

Module – 1**10 Hours**

Introduction: Data Structures, Classifications (Primitive & Non Primitive), Data structure Operations, Review of Arrays, Structures, Self-Referential Structures, and Unions. Pointers and Dynamic Memory Allocation Functions. Representation of Linear Arrays in Memory, Dynamically allocated arrays, Array Operations: Traversing, inserting, deleting, searching, and sorting. Multidimensional Arrays, Polynomials and Sparse Matrices. Strings: Basic Terminology, Storing, Operations and Pattern Matching algorithms. Programming Examples.

Data Structure

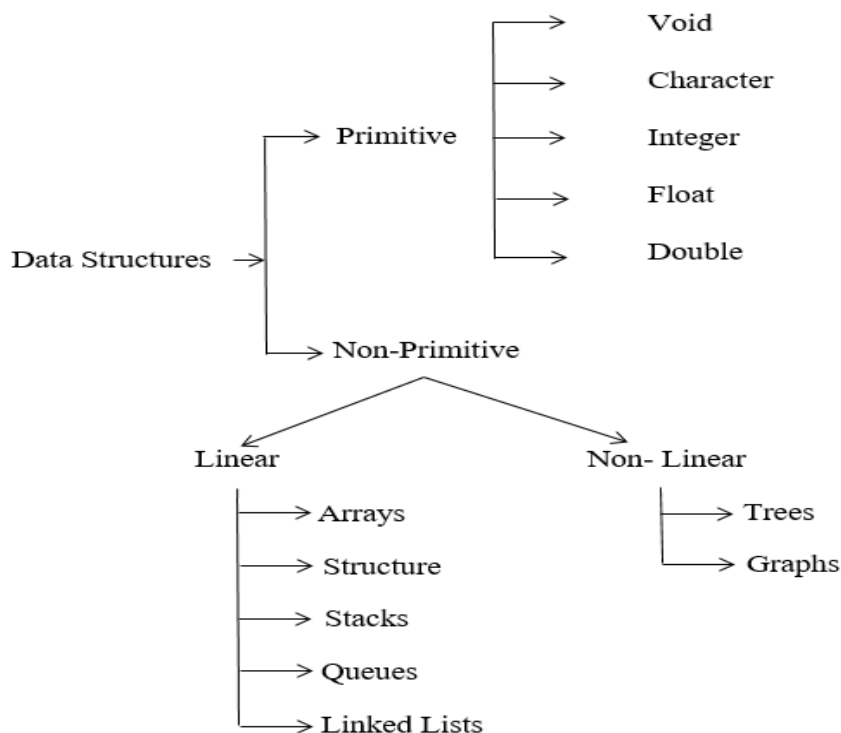
Data structure is a representation of the logical relationships existing between individual elements of data. A data structure is a way of organizing all data items that considers not only the elements stored but also their relationship to each other.

The logical or mathematical model of a particular organization of data is called a data structure.

Classification of Data Structures

Data Structures can be divided into two categories,

- i) Primitive Data Structures
- ii) Non-Primitive Data Structures



Primitive Data Structures

These are basic data structures and are directly operated upon by the machine instructions.

Example: Integers, Floating Point Numbers, Characters and Pointers etc.

Non-Primitive Data Structures

These are derived from the primitive data structures. The non-primitive data structures emphasize on structuring of a group of homogeneous or heterogeneous data items.

Example: Arrays, Lists and Files etc.

Operations on Data Structures

The commonly used operations on data structures are as follows,

- ❖ **Create:** The Create operation results in reserving memory for the program elements. The creation of data structures may take place either during compile time or during run time.
- ❖ **Destroy:** The Destroy operation destroys the memory space allocated for the specified data structure.
- ❖ **Selection:** The Selection operation deals with accessing a particular data within a data structure.
- ❖ **Updating:** The Update operation updates or modifies the data in the data structure.
- ❖ **Searching:** The Searching operation finds the presence of the desired data item in the list of data items.
- ❖ **Sorting:** Sorting is the process of arranging all the data items in the data structure in a particular order, say for example, either in ascending order or in descending order.
- ❖ **Merging:** Merging is a process of combining the data items of two different sorted list into a single list.

Arrays

An array is collection of data items of same data type, which are in consecutive memory locations. Each value (data item) in an array is indicated by same name that is array name and an index which indicates the position of value in an array.

Types of Arrays

- i) One dimensional array
- ii) Two dimensional array
- iii) Multi-dimensional array

One Dimensional Array

The one dimensional array or single dimensional array is the simplest type of array that contains only one row for storing the values of same type.

Declaration of One-Dimensional Array

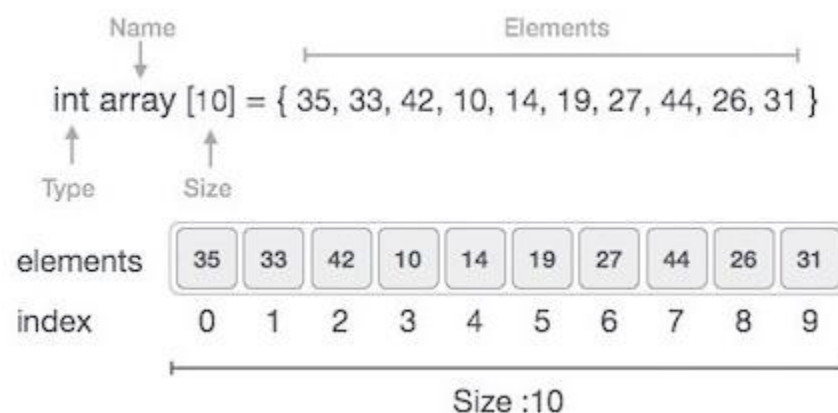
Syntax:

```
data _type array_name [array size];
```

Where,

- data type defines the type of an array or type of the data stores in it.
- array name is the name given to represent the array.
- array size tells number of values that can be stores in an array.

Example: `int a[25];`



Memory Occupied by One-Dimensional Array

$$\text{Total memory} = \text{array size} * \text{size of (data type)}$$

In the above example, int a[25];

Array size=25

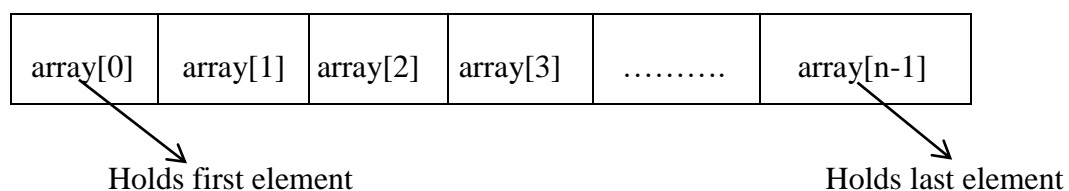
Data type is int so sizeof (int) =2 bytes (in 16 bit machine)

So, total memory=25*2 =50 bytes

Structure of one-dimensional array:

Index for an array for n elements starts from zero and ends with n-1

Consider int array[n];



Representation of Linear Arrays in Memory

The elements of linear array are stored in consecutive memory locations. The computer does not keep track of address of each element of array. It only keeps track of the base address of the array and on the basis of this base address the address or location of any element can be found. We can find out the location of any element by using following formula:

$$\text{LOC (LA [K])} = \text{Base (LA)} + w (K - \text{LB})$$

Here,

LA is the linear array.

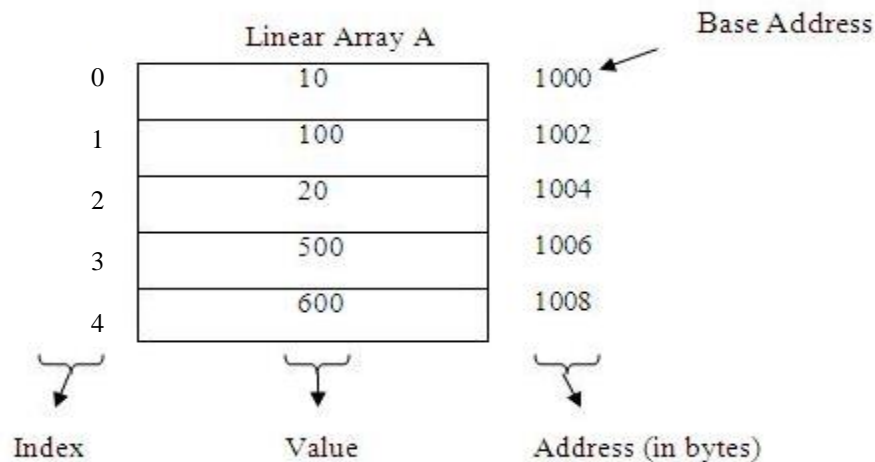
LOC(LA [K]) is the location of the Kth element of LA.

Base (LA) is the base address of LA.

w is the number of bytes taken by one element.

K is the Kth element.

LB is the lower bound.



Suppose we want to find out LOC (A [3]). Here,

Base (A) = 1000

w = 2 bytes (Because an integer takes two bytes in the memory).

