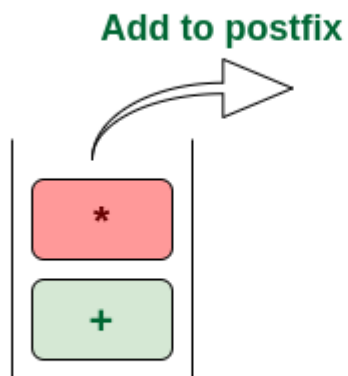


postfix = "abc"

'c' is an operand. Add it in postfix expression

Add 'c' in the postfix

6th Step: Now $i = 5$ and $\text{exp}[i] = '+'$ i.e., an operator. The topmost element of the stack has higher precedence. So pop until the stack becomes empty or the top element has less precedence. '*' is popped and added in postfix. So **postfix = "abc*"** and **stack = {+}**.

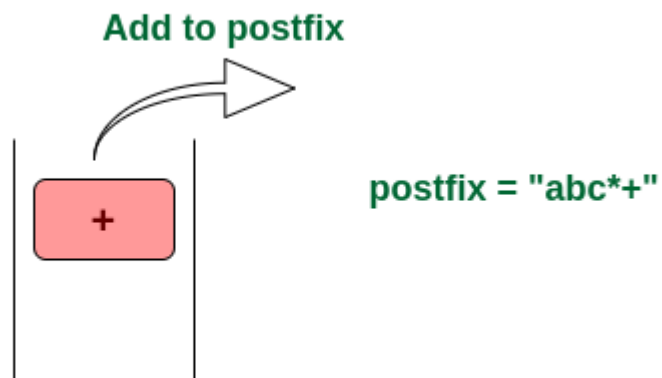


postfix = "abc*"

stack top has higher precedence than +

Pop '*' and add in postfix

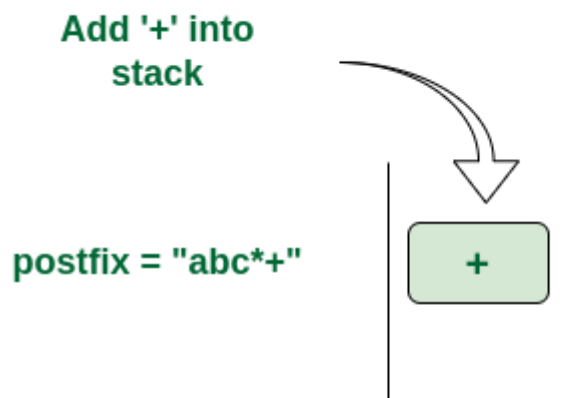
Now top element is '+' that also doesn't have less precedence. Pop it. **postfix = "abc*+"**.



'+' and stack top has same precedence

Pop '+' and add it in postfix

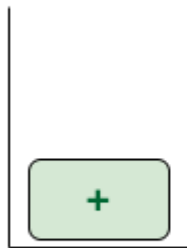
Now stack is empty. So push '+' in the stack. **stack** = {+}.



Stack is empty. Push '+' into stack

Push '+' in the stack

7th Step: Now $i = 6$ and $\text{exp}[i] = 'd'$ i.e., an operand. Add this in the postfix expression.
postfix = "abc*+d".

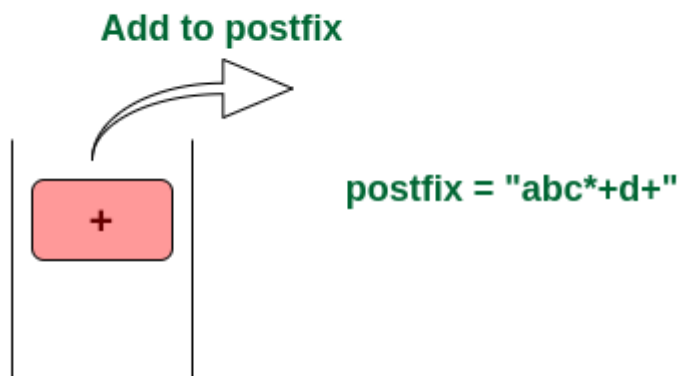


postfix = "abc*+d"

'd' is an operand. Add it in postfix expression

Add 'd' in the postfix

Final Step: Now no element is left. So empty the stack and add it in the postfix expression. postfix = "abc*+d+".