K = 3

LB = 0

After putting these values in the given formula, we get:

$$\text{LOC (A [3])} = 1000 + 2 (3 - 0)$$

$$= 1000 + 2 (3)$$

$$= 1000 + 6$$

$$= 1006$$

Initialization one-dimensional array:

- Array can be initialized statically one by one and in a single statement or dynamically using loop statements.
- Array elements can be initialized at the time of declaration.

Syntax:

`data_type array_name[array_size]={ v1,v2,v3,.....vn};`

Here,

- data_type is the type of data to be stored in an array, data type can be char, int, float, double or string.
- array_name is the name given to the array.

- array_size specifies the number of values given to the array that is array size specifies size of array.
- v1, v2, v3, v4....., vn are the values given to the array.
- Number of values should not exceed the size of the array.

Various Method of Initializing One-Dimensional Array

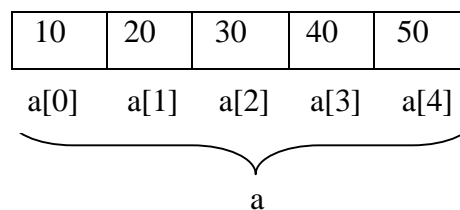
- Initializing all specified memory location
- Partial array initialization
- Initialization without size
- String initialization
- Dynamic initialization with for or while loop

i) Initializing all Specified Memory Location

Consider the example,

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

Here , five continuous memory locations are reserved for array 'a' and all five locations are initialized as shown below,

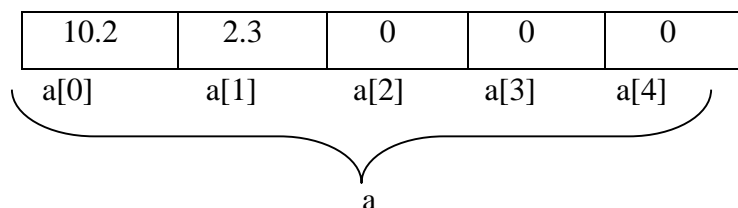


ii) Partial Array Initialization

Here the number of elements to be stored in an array will be less than the size specified

```
Example: float a[5]={ 10.2,2.3};
```

First two locations of the array 'a' will be initialized and left out spaces will be filled with zeros



iii) Initialization without Size

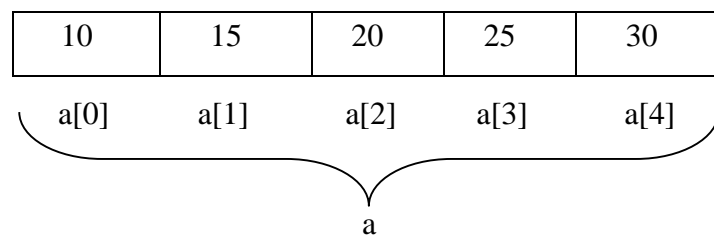
Here array size is not specified,

Example:

```
int a[]={10,15,20,25,30};
```

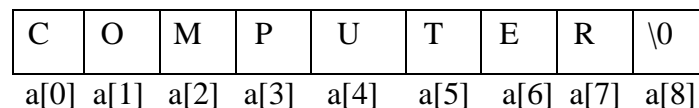
Here though we have not specified the size of the array it will be set to total number of elements to be stored.

- The compiler will calculate size of the array using number of elements specified
- For above example $5 \times 2 = 10$ bytes will be reserved and array will be initialized as follows,

**iv) Array Initialization with String**

Sequence of characters entered enclosed within double quotes is called as string always ends with a NULL character (\0)

Example: a) `char a[]="COMPUTER";`

**v) Dynamic Initialization with for or while Loop**

Consider, `int arr[n];` //Array declaration

/* Initialization using for loop */

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

```
{
```

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

```
}
```

/*Initialization using while loop*/

```
i=0;
```

```
while(i<n)
```

```
{
```

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

```
i++;
```

```
}
```

```
/*Reading and displaying elements to and from an one dimensiona array*/
#include<stdio.h>
#include<conio.h>
void main()
{
int array[25],n,i;
printf("Enter number of elements\n");
scanf("%d",&n);
printf("Enter %d elements\n",n);

for(i=0;i<n;i++)
{
scanf("%d",&array[i]);
}
printf("Array elements are \n");
for(i=0;i<n;i++)
{
printf("%d\t",arra[i]);
}
getch();
}
```

Output:

Enter number of elements

5

Enter 5 elements

11 12 13 14 15

Array elements are

11 12 13 14 15

Basic Operations

Following are the basic operations supported by an array,

- **Traverse** – print all the array elements one by one.
- **Insertion** – Adds an element at the given index.
- **Deletion** – Deletes an element at the given index.
- **Search** – Searches an element using the given index or by the value.
- **Update** – Updates an element at the given index.
- **Sorting** – Arrange the elements in ascending or descending order

Algorithms for Traversing an Array

Algorithm 1: (Traversing a Linear Array) Here LA is a linear array with lower bound LB and upper bound UB. This algorithm traverses LA applying an operation PROCESS to each element of LA.

1. Set $K := LB$.
2. Repeat Steps 3 and 4 while $K \leq UB$
3. [Visit element.] Apply PROCESS to $LA[K]$.
4. [Increase counter.] Set $K := K + 1$.
 [End of Step 2 loop.]
5. Exit.

Algorithm 2: (Traversing a Linear Array) This algorithm traverses a linear array LA with lower bound LB and upper bound UB.

1. Repeat for $K = LB$ to UB
 Apply PROCESS to $LA[K]$.
 [End of loop.]
2. Exit.

Algorithm for Inserting and Deleting an Element from an Array

Algorithm 3: (Inserting into a Linear Array) INSERT(LA, N, K, ITEM)

Here LA is a linear array with N elements and K is a positive integer such that $K \leq N$. This algorithm inserts an ITEM into the K^{th} position in LA.

1. [Initialize counter.] Set $J := N$
2. Repeat Steps 3 and 4 while $J \geq K$
3. [Move J^{th} element downward.] Set $LA[J+1] := LA[J]$.
4. [Decrease counter.] Set $J := J - 1$.
 [End of Step 2 loop.]
5. [Insert element.] Set $LA[K] := \text{ITEM}$.
6. [Reset N.] Set $N := N + 1$.
7. Exit.

Algorithm 4: (Deleting from a Linear Array) DELETE(LA, N, K, ITEM)

Here LA is a linear array with N elements and K is a positive integer such that $K \leq N$. This algorithm deletes the K^{th} element from LA.

1. [Initialize counter.] Set $\text{ITEM} := \text{LA}[K]$.
2. Repeat for $J = K$ to $N - 1$:
 - [Move (J+1) element upward.] Set $\text{LA}[J] := \text{LA}[J + 1]$.
3. [Decrease counter.] Set $J := J - 1$.
[End of Step 2 loop.]
4. [Reset the number N of elements in LA.] Set $N := N - 1$.
5. Exit.

Algorithm for Sorting the Elements of an Array**Algorithm 5:** (Bubble Sort) BUBBLE(DATA, N)

Here DATA is an array with N elements. This algorithm sorts the elements in DATA.

1. Repeat Steps 2 and 3 $K = 1$ to $N - 1$
2. Set $\text{PTR} := 1$. [Initialize pass pointer PTR.]
3. Repeat while $\text{PTR} \leq N - K$: [Execute pass.]
 - a. If $\text{DATA}[\text{PTR}] > \text{DATA}[\text{PTR} + 1]$, then:
 - Interchange $\text{DATA}[\text{PTR}]$ and $\text{DATA}[\text{PTR} + 1]$.
 - [End of If structure.]
 - b. Set $\text{PTR} := \text{PTR} + 1$.
[End of inner loop.]
- [End of Step 1 outer loop.]
4. Exit.

Algorithms for Searching an Element of an Array**Algorithm 6:** (Linear Search) LINEAR(DATA, N, ITEM, LOC)

Here DATA is a linear array with N elements, and ITEM is a given item of information. This algorithm finds the location LOC of ITEM in DATA, or sets $\text{LOC} := 0$ if the search is unsuccessful.

1. [Insert ITEM at the end of DATA.] Set $DATA[N+1] := ITEM$.
2. [Initialize counter.] Set $LOC := 1$.
3. [Search for ITEM.]
 Repeat while $DATA[LOC] \neq ITEM$:
 Set $LOC := LOC + 1$.
 [End of loop.]
4. [Successful?] If $LOC = N+1$, then: Set $LOC := 0$.
5. Exit.

Algorithm 7: (Binary Search) $BINARY(DATA, LB, UB, ITEM, LOC)$

Here DATA is a linear array with lower bound LB and upper bound UB, and ITEM is a given item of information. The variables BEG, END and MID denote, respectively, the beginning, end and middle locations of a segment of elements of DATA. This algorithm finds the location LOC of ITEM in DATA or sets $LOC = NULL$.

1. [Initialize segment variables.]
 Set $BEG := LB$, $END := UB$ and $MID = \text{INT}((BEG + END) / 2)$.
2. Repeat Steps 3 and 4 while $BEG \leq END$ and $DATA[MID] \neq ITEM$
3. If $ITEM < DATA[MID]$, then
 Set $END := MID - 1$.
 Else:
 Set $BEG := MID + 1$.
 [End of If structure.]
4. Set $MID := \text{INT}((BEG + END) / 2)$.
 [End of Step 2 loop.]
5. If $DATA[MID] = ITEM$, then:
 Set $LOC := MID$.
 Else:
 Set $LOC := NULL$.
 [End of If structure.]
6. Exit.

Insertion Operation

Insert operation is to insert one or more data elements into an array. Based on the requirement, a new element can be added at the beginning, end, or any given index of array.

Example: Let **la** be a Linear Array (unordered) with **n** elements and **k** is a positive integer such that **k** ≤ **n**. Following is the algorithm where item is inserted into the **kth** position of **la**.

```
#include <stdio.h>

main()

{
    int la[25];
    int item, k, n;
    int i, j = n;
    printf("Enter the number of array elements\n");
    scanf("%d",&n);
    printf("Enter the array elements\n");
    for(i=0;i<n;i++)
    {
        scanf("%d",&la[i]);
    }
    printf("The original array elements are :\n");
    for(i = 0; i<n; i++)
    {
        printf("la[%d] = %d \n", i, la[i]);
    }
    printf("Enter the position and item to be inserted\n");
    scanf("%d%d",&k,&item);
    n = n + 1;
    while( j >= k)
    {
        la[j+1] = la[j];
```

Output:

Enter the number of array elements

5

Enter the array elements

1 3 5 7 8

The original array elements are :

la[0] = 1

la[1] = 3

la[2] = 5

la[3] = 7

la[4] = 8

Enter the position and item to be inserted

3

10

The array elements after insertion :

la[0] = 1

la[1] = 3

la[2] = 5

la[3] = 10

la[4] = 7

la[5] = 8

```

    j = j - 1;
}
la[k] = item;
printf("The array elements after insertion :\n");
for(i = 0; i < n; i++)
{
    printf("la[%d] = %d \n", i, la[i]);
}
}

```

Deletion Operation

Deletion refers to removing an existing element from the array and re-organizing all elements of an array.

Example: Consider **la** is a linear array with **n** elements and **k** is a positive integer such that **k ≤ n**. Following is the algorithm to delete an element available at the **kth** position of **la**.

```

#include<stdio.h>
main( )
{
    int la[25];
    int k, n;
    int i, j;
    printf("Enter the number of array elements\n");
    scanf("%d",&n);
    printf("Enter the array elements\n");
    for(i=0;i<n;i++)
    {
        scanf("%d",&la[i]);
    }
    printf("The original array elements are :\n");
    for(i = 0; i < n; i++)

```

```

{
    printf("la[%d] = %d \n", i, la[i]);
}
printf("Enter the position of element to be deleted\n");
scanf("%d",&k);
j = k;
while( j < n)
{
    la[j-1] = la[j];
    j = j + 1;
}
n = n - 1;
printf("The array elements after deletion:\n");
for(i = 0; i<n; i++)
{
    printf("la[%d] = %d \n", i, la[i]);
}
}

```

Output:

Enter the number of array elements

5

Enter the array elements

1 3 5 7 8

The original array elements are :

la[0] = 1

la[1] = 3

la[2] = 5

la[3] = 7

la[4] = 8

Enter the position of element to be deleted

3

The array elements after deletion :

la[0] = 1

la[1] = 3

la[2] = 7

la[3] = 8

Search Operation

Search's for an array element based on its value or its index.

Example: Consider **la** is a linear array with **n** elements and **k** is a positive integer such that **k<=n**.

Following is the algorithm to find an element with a value of item using sequential search.

```

#include <stdio.h>
main( )
{
    int la[25];
    int item, n;
    int i, j = 0, flag=0;
    printf("Enter the number of array elements\n");
    scanf("%d",&n);

```

```

printf("Enter the array elements\n");
for(i=0;i<n;i++)
{
    scanf("%d",&la[i]);
}
printf("The original array elements are :\n");
for(i = 0; i<n; i++)
{
    printf("la[%d] = %d \n", i, la[i]);
}
printf("Enter the element to be searched\n");
scanf("%d",&item);
while( j < n)
{
    if( la[j] == item )
    {
        Flag=1;
        break;
    }
    j = j + 1;
}
if(flag)
printf("Found element %d at position %d\n", item, j+1);
else
printf("Element not found");
}

```

Output:

Enter the number of array elements

5

Enter the array elements

1

3

5

7

8

The original array elements are :

la[0] = 1

la[1] = 3

la[2] = 5

la[3] = 7

la[4] = 8

Enter the element to be searched

5

Found element 5 at position 3

Update Operation

Update operation refers to updating an existing element from the array at a given index.

Example: Consider **la** is a linear array with **n** elements and **k** is a positive integer such that **k** ≤ **n**.

Following is the algorithm to update an element available at the **kth** position of **la**.

```

#include<stdio.h>

main( )
{
    int la[25];
    int k, n, item;
    int i, j;
    printf("Enter the number of array elements\n");
    scanf("%d",&n);
    printf("Enter the array elements\n");
    for(i=0;i<n;i++)
    {
        scanf("%d",&la[i]);
    }
    printf("The original array elements are :\n");
    for(i = 0; i<n; i++)
    {
        printf("la[%d] = %d \n", i, la[i]);
    }
    printf("Enter the position and new value of element to be updated\n");
    scanf("%d%d",&k,&item);
    la[k-1] = item;
    printf("The array elements after updation:\n");
    for(i = 0; i<n; i++)
    {
        printf("la[%d] = %d \n", i, la[i]);
    }
}

```

Output:

Enter the number of array elements

5

Enter the array elements

1 3 5 7 8

The original array elements are :

la[0] = 1

la[1] = 3

la[2] = 5

la[3] = 7

la[4] = 8

Enter the position and new value of element to be updated

3

9

The array elements after updation:

la[0] = 1

la[1] = 3

la[2] = 9

la[3] = 7

la[4] = 8

Sorting

Ordering of elements of an array in either ascending or descending sequence is termed as sorting.

Example: Program to print the elements of array in ascending and descending order.

```
#include <stdio.h>

int main( )
{
    int a[10], i=0, j=0, n, t;
    printf ("\n Enter the no. of elements: ");
    scanf ("%d", &n);
    printf ("\n");
    for (i = 0; i <n; i++)
    {
        printf ("\n Enter the %dth element: ", (i+1));
        scanf ("%d", &a[i]);
    }
    for (j=0 ; j<(n-1) ; j++)
    {
        for (i=0 ; i<(n-1) ; i++)
        {
            if (a[i+1] < a[i])
            {
                t = a[i];
                a[i] = a[i + 1];
                a[i + 1] = t;
            }
        }
    }
    printf ("\n Ascending order: ");
    for (i=0 ; i<n ; i++)
    {
        printf (" %d", a[i]);
```

Output:

Enter the no. of elements: 5

Enter the 1th element: 25

Enter the 2th element: 50

Enter the 3th element: 75

Enter the 4th element: 35

Enter the 5th element: 100

Ascending order: 25 35 50 75 100

Descending order: 100 75 50 35 25

```

    }
    printf ("\n Descending order: ");
    for (i=n ; i>0 ; i--)
    {
        printf (" %d", a[i-1]);
    }
    /* indicate successful completion */
    return 0;
}

```

Example: Program for generating Fibonacci series using array

/*Program for generating Fibonacci series using array*/

```

#include<stdio.h>
#include<conio.h>
void main()
{
    int fib[20],n,i;
    printf("Enter Number of terms to be generated\n");
    scanf("%d",&n);
    fib[0] = 0;
    fib[1] = 1;
    printf("Fibonacci Series\n");
    for(i=2 ; i<n ; i++)
    {
        fib[i] = fib[i-1] + fib[i-2];
    }
    for(i=0 ; i<n ; i++)
    {
        printf("%d\t",fib[i]);
    }
    getch( ); }

```

Output:

```

Enter the number of terms to be generated
7
Fibonacci Series
0 1 1 2 3 5 8

```

Two Dimensional Array

- An array where elements can be stored row-wise and column-wise is called a two dimensional array.
- A two dimensional array is used to store a table of values of same data type.

Declaration of Two Dimensional Array

Syntax :

```
data_type array_name[row_size][column_size];
```

Where,

- data_type indicates type of an array or type of data stored in array.
- array_name is the name given to represent the array.
- row_size defines the size of the number of rows in the array.
- column_size defines the size of the number of columns in the array.

Example: `int array[10][15];`

Memory Occupied by Two Dimensional Array

$\text{Total Memory} = \text{row_size} * \text{column_size} * \text{sizeof}(\text{data_type})$

Consider the example, `int array[10][15];`

`row_size = 10`

`column_size = 15`

`data_type = int`, so size of (int) = 2 bytes

Total Memory = $10 * 15 * 2$

= 100 bytes

Structure of Two Dimensional Array

- There are two index values, one of for representing position in terms of rows and another for representing position in terms of columns.