Nothing left. So pop all operators

Pop '+' and add it in postfix

Problem.1: Convert the following infix expression into postfix expression:

$$A - (B / C + (D \% E * F) / G) * H$$

Solution:given in Table 1.(A – (B / C + (D % E * F) / G)* H)

Table 1: Solution for Problem 1

Infix CharacterScanned	Stack	Postfix Expression
------------------------	-------	--------------------

((
A	(A
-	(-	A
((- (A
B	(- (AB
/	(- (/	AB
C	(- (/	ABC
+	(- (+	ABC/
((- (+ (ABC/
D	(- (+ (ABC/D
%	(- (+ (%	ABC/D
E	(- (+ (%	ABC/DE
*	(- (+ (*	ABC/DE%
F	(- (+ (*	ABC/DE%F
)	(- (+	ABC/DE%F*
/	(- (+ /	ABC/DE%F*
G	(- (+ /	ABC/DE%F*G
)	(-	ABC/DE%F*G/+
*	(- *	ABC/DE%F*G/+
H	(- *	ABC/DE%F*G/+H
)		ABC/DE%F*G/+H*-

Problem 2: Convert the following infix expression into postfix expression:

$$(((A+(B-C)*D)^E+F)$$

Solution:given in Table 2.

Table 2: Solution for Problem 2

Infix Character Scanned	Stack	Postfix Expression
((
(((
((((
A	(((A
+	(((+	A
((((+ (A
B	(((+ (AB
-	(((+ (-	AB
C	(((+ (-	ABC
)	(((+	ABC-
*	(((+ *	ABC-
D	(((+ *	ABC-D
)	(((ABC-D*+
^	(((^	ABC-D*+
E	(((^	ABC-D*+E
)	(((ABC-D*+E^
+	((+	ABC-D*+E^
F	((+	ABC-D*+E^F
)		ABC-D*+E^F+

Example: Write a program for Converting Infix to Postfix Expression.

```
#include<stdio.h>

void infix_to_postfix();
void push(char);
char pop();
int priority(char);
char infix[30], postfix[30],stack[30];
int top=-1;

void main()
{
    printf("Enter the valid Infix expression \n");
    scanf("%s",infix);
    infix_to_postfix();
    printf("\n Infix expression : %s",infix);
    printf("\n Postfix expression : %s\n",postfix);
}

// push symbol to stack
void push(char item)
{
    stack[++top]=item;
} // end of function push

// pop symbol from stack
char pop()
{
    return stack[top--];
} // end of function pop

// check the priority of operator
int priority(char symb)
{
    int p;
    switch(symb)
    {
```

```
        case '+':
        case '-':p=1;
break;
        case '*':
        case '/':
        case '%':    p=2;
break;
        case '^':
        case '$':    p=3;
break;
        case '(':
        case ')':p=0;
break;
        case '#':    p=-1;
break;
    } // end of switch
    return p;
} // end of function priority
```

//converting an Infix Expression to Postfix Expression

```
void infix_to_postfix()
```

```
{
    int i=0,j=0;
    char symb,temp;
    push('#');
    for(i=0;infix[i]!='\0';i++)
    {
        symb=infix[i];
        switch(symb)
        {
            case '(': push(symb); // push all symbols inside the ( to top of stack
            break;
            case ')': temp=pop(); //pop symbol from top of stack
            while(temp!='(') //pop all symbols from top of stack and store in postfix until (
            {
```

```
        postfix[j++]=temp;
        temp=pop();
    } // end of while
    break;

    case '+':
    case '-':
    case '*':
    case '/':
    case '%':
    case '^':

    case '$': while(priority(stack[top])>=priority(symb)) // check for priority of
operator
    {
        temp=pop();
        postfix[j++]=temp;
    }

        push(symb);
        break;
    default: postfix[j++]=symb;
} // end of switch
} // end of for
while(top>0) // pop remaining all symbols form top of stack and store to
postfix
{
    temp=pop();
    postfix[j++]=temp;
} // end of while
    postfix[j]='\0'; // end string postfix
} // end of function infix_to_postfix
```


c. Evaluation of Postfix Expression

- The ease of evaluation acts as the driving force for computers to translate an infix notation into a postfix notation. That is, given an algebraic expression written in infix notation, the computer first converts the expression into the equivalent postfix notation and then evaluates the postfix expression.
- Both these tasks make extensive use of stacks as the primary tool. Using stacks, any postfix expression can be evaluated very easily. Every character of the postfix expression is scanned from left to right.
- If the character encountered is an operand, it is pushed on to the stack. However, if an operator is encountered, then the top two values are popped from the stack and the operator is applied on these values.
- The result is then pushed on to the stack. Let us look at Fig.8 which shows the algorithm to evaluate a postfix expression.