Consider, `int array[10][15];`

- Here, row index starts from 0 and ends with m-, where m is number of rows.
- Column index starts from 0 and ends with n-1, where n is the number of columns.

array[0][0] holds first element
 array[0][1] holds the second element
 :
 array[m-1][n-1] holds the last element

array[0][0]	array[0][1]	array[0][2]	array[0][n-1]
array[1][0]	array[1][1]	array[1][2]	array[1][n-1]
.
.
.
array[m-1][0]	array[m-1][1]	array[m-1][n-1]

Initialization of Two Dimensional Array

Syntax:

```
data_type    array_name[row_size][column_size]={ { a1,a2,...,an }, { b1,b2,...,bn }, . . . , { z1,z2,...,zn } };
```

Here,

- row_size specifies number of rows and column_size specifies the number of columns.
- { a₁,a₂,...,a_n } are the values assigned to I row.
- { b₁,b₂,...,b_n } are values assigned to II row.

Representation of Two-Dimensional Array in Memory

A Two-Dimensional array elements are stored in continuous memory locations. It can be represented in memory in the following ways,

- i. Column-Major Order
- ii. Row-Major Order

Column Major Order

In this method the elements are stored column wise, i.e., m elements of first column are stored in first m locations, m elements of second column are stored in next m locations and so on.

A 3 X 4 array is stored as follows,

Elements	Subscript	
	(0,0)	← Column 1
	(1,0)	
	(2,0)	
	(0,1)	← Column 2
	(1,1)	
	(2,1)	
	(0,2)	← Column 3
	(1,2)	
	(2,2)	
	(0,3)	← Column 4
	(1,3)	
	(2,3)	

We can find out the location of any element by using following formula:

$$\text{LOC (A [J, K])} = \text{Base (A)} + w [\text{M (K)} + (\text{J})]$$

LOC (A [J, K]) is the location of the element in the Jth row and Kth column.

Base (A) is the base address of the array A.

w is the number of bytes required to store single element of the array A.

M is the total number of rows in the array.

J is the row number of the element and K is the column number of the element.

Example: Consider a 3 X 4 array represented below,

int A[3][4] = {{1,2,3,4},{5,6,7,8},{9,10,11,12}};

Subscript	Elements	Address
(0,0)	1	1000
(1,0)	5	1002
(2,0)	9	1004
(0,1)	2	1006
(1,1)	6	1008
(2,1)	10	1010
(0,2)	3	1012
(1,2)	7	1014
(2,2)	11	1016
(0,3)	4	1018
(1,3)	8	1020
(2,3)	12	1022

Suppose we have to find the location of A [2, 3]. The required values are:

$$\text{Base (A)} = 1000$$

$$w = 2 \text{ (because an integer takes 2 bytes in memory in 16 bit machine)}$$

$$M = 3$$

$$J = 2$$

$$K = 3$$

Now put these values in the given formula as below:

$$\text{LOC (A [2, 1])} = 1000 + 2 [3 (3) + (2)]$$

$$= 1000 + 2 [3 (3) + 2]$$

$$= 1000 + 2 [9 + 2]$$

$$= 1000 + 2 [11]$$

$$= 1000 + 22$$

$$= 1022$$

Row-Major Order

In this method the elements are stored row wise, i.e. n elements of first row are stored in first n locations, n elements of second row are stored in next n locations and so on.

A 3 x 4 array will stored as below,

Elements	Subscript
	(0,0)
	(0,1)
	(0,2)
	(0,3)
	(1,0)
	(1,1)
	(1,2)
	(1,3)
	(2,0)
	(2,1)
	(2,2)
	(2,3)

} ← Row 1
 } ← Row 2
 } ← Row 3

We can find out the location of any element by using following formula:

$$\text{LOC (A [J, K])} = \text{Base (A)} + w [N (J-1) + (K-1)]$$

Here

LOC (A [J, K]) is the location of the element in the J^{th} row and K^{th} column.

Base (A) is the base address of the array A.

w is the number of bytes required to store single element of the array A.

N is the total number of columns in the array.

J is the row number of the element.

K is the column number of the element.

Example: Consider a 3 X 4 array represented below,

`int A[3][4] = { { 1,2,3,4}, {5,6,7,8}, {9,10,11,12} };`

Subscript	Elements	Address
(0,0)	1	1000
(0,1)	2	1002
(0,2)	3	1004
(0,3)	4	1006
(1,0)	5	1008
(1,1)	6	1010
(1,2)	7	1012
(1,3)	8	1014
(2,0)	9	1016
(2,1)	10	1018
(2,2)	11	1020
(2,3)	12	1022

Suppose we have to find the location of A [2, 1]. The required values are:

Base (A) = 1000

w = 2 (because an integer takes 2 bytes in memory in 16 bit machine)

N = 4

J = 2

$K = 1$

Now put these values in the given formula as below:

$$\text{LOC}(A[3, 2]) = 1000 + 2[4(2) + (1)]$$

$$= 1000 + 2[4(2) + 1]$$

$$= 1000 + 2[8 + 1]$$

$$= 1000 + 2[9]$$

$$= 1000 + 18$$

$$= 1018$$

Types of Two Dimensional Array Initialization

i) Initializing all specified memory locations

Consider,

```
int a[4][3] = {{11,12,13},{44,55,66},{33,66,99}};
```

Here, 4 rows and 3 columns will be reserved for array 'a' and all 4 rows and 3 columns are initialized with some value as shown below

		0	1	2
Rows	0	11	12	13
	1	44	55	66
	2	11	10	9
	3	33	66	99
		Columns		

ii) Partial Array Initialization:

- Here, number of values to be initialized is less than the size of an array. The remaining location will be initialized to zero automatically.

Example:

```
int a[4][4] = {{11,22},{33,44},{55,66},{77,88}};
```


- Here, the array 'a', has 4 rows and 4 columns, of which only first two columns of each row are initialized

		0	1	2	3
Rows ↓	0	11	22	0	0
	1	33	44	0	0
	2	55	66	0	0
	3	77	88	0	0
		Columns →			

Initialization can also be done as below:

```
int a[4][3] = { 10,11,12,13,14,15,16,17,18,19,20,21,22};
```

```
int a[ ][3] = { 10,11,12,13,14,15,16,23,34,45,56,67,78}
```

Note: While initializing 2D array, it is compulsory to mention the column_size, where as row_size is optional.

```
int a[4][ ] = { 10,20,30,40};
int a[ ][ ] = { 15,30,45,60};
```

} Both are illegal because
column size not mentioned.

iii) Using for loop to initialize the 2D array blocks:

Example:

```
int a[4][3];
for(i=0 ; i<4 ; i++)
{
    for(j=0 ; j<3 ; j++)
    {
        printf("a[%d][%d] = ");
        scanf("%d",&a[i][j]);
    }
}

/*Reading and displaying elements from a two dimensional array*/
#include<stdio.h>
```

```

#include<conio.h>
void main()
{
int  a[20][2],n,m,i,j;
printf("Enter number of rows and columns\n");
scanf("%d%d",&m,&n);
printf("Enter the %d elements\n",m*n);
for(i=0; i<m ; i++)
{
for(j=0; j<m ; j++)
{
scanf("%d",&a[i][j]);
}
}
printf("Array Elements are\n");
for(i=0; i<m ; i++)
{
for(j=0; j<m ; j++)
{
printf("%d\t",a[i][j]);
}
printf("\n");
}
getch();
}

```

Output:

Enter number of rows and columns

2

3

Enter 6 array elements

11 22 33 44 55 66

Array Elements are

11 22 33

44 55 66

Multi-Dimensional Arrays

- C allows array of three or more dimensions.

The general form of a multi dimensional array is,

data_type array_name[s1][s2][s3]...[sn];

Where,

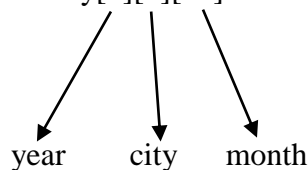
- data_type is the type of items in the array (int,float,char etc.).
- array_name is the name to represent the array.
- s1 is size of 1st dimension.
- s2 is size of 2nd dimension.
- s3 is size of 3rd dimension.
- :
- s_n is size of nth dimension.

Example: `int survey[3][5][12];` {Total memory = $3*5*12*\text{sizeof}(\text{int})$ }

- In the above example array survey may represent a survey data of rain fall during the three years from January to December in five cities.
- If the first index denotes year, the second index city and third index month.
- The three dimensional array can be represented as a series of two dimensional arrays shown as below,

Consider,

`survey[2][3][10]` → Denotes the rainfall in the month of October during second year in second city



Representation of Multidimensional Arrays

In C, multidimensional arrays are represented using the array-of-array representation.

Consider an array declared as follows,

`a[upper0][upper1][upper2]...[uppern-1]`, then the number of elements in the array is,

$$\prod_{i=0}^{n-1} \text{upper}_i$$

Where Π is the product of the upper_i's. For Example, if we declare a as `a[10][10][10]`, then we require $10 \times 10 \times 10 = 1000$ units of storage to hold the array. Multidimensional arrays are represented using row major order and column major order.

Row major order stores multidimensional arrays by rows.

For Example consider the two-dimensional array, $A[\text{upper}_0][\text{upper}_1]$ as upper_0 rows, $\text{row}_0, \text{row}_1, \dots, \text{row}_{\text{upper}_0-1}$, each row containing upper_1 elements.

If we assume that α is the base address i.e. address of $A[0][0]$, then the address of $A[i][0]$ is, $\alpha + i \cdot \text{upper}_1$ because there are i rows, each of size upper_1 , preceding the first element in the i^{th} row.

It should be then multiplied by the size of the element.

The address of an arbitrary, $a[i][j]$, is $\alpha + i \cdot \text{upper}_1 + j$.

To represent a three dimensional array, $A[\text{upper}_0][\text{upper}_1][\text{upper}_2]$ it is interpreted as upper_0 two-dimensional arrays of dimension $\text{upper}_1 \times \text{upper}_2$.

$$A[i][0][0] = \alpha + i \cdot \text{upper}_1 \cdot \text{upper}_2$$

Obtain formula for

$$A[i][j][k] = \alpha + i \cdot \text{upper}_1 \cdot \text{upper}_2 + j \cdot \text{upper}_2 + k$$

Generalizing on the preceding discussion we obtain the addressing formula for any element

$A[i_0][i_1][i_2] \dots A[i_{n-1}]$ in an n -dimensional array which is declared as,

$A[\text{upper}_0][\text{upper}_1] \dots [\text{upper}_{n-1}]$ If α is the address of $A[0][0] \dots [0]$ then the address of $a[i_0][0][0]$ is: $\alpha + i \cdot \text{upper}_1 \cdot \text{upper}_2 \dots \text{upper}_{n-1}$

The address of $a[i_0][i_1][0] \dots [0]$ is,

$$\alpha + i_0 \cdot \text{upper}_1 \cdot \text{upper}_2 \dots \text{upper}_{n-1} + i_1 \cdot \text{upper}_2 \cdot \text{upper}_3 \dots \text{upper}_{n-1}$$

The address of $A[i_0][i_1][i_2] \dots A[i_{n-1}]$ is,

$$\begin{aligned} & \alpha + i_0 \text{upper}_1 \cdot \text{upper}_2 \dots \text{upper}_{n-1} \\ & + i_1 \text{upper}_2 \cdot \text{upper}_3 \dots \text{upper}_{n-1} \\ & + i_2 \text{upper}_3 \cdot \text{upper}_4 \dots \text{upper}_{n-1} \\ & \cdot \\ & \cdot \\ & \cdot \\ & + i_{n-2} \text{upper}_{n-1} + i_{n-1} \\ & = \alpha + \left(\sum_{j=0}^{n-1} i_j a_j \right) * \text{sizeof}(\text{type}) \end{aligned}$$

$$\text{where, } a_j = \prod_{k=j+1}^{n-1} \text{upper}_k \quad 0 \leq j < n-1$$

$$a_{n-1} = 1$$

Advantages of Arrays

- Arrays can be used to store numerous values of the same data type (homogeneous data).
- Programmer is freed from creating many variables. A single array of required size can be created.
- Reading and writing from and to the array is simple (loops can be used for this purpose).

Disadvantages of Arrays

- A single array cannot store heterogeneous data (data of different types).
- Arrays demands contiguous memory locations to store data.
- Size of the array has to be mentioned at the beginning of the program itself. This may lead to inefficient memory utilization since all the reserved space may not be utilized.
- Addition and deletion of elements at the middle of the array is problematic.

Strings

- String is an array of characters or a pointer to a portion of memory containing ASCII characters.
- A string can also be defined as sequence of zero or more characters followed by a NULL '\0' character.
- char type values are enclosed in single quotes.
- Example: 'a' is char value.
- String type values are enclosed in double quotes.
- Example: "a" is string value.

Declaring String Variables

Syntax:

```
char string_name[string_size];
```

Where,

- char is data type of strings.
- string_name is name of string variables.
- string_size is length of string which is to be stored.
 - string_size is the length of string i.e. no. of characters stored in the string (excluding NULL character).
 - Example: char name[21];
 - Here, 20 characters can be stored and 1 null character should be appended at the end.

Initialization of String

```
char name[11]={ 'P','R','A','K','H','Y','A','T','H','\0'};  
OR  
char name[11]={ "PRAKHYATH"};
```

- Both are same, if we specify characters separately we should use ' ' for each character and finally enclosing all the characters within flower braces { }, and if we directly specify entire string at a time we use " ".

- In the above example,

```
char name[11]={ 'P','R','A','K','H','Y','A','T','H','\0' };
```

name[0] holds 'P' name[1] holds 'R' name[2] holds 'A' and so on name[9] hold '\0' remaining allocated memory will be wasted.

Consider the Example,

```
char name[11]="PRAKHYATH";
```

- Here string is initialized as a unit, and null character is supplied by the compiler automatically.
- The values stored in these two strings are identical.
- If the string itself has a double quote, an escape character is needed in-front of it to tell the compiler that it is not a delimiter.
- Similarly it has to be done to include backslash(\) in string.

Example:

- i) won't fit `char str1[20]="won\t fit";`
 ii) the "King" `char str2[20]="the \"king\" ";`
 iii) c:\mydocs `char str3[20]="c:\\mydocs";`

i)

w	o	n	'	t		f	i	t	\0
---	---	---	---	---	--	---	---	---	----

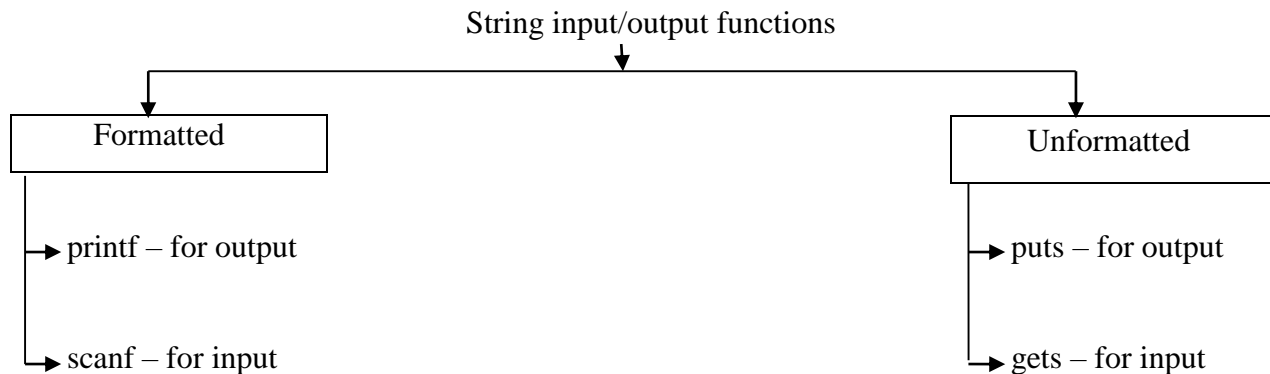
ii)

t	h	e		"	k	i	n	g	"	\0
---	---	---	--	---	---	---	---	---	---	----

iii)

c	:	\	m	y	d	o	c	s	\0
---	---	---	---	---	---	---	---	---	----

String Input/Output Functions



*/*Program to illustrate formatted I/O for strings*/*

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

```
void main()
```

```
{
```

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

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

```
scanf("%s",name);
```

```
printf("Your name is %s",name);
```

```
}
```

Output:

Enter name : Prakhyath

Your name is Prakhyath

*/*Program to illustrate unformatted I/O for strings*/*

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

```
void main()
```

```
{
```

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

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

```
gets(name);
```

```
printf("Your name is");
```

```
puts(name);
```

```
}
```

Output:

Enter name : Prakhyath

Your name is Prakhyath

Array of Strings / Multidimensional Strings

- Two dimensional character array which consists of strings as its individual elements, is said to be an array of strings.

Syntax:

```
char string_name[size_of_rows][size_of_columns];
```

Example : `char names[3][10];`

Initialization

```
char day[7][10]={"Sunday","Monday","Tuesday","Wednesday","Thursday","Friday",
"Saturday"};
```

- The above initialization creates 7 rows and 10 columns using which different strings are stored in day array as shown below.
- Last character of each string is always null character “\0”.
- day[0] contains first string i.e. “Sunday”.
day[1] contains second string i.e. “Monday”.
:
day[6] contains seventh string i.e. “Saturday”.