Step 1: Scan the symbol from left to right
 Step 2: If the scanned symbol is an operand, Push it on to the stack.
 Step 3: If the scanned symbol is an operator, pop two elements from the top of stack. The first popped element is op2 and second popped element is op1.
 Step 4: Then perform the indicated operation: $res = op1 \text{ op } op2$
 Step 5: Push the result on to stack
 Step 6: Repeat the procedure until the end of input.

Fig. 8: Algorithm for Evaluation of Postfix Expression.

Problem 1: Evaluate the following postfix expression FOR A=1, B=2 and C=3:

ABC+*CBA-+*

Solution: given in Table 1. The above expression can be written as 123+*321-+*

Table 1: Solution for Problem 1.

Postfix Expression	Symbol Scanned	Op2	Op1	Res = Op1 Op Op2	Stack
123+*321-+*	1				1
23+*321-+*	2				1,2
3+*321-+*	3				1,2,3
+*321-+*	+	3	2	2+3=5	1,5
321-+	*	5	1	1*5=5	5
321-+*	3				5,3
21-+*	2				5,3,2
1-+*	1				5,3,2,1
-+*	-	1	2	2-1=1	5,3,1
+*	+	1	3	3+1=4	5,4

*	*	4	5	5*4=20	<u>20</u>
---	---	---	---	--------	-----------

Problem 2: Evaluate the following postfix expression FOR A=1, B=2 and C=3:

$$AB+C-BA+C^--$$

Solution:given in Table 2.The above expression can be written as 12+3-21+3^--

Table 2: Solution for Problem 2.

Postfix Expression	Symbol Scanned	Op2	Op1	Res = Op1 Op Op2	Stack
12+3-21+3^--	1				1
2+3-21+3^--	2				1,2
+3-21+3^--	+	2	1	1+2=3	3
3-21+3^--	3				3,3
-21+3^--	-	3	3	3-3=0	0
21+3^--	2				0,2
1+3^--	1				0,2,1
+3^--	+	1	2	2+1=3	0,3
3^--	3				0,3,3
^--	^	3	3	3^3=27	0,27
-	-	27	0	0-27=-27	<u>-27</u>

Problem 3: Convert the following infix expression to postfix expression and evaluate the postfix expression FOR A=6, B=3, C=1,D=2, E=4:

$$A/B-C+D*E-A*C$$

Solution:Fully parenthesized expression is: (((A/B)-C)+(D*E))-(A*C)). Refer table 3 for conversion of infix to postfix expression. We get AB/C-DE*+AC*- as postfix expression. Now to evaluate this postfix expression, refer table 4.10. When we assign the given values to the postfix expression, we get: 63/1-24*+61*-.

Table 3: Solution for Problem 3.

Postfix Expression	Symbol Scanned	Op2	Op1	Res = Op1 Op Op2	Stack
63/1-24*+61*-	6				6
3/1-24*+61*-	3				6,3
/1-24*+61*-	/	3	6	6/3=2	2
1-24*+61*-	1				2,1
-24*+61*-	-	1	2	2-1=1	1
24*+61*-	2				1,2
4*+61*-	4				1,2,4
+61-	*	4	2	2*4=8	1,8
+61*-	+	8	1	1+8=9	9
61*-	6				9,6
1*-	1				9,6,1
*-	*	1	6	6*1=6	9,6
-	-	6	9	9-6=3	<u>3</u>

Example program to evaluate that postfix expression.

/* .Design, Develop and Implement a Program in C for the following Stack Applications

To Evaluate the postfix expression with single digit operands and operators: +, -, *, /, %, ^*/

```
#include<stdio.h>
#include<math.h>
void push(float);
float pop();
void evaluate(char[]);
float stack[20];
int top=-1;
void main()
{
    int choice,n;
    char postfix[100];
    while(1) // infinite loop for menu
    {
        printf("\n STACK APPLICATIONS");
        printf("\n Enter your Choice: ");
        printf("\n 1. Evaluation of postfix expression with single digit operands and operators");
        printf("\n 2. Exit \n");
        scanf("%d", &choice);
        switch(choice)
        {
            case 1 : printf("Enter a valid postfix expression\n");
                    scanf("%s",postfix);
                    evaluate(postfix);
                    break;
            case 2 : return;
            default : printf("\n Invalid Choice");
        } // end of switch
    } // end of menu
} // end of main

// push item to stack
void push(float item)
```

```

{
    stack[++top]=item;
} // end of push

// pop item from stack
float pop()
{
    return stack[top--];
} // end of pop

// function to postfix expression with single digit operands and operators: +, -, *, /, %, ^
void evaluate(char postfix[100])
{
    int i;
    float op1, op2, res;
    char symb;
    for(i=0;postfix[i]!='\0';i++) // repeate until end of string
    {
        symb=postfix[i];
        if(isdigit(symb)) // check for digit or not
            push(symb-'0'); // if digit push to top of stack -'0' is for ascii to number
conversion
        switch(symb)
        {
            case '+':op2=pop();
                op1=pop();
                res=op1+op2;
                push(res);
                break;
            case '-':op2=pop();
                op1=pop();
                res=op1-op2;
                push(res);
                break;