	0	1	2	3	4	5	6	7	8	9
0	S	u	n	d	a	y	\0			
1	M	o	n	d	a	y	\0			
2	T	u	e	s	d	a	y	\0		
3	W	e	d	n	e	s	d	a	y	\0
4	T	h	u	r	s	e	d	a	y	\0
5	F	r	i	d	a	y	\0			
6	S	a	t	u	r	d	a	y	\0	

} Memory is created for the
above initialization as illustrated

String Manipulation Functions

String Functions	Description
<code>strlen(str_name);</code>	This function returns the length of the string
<code>strcpy(destination_str,source_str);</code>	This function copies the string from one variable to another variable
<code>strncpy(destination_str,source_str,length);</code>	This function also copies the string from one variable to another variable, but only upto the specified length
<code>strcmp(string1,string2);</code>	This function is used to compare two strings. They are case sensitive.
<code>stricmp(string1,string2);</code>	This function is also used to compare two strings, but they are not case sensitive.
<code>strncmp(string1,string2,length);</code>	This function compares two strings only upto a specified length. They are case sensitive.
<code>strnicmp(string1,string2,length);</code>	This function also compares two strings only upto a specified length, but not case sensitive.
<code>strlwr(string_name);</code>	This function converts upper case character to lower case.
<code>strupr(string_name);</code>	This function converts lower case character to upper case.
<code>strcat(string1,string2);</code>	This function is used to concatenate (join) two strings.
<code>strrev(string_name);</code>	This function is used to reverse the characters in a given string.
<code>strset(string,symbol);</code>	This function replaces all the characters of a string with a given symbol or character.
<code>strnset(string,symbol,length);</code>	This function also replaces all the characters of a string with a given symbol or character but only to a specified length.
<code>strstr(string_name, pattern);</code>	Return pointer to start of pattern in string string_name

strlen():

```

/*Illustrating string length function i.e strlen( ) */
#include<stdio.h>
#include<conio.h>
#include<string.h>
main( )
{
char a[30];
int l;
printf("Enter a string");
scanf("%s",a);
l=strlen(a);
printf("\n Length of string is %d\n",l);
getch( );
}

```

Output:

Enter a string:Sahyadri
Length of sting is 8

strcpy():

```

/*Program to illustrate strcpy( ) function*/
#include<stdio.h>
#include<string.h>
main( )
{
char a[10],b[10];
printf("Enter a string\n");

scanf("%s",a);
strcpy(b,a);
printf("Value of a:%s\n",a);
printf("Value of b:%s\n",b);
}

```

Outpt:

Enter a string PCD
Value of a PCD
Value of b PCD

strncpy():

```
/*Program to illustrate strcpy( ) */  
#include<stdio.h>  
#include<string.h>  
main( )  
{  
char a[10],b[10];  
printf("Enter a string\n");  
scanf("%s",a);  
strncpy(b,a,3);  
printf("Value of a: %s\n",a);  
printf("Value of b: %s\n",b);  
}
```

Output:

Enter a string :yuvraj
Value of a :yuvraj
Value of b :yuv

strcmp():

```
/* Program to illustrate strcmp( ) function*/  
#include<stdio.h>  
#include<string.h>  
main()  
{  
char a[10],b[10];  
int n;  
printf("Enter string one:\n");  
scanf("%s",a);  
printf("Enter string two:\n");  
scanf("%s",b);  
n=strcmp(a,b);  
if(n==0)  
{  
printf("Both strings are equal\n");  
}
```

Output:

Enter string one :dravid
Enter string two :dravid
Both strings are equal

```
}  
else  
{  
printf("Strings are not equal");  
}  
}
```

strcmp():

```
/*Program to illustrate strcmp( ) function*/  
#include<stdio.h>  
#include<conio.h>  
#include<strings>  
main( )  
{  
char a[30],b[30];  
int n;  
printf("Enter string one :");  
scanf("%s",a);  
printf("Enter string two:");  
scanf("%s",b);  
n=strcmp(a,b);  
if(n==0)  
{  
printf("Both strings are equal\n");  
}  
else  
{  
printf("Strings are not equal\n");  
getch( );  
}
```

Output:

```
Enter string one :raydu  
Enter string two :RAYDU  
Both strings are equal
```

strncmp():

```
/* Program to illustrate strcmp( ) function*/  
#include<stdio.h>  
#include<conio.h>  
#include<strings>  
void main( )  
{  
char a[30],b[30];  
int n;  
printf("Enter string one:");  
scanf("%s",a);  
printf("Enter string two:");  
scanf("%s",b);  
n=strncmp(a,b,2);  
if(n==0)  
{  
printf("Both strings are equal up to 2 characters");  
}  
else  
{  
printf("Strings are not equal");  
}  
getch();  
}
```

Output :

Enter string one: ragav

Enter string two: ragev

Both strings are equal upto 3 characters

strnicmp():

```
/* Program to illustrate strnicmp() function*/  
#include<stdio.h>  
#include<conio.h>  
#include<string.h>  
main()  
{  

```

```
{
char a[30],b[30];
int n;
printf("Enter string one:");
scanf("%s",a);
printf("Enter string two:");
scanf("%s",b);
n=strncmp(a,b,2);
if(n==0)
{
printf("Both strings are equal upto first 2 characters");
}
else
{
printf("Strings are not equal");
}
getch();
}
```

Output:

Enter string one:devi

Enter string two:debi

Both strings are equal upto first 2 characters

strlwr():

```
/*Program to illustrate strlwr() function*/
#include<stdio.h>
#include<string.h>
main()
{
char a[30];
printf("Enter a string in upper case:\n");
scanf("%s",a);
printf("Final string is:%s",strlwr(a));
}
```

Output:

Enter a string in uppercase :DEVI

Final string is :devi

strupr():

```
/*Program to illustrate strupr() function*/
#include<stdio.h>
#include<conio.h>
#include<string.h>
void main()
{
char a[30];
printf("Enter a string in lowercase:");
scanf("%s",a);
printf("Final string is :%s",strupr(a));
getch();
}
```

Output:

Enter a string in lowercase :kohli
Final string is :KOHLI

strcat():

```
/*Program to illustrate strcat() functions*/
#include<stdio.h>
#include<conio.h>
#include<string.h>
main()
{
char a[30],b[30];
printf("Enter string one:");
scanf("%s",a);
printf("Enter string two:");
scanf("%s",b);
strcat(a,b);
printf("Final string is:%s",a);
getch();
}
```

Output:

Enter string one:Prakhyath
Enter string two:Rai
Final string is:PrakhyathRai

strrev():

```

/*Program to illustrate strrev() fuctions*/
#include<stdio.h>
#include<conio.h>
#include<string.h>
main()
{
char a[30];
printf("Enter string:");
scanf("%s",a);
strrev(a);
printf("Reversed string :%s",a);
getch( );
}

```

Output:

Enter string:Prakhyath
Reversed string is:htayhkarP

strset():

```

/*Program to illustrate strset() fuctions*/
#include<stdio.h>
#include<conio.h>
#include<string.h>
main( )
{
char a[30];
char b;
printf("Enter a string:");
gets(a);
printf("Enter a symbol to replace the string:");
scanf("%c",&b);
strset(a,b);
printf("After strset:%s",a);
getch();
}

```

Output:

Enter a string:Vikhyath
Enter a symbol to replace the string:#
After strset:#####

```
}
```

strnset():

```
/*Program to illustrate strnset() fuctions*/
#include<stdio.h>
#include<conio.h>
#include<string.h>
main( )
{
char a[30];
char b;
printf("Enter a string:");
scanf("%s",a);
printf("Enter a symbol to replace the string:");
scanf("%c",&b);
strnset(a,b,3);
printf("After strnset:%s",a);
getch();
}
```

Output:

Enter a string:Vikhyath

Enter a symbol to replace the string:#

After strnset:###hyath

Example: Program to sort the strings in alphabetical order

```
/*Program to sort strings in alphabetical order*/
#include<stdio.h>
#include<conio.h>
#include<string.h>
main( )
{
char str[20][25],temp[25];
int i,j,n;
printf("How Many Strings");
```

```
scanf("%d",&n);
for(i=0;i<n;i++)
{
printf("Enter string %d",i+1);
scanf("%s",str[i]);
}
for(i=0;i<n-1;i++)
{
for(j=0;j<(n-1-i);j++)
{
if(strcmp(str[j],str[j+1])>0)
{
strcpy(temp,str[j]);
strcpy(str[j],str[j+1]);
strcpy(str[j+1],temp);
}
}
}
printf("Sorted strings are :\n");
for(i=0;i<n;i++)
{
printf("%s\n",str[i]);
}
}
```

Output:

How Many Strings

5

Enter string 1

Ajay

Enter string 2

Vicky

Enter string 3

Bharath

Enter string 4

Rahul

Enter string 5

Chethan

Sorted strings are:

Ajay

Bharath

Chethan

Rahul

Vicky

Structures

A Structure is a collection of one or more variables of same or different data types grouped together under single tag name.

Structures provide mechanism to create new user defined data type. Structure is a user defined data type that can store related information about an entity. Structure is nothing but records about particular entity.

Structure Declaration

A structure is declared using the keyword **struct** followed by a structure tag. All member variables of structure are declared within structures. A ; (semicolon) should be placed after definition of structure.