```

```
case '*':op2=pop();
    op1=pop();
    res=op1*op2;
    push(res);
    break;
case '/':op2=pop();
    op1=pop();
    if(op2==0)
    {
        printf("Division by zero Error\n");
        return;
    }
    res=op1/op2;
    push(res);
    break;
case '%': op2=pop();
    op1=pop();
    if(op2==0)
    {
        printf("Division by zero Error\n");
        return;
    }
    res=(int)op1%(int)op2;
    push(res);
    break;
case '^':op2=pop();
    op1=pop();
    res=pow(op1,op2);
    push(res);
    break;
        } // end of switch
    } // end of for
res=pop(); // pop the final answer from top of stack
if(top==-1)
```

```
    printf("\n Result: %f\n ",res); // Display the final answer
else
{
    printf("\nINVALID POSTFIX EXPRESSION\n");
    top=-1;
}
} // end of evaluate function
```

Question Bank

1. Define data structures
2. Explain classifications of data structures
3. Explain the different operations of data structures
4. Define structures. Explain different types of structure declaration and initialization
5. Write the difference between structures and union
6. Explain pointers with an example
7. What are pointer variables? How to declare a pointer variable?
8. Define a pointer. Write a C function to swap two numbers using pointers.
9. Explain self-referential structures
10. Define array and explain with an example
11. What are the different operations of array? Explain
12. Explain multi-dimensional array
13. Explain sparse matrix and its representation
14. Explain with example the functions supported by C to carryout dynamic memory allocation
15. What are the various memory allocation techniques? Explain how memory can be dynamically allocated using malloc().
16. Write the difference between malloc and calloc
17. Explain how the 2-D array can be created dynamically
18. Develop a structure to represent a solar system. Each planet has 3 fields - name, distance from sun, no of moons. Write a program to read the data for each planet and store. Also print the name of the planets that has the highest number of moons.
19. What is a polynomial? What is the degree of a polynomial? Write a function to add two polynomials.
20. Define structure and unions with example.

21. Write a C program with suitable structure definition and variable declaration to store information about employee. Consider the following fields: Ename, EID, DOJ(Date, month, year), and salary (basic, DA, HRA).
22. Write a C program for demonstrating all array operations.
23. For the given sparse matrix and its transpose, give the triplet representation using one dimensional array.

15	0	0	22	0	-15
0	11	3	0	0	0
0	0	0	-6	0	0
0	0	0	0	0	0
91	0	0	0	0	0
0	0	28	0	0	0

24. Consider two polynomials $A(x)=2x^{1000}+1$ and $B(x)=x^4+10x^3+3x^2+1$ with a diagram show how these two polynomials are stored using 1D array and also give its C representation.
25. Define stack. Give the C implementation of push, pop, overflow and underflow functions for stack using arrays.
26. Define stack. List the operation on stack.
27. Obtain postfix expression for $((A+(B-C)*D)^E)+F$.
28. Write the postfix form of the following expression.
 - i. $(a+b)*d+e/(f+a*d)+c$
 - ii. $((a/(b-c+d))*(e-a)*c)$
 - iii. $a/b-c+d*e-a*c$
29. Write an algorithm to convert infix to postfix expression and apply the same to convert the following expression from infix to postfix :
 - i) $(a * b) + c/d$
 - ii) $((a/b)-c) + (d * e) - (a * c)$.
30. Convert the infix expression $((a/b)-c) + (d * e) - (a * c)$ into postfix expression. Write a function to evaluate that postfix expression and trace for the data: $a=6, b=3, c=1, d=2, e=4$
31. Write an algorithm to convert infix to postfix expression and trace it for the expression $a * (b + c)*d$