Syntax:

```
struct structure_tag
{
    data_type member_variable1;
    data_type member_variable2;
    .
    .
    .
    data_type member_variablen;
};
```

Example:

```
struct student
{
    char name[20];
    int rollno,age;
    float cgpa;
};
```

Declaration of Structure Variables

Syntax:

```
struct structure_tag
{
    data_type member_variable1;
    data_type member_variable2;
    .
    .
    .
    data_type member_variablen;
}structure_variable_name;
```

Or

```
struct structure-tag structure_variable_name;
```

Example 1:

Consider the structure student defined earlier the variables for student structure can be declared as,

```
struct student s1;
```

Example 2:

Structure variables can be declared during structure definition as follows,

```
struct student
{
    char name[20];
    int rollno,age;
    float cgpa;
}s1;
```

Accessing the member of structure is done by member access operator . (dot operator).

```
structure_variable_name.member;
```

Example:

```
#include<stdio.h>

struct student
{
    char name[20];
    int rollno,age;
    float cgpa;
}s1;

main ( )
{
    s1.name="Dhanraj";
    s1.rollno=15;
    s1.age=25;
    s1.cgpa=8.5;
    printf ("Student name is %s", s1.name);
    printf ("Student Roll Number is %s", s1.rollno);
    printf ("Student Age is %s", s1.age);
    printf ("Student CGPA is %s", s1.cgpa);
}
```

Output:

```
Student name is Dhanraj
Student Roll Number is 15
Student Age is 25
Student CGPA is 8.5
```

Initialization of Structure

Initialization of structure means assigning some constants to the members of structures.

Example 1:

```
#include<stdio,h>

struct student
{
    char  name[20];
    int   rollno,age;
    float cgpa;
};

main( )
{
    struct student stud={"Ashwin",5,25,8.7};
    printf("Name is %s\nRoll Number is %d\nAge is %d\nCGPA is %f", stud.name, stud.rollno,
    stud.age, stud.cgpa);
}
```

Output:

```
Name is Ashwin
Roll Number is 5
Age is 25
CGPA is 8.7
```

Example 2:

/* Write a C program to read an employee name, number, age, and salary. Also print the employee details using structure. */

```
#include<stdio.h>

struct employee
{
    char empname[25];
```

```
int empnumber;
int empage;
float empsalary;
};

main( )
{
    struct employee e1;
    printf("Enter Employee Name\n");
    scanf("%s",&e1.empname);
    printf("Enter Employee Number\n");
    scanf("%d",&e1.empnumber);
    printf("Enter Employee Age\n");
    scanf("%d",&e1.empage);
    printf("Enter Employee Salary\n");
    scanf("%f",&e1.empsalary);
    printf("Employee Details are:\n");
    printf("Employee Name is %s\n",e1.empname);
    printf("Employee Number is %d\n",e1.empnumber);
    printf("Employee Age is %d\n",e1.empage);
    printf("Employee Salary is %f\n",e1.empsalary);
}
```

Output:

```
Enter Employee Name
Avneesh
Enter Employee Number
909
Enter Employee Age
25
Enter Employee Salary
54000
Employee Details are:
Employee Name is Avneesh
Employee Number is 909
Employee Age is 25
Employee Salary is 54000
```

Array of Structures

Considering the employee structure,

```
struct employee
{
    char empname[25];
    int empnumber;
    int empage;
    float empsalary;
};
```


If we want to read more than one employee details using same structure, then structure variable name has to be declared as an array. i.e.

```
struct employee emp[25];
```

Now,

emp[0].empname, emp[0].empnumber, emp[0].empage, emp[0].empsalary stores details of first employee.

emp[1].empname, emp[1].empnumber, emp[1].empage, emp[1].empsalary stores details of second employee.

Reading and Displaying of multiple employee records can be done using looping constructs.

Example:

```
/* Write a C program to read n employees details and display using structure concept. */
```

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

```
struct employee
```

```
{
```

```
char empname[25];
```

```
int empnumber;
```

```
int empage;
```

```
float empsalary;
```

```
};
```

```
main( )
```

```
{
```

```
struct employee e1[25];
```

```
int i,n;
```

```
printf("Enter number of Employees\n");
```

```
scanf("%d",&n);
```

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

```
{
```

```
printf("Enter Employee %d details \n",i+1);
```

```
printf("Enter Employee Name\n");
```

```
scanf("%s",e1[i].empname);
```

```
printf("Enter Employee Number\n");
scanf("%d",&e1[i].empnumber);
printf("Enter Employee Age\n");
scanf("%d",&e1[i].empage);
printf("Enter Employee Salary\n");
scanf("%f",&e1[i].empsalary);
}
for(i=0;i<n;i++)
{
printf("Employee %d details are\n",i+1);
printf("Employee Name is %s\n",e1[i].empname);
printf("Employee Number is %d\n",e1[i].empnumber);
printf("Employee Age is %d\n",e1[i].empage);
printf("Employee Salary is %f\n",e1[i].empsalary);
}
}
```

Nested Structures

Nested structure is a structure that contains another structure as its member.

Two ways to create nested structures are,

- a. Having structure variable as data member inside another structure.
- b. Declaring a structure inside another structure.

Structure variable as data member	Declaring a structure inside another structure
<p>Syntax:</p> <pre> struct structure1 { datatype structure1_member_name; . . . }; struct structure2 { datatype structure2_member_name; . . . struct structure1 structure1_variable; }; </pre>	<p>Syntax:</p> <pre> struct structure2 { datatype structure2_member_name; . . . struct structure1 { datatype structure1_member_name; . . . }structure1_variable; }; </pre>

Example:

```
/* Program to illustrate nested structures */
#include<stdio.h>
struct date_of_birth
{
    Int dd,mm,yr;
    Char dy[25];
};
struct employee
{
    char empname[25];
    int empnumber;
    int empage;
    float empsalary;
    struct date_of_birth dob;
};
main( )
{
    struct employee e1;
    printf("Enter Employee Name\n");
    scanf("%s",e1.empname);
    printf("Enter Employee Number\n");
    scanf("%d",&e1.empnumber);
    printf("Enter Employee Age\n");
    scanf("%d",&e1.empage);
    printf("Enter Employee Salary\n");
    scanf("%f",&e1.empsalary);
    printf("Enter Employee date of birth as date, month, year and day \n");
    scanf("%d%d%d%s",&e1.dob.dd,&e1.dob.mm,&e1.dob.yy,e1.dob.dy);
    printf("Employee Details are:\n");
    printf("Employee Name is %s\n",e1.empname);
```

```
printf("Employee Number is %d\n",e1.empnumber);  
printf("Employee Age is %d\n",e1.empage);  
printf("Employee Salary is %f\n",e1.empsalary);  
printf("Employee Date of Birth is%d:%d:%d, %s\n",e1.dob.dd,e1.dob.mm,e1.dob.yy,e1.dob.dy);  
}
```

Output:

Enter Employee Name

Vicky

Enter Employee Number

9

Enter Employee Age

25

Enter Employee Salary

45000

Enter Employee date of birth as date, month, year and day

9

9

1991

Monday

Employee Details are:

Employee Name is Vicky

Employee Number is 9

Employee Age is 25

Employee Salary is 45000.000000

Employee Date of Birth is 9:9:1991, Monday

Structures and Functions

Structure can be passed to functions and returned from functions.

Function can access members of structures in the following ways,

- a) Passing individual members of the structure to function
- b) Passing entire structure or structure variable to the function
- c) Passing address of structure to the function

Passing Individual Members

In this method the individual members are passed to the function i.e. in the function call structure members are passed as actual parameters.

```
/* Program to illustrate structures with functions (by passing individual members) */
```

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

```
void display(char n[],int en,int eage,float esal);
```

```
struct employee
```

```
{
```

```
char empname[25];
```

```
int empnumber;
```

```
int empage;
```

```
float empsalary;
```

```
};
```

```
main( )
```

```
{
```

```
struct employee e1;
```

```
printf("Enter Employee Name\n");
```

```
scanf("%s",&e1.empname);
```

```
printf("Enter Employee Number\n");
```

```
scanf("%d",&e1.empnumber);
```

```
printf("Enter Employee Age\n");
```

```
scanf("%d",&e1.empage);
```

```
printf("Enter Employee Salary\n");
```

```
scanf("%f",&e1.empsalary);
```

```
display(e1.empname,e1.empnumber,e1.empage,e1.empsalary);
```

```
}  
void display(char n[],int en,int eage,float esal)  
{  
printf("Employee Details are:\n");  
printf("Employee Name is %s\n",n);  
printf("Employee Number is %d\n",en);  
printf("Employee Age is %d\n",eage);  
printf("Employee Salary is %f\n",esal);  
}
```

Passing Entire Structure or Structure Variable

In this method the variable of the structure is passed as argument in the function call.

/* Program to illustrate structures with functions (by passing structure variable) */

```
#include<stdio.h>  
struct employee  
{  
char empname[25];  
int empnumber;  
int empage;  
float empsalary;  
};  
void display(struct employee e);  
main( )  
{  
struct employee e1;  
printf("Enter Employee Name\n");  
scanf("%s",&e1.empname);  
printf("Enter Employee Number\n");  
scanf("%d",&e1.empnumber);  
printf("Enter Employee Age\n");  
scanf("%d",&e1.empage);
```

```
printf("Enter Employee Salary\n");
scanf("%f",&e1.empsalary);
display(e1);
}

void display(struct employee e)
{
printf("Employee Details are:\n");
printf("Employee Name is %s\n",e.empname);
printf("Employee Number is %d\n",e.empnumber);
printf("Employee Age is %d\n",e.empage);
printf("Employee Salary is %f\n",e.empsal);
}
```

Passing Structure through Pointers

Structure can be passed to a function using pointer to the structure variable.

Pointer to a structure variable can be declared as follows,

```
struct structure_name structure_variable,*structure_pointer_name;

structure_pointer_name=&structure_variable;
```

/* Program to illustrate structures with functions (by passing structure through pointers) */

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

```
struct employee
```

```
{
```

```
char empname[25];
```

```
int empnumber;
```

```
int empage;
```

```
float empsalary;
```

```
};
```

```
void display(struct employee *e);
```

```
main( )
```



```
{
struct employee e1;
printf("Enter Employee Name\n");
scanf("%s",&e1.empname);
printf("Enter Employee Number\n");
scanf("%d",&e1.empnumber);
printf("Enter Employee Age\n");
scanf("%d",&e1.empage);
printf("Enter Employee Salary\n");
scanf("%f",&e1.empsalary);
display(&e1);
}

void display(struct employee *e)
{
printf("Employee Details are:\n");
printf("Employee Name is %s\n",e->empname);
printf("Employee Number is %d\n",e->empnumber);
printf("Employee Age is %d\n",e->empage);
printf("Employee Salary is %f\n",e->empsal);
}
```

Type Definition

The typedef keyword enables the programmer to create a new datatype name for an existing data type. Using typedef, an alternate name is given to data type.

Syntax:

```
typedef existing_data_type new_data_type_name;
```

Example:

```
typedef int integer;
```

In the above example the int datatype is renamed to integer using typedef keyword, hence now we can declare as integer a=10; which is equivalent to declaring int a=10;

Similarly we can apply for structure type as follows,

Without Using typedef	Using typedef
<pre>struct employee { char empname[25]; int empnumber; int empage; float empsalary; }; struct employee emp;</pre>	<pre>typedef struct employee { char empname[25]; int empnumber; int empage; float empsalary; }; employee emp;</pre>

Advantages of typedef

- typedef is used to make the construct shorter with more meaningful name for types already defined by C.
- alternate name can be given to known data type.

Self-Referential Structures

A self-referential structure is essentially a structure definition which includes at least one member that is a pointer to the structure of its own kind.

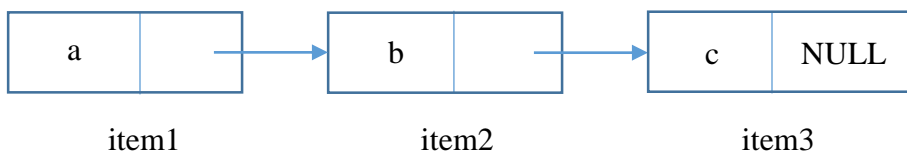
```
struct structure_name
{
    data_type member 1;
    data_type member 2;
    . . .
    structure_name *pointer;
};
```

Example:

```
struct list
```

```
{  
    char data;  
    struct list *link;  
};  
struct list item1,item2,item3;  
item1.data='a';  
item2.data='b';  
item3.data='c';  
item1.link=&item2;  
item2.link=&item3;  
item3.link=NULL;
```

A linear linked list illustration:



Unions

A **union** is a special data type available in C that allows to store different data types in the same memory location. We can define a union with many members, but only one member can contain a value at any given time. Unions provide an efficient way of using the same memory location for multiple-purpose.

Union Declaration

A union is declared using the keyword **union** followed by a union tag. All member variables of union are declared within union. A ; (semicolon) should be placed after definition of union.

Syntax:

```
union union_tag
{
    data_type member_variable1;
    data_type member_variable2;
    .
    .
    .
    data_type member_variablen;
};
```

Example:

```
union student
{
    char name[20];
    int rollno, age;
    float cgpa;
};
```

Declaration of Union Variables

Syntax

```
union union_tag
{
    data_type member_variable1;
    data_type member_variable2;
    .
    .
    .
    data_type member_variablenn;
}union_variable_name;
```

Or

```
union union_tag union_variable_name;
```

Example 1:

Consider the union student defined earlier the variables for student union can be declared as,
union student s1;

Example 2:

Union variables can be declared during union definition as follows,

```
union student
{
    char name[20];
    int rollno,age;
    float cgpa;
}s1;
```

Accessing the member of union is done by member access operator . (dot operator).

```
union_variable_name.member;
```

```
/*Program to illustrate the memory size occupied by a union*/
```

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

```
#include <string.h>
```

```
union Data
```

```
{
```

```
    int i;
```

```
    float f;
```

```
    char str[20];
```

```
};
```

```
int main( )
```

```
{
```

```
    union Data data;
```

```
    printf( "Memory size occupied by data: %d\n", sizeof(data))
```

```
    return 0;
```

```
}
```

Output:

Memory size occupied by data: 20

```
/*Program to illustrate the usage of unions*/
```

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

```
#include <string.h>
```

```
union Data
```

```
{
```

```
    int i;
```

```
    float f;
```

```
    char str[20];
```

```
};
```

```
int main( )
```

```
{
```

```
    union Data data;
```

```
    data.i = 10;
```

```
    printf( "data.i: %d\n", data.i);
```

Output:

data.i: 10

data.f: 220.500000

data.str: C Programming

```
data.f = 220.5;
printf( "data.f: %f\n", data.f);
strcpy( data.str, "C Programming");
printf( "data.str: %s\n", data.str);
return 0;
}
```

Note: In the above program we have used an union which contains 3 members, with a different data type. However we can use only one of them at a time. This is due to the fact that only one location is allocated for a union variable, irrespective of its size.

The compiler allocates a piece of storage that is large enough to hold the largest variable type in the union.

In the declaration above, the member str requires 20 bytes which is the largest among the numbers.

```
/*Program to illustrate the mistake in usage of unions*/
```

```
/*Program to illustrate the usage of unions*/
```

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

```
#include <string.h>
```

```
union Data
```

```
{
```

```
int i;
```

```
float f;
```

```
char str[20];
```

```
};
```

```
int main( )
```

```
{
```

```
union Data data;
```

```
data.i = 10;
```

```
data.f = 220.5;
```

Output:

```
data.i : 1917853763
```

```
data.f : 4122360580327794860452759994368.000000
```

```
data.str : C Programming
```

```
strcpy( data.str, "C Programming");  
printf( "data.i: %d\n", data.i);  
printf( "data.f: %f\n", data.f);  
printf( "data.str: %s\n", data.str);  
return 0;  
}
```

Note: In the above program we can see that the values of **i** and **f** members of union got corrupted because the final value assigned to the variable has occupied the memory location and this is the reason that the value of **str** member is getting printed very well.

Pointers

Every byte of memory (RAM) has a unique address associated with it.

The memory units are allocated to the variables declared in the program.

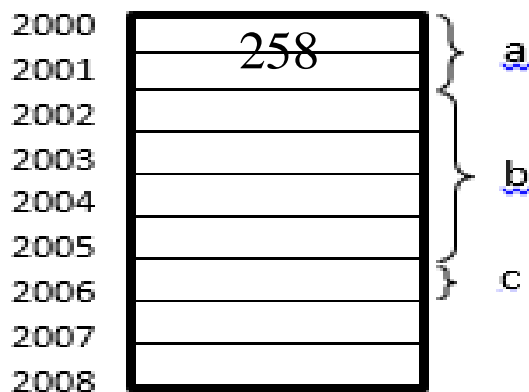
Example: If variable is of type character then 1 byte is allocated to the variable. Similarly if the variable is of type int then 2 bytes of memory is allocated.

Consider the following,

```
int a=258;
```

```
float b;
```

```
char c;
```



Address - It is a unique member associated with a memory unit.

Example: 258 is the data stored in the variable **a** and 2000 is the address of variable **a**.

/* Write a c program to print the contents and address of memory location */

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

```
void main()
```

```
{
```

```
int num=365;
```

```
printf("Contents of variable=%d",num);
```

```
printf("Address of memoy location=%x",&num);
```

```
}
```

(Note: format specifier %x or %p can be used to print address)

From the above example we see that,

- Contents of memory location can be obtained by mentioning the name of the variable.
- Address of memory location can be accessed by prefixing “&” to the variable name.

Address Operator (&) and Dereference Operator (*)

Address Operator

It is used with variables as:

`&variable-name`

The above expression yields the address of the variable

Note: Address cannot be used with constants and expressions

- `&100` // Invalid since 100 is a constant
- `&(a+5)` // Invalid since (a+5) is an expression
- `&'x'` // Invalid since 'x' is a constant (Character Constant)

Dereference Operator (*)

- It is a unary prefix operator.
- It can be used with any pointer variable as

`*pointer_variable`

- The above expression gives the value of the variable pointer.

Pointer and Address

A pointer provides access to a variable by using the address of that variable.

A pointer is a variable that stores the address of another variable.

Syntax:

`datatype *ptr_name;`

Example: `int a;`

`int *pa;`

`pa=&a;`

/* Program to illustrate the declaration and initialization of pointers */

```
#include<stdio.h>

main()
{
    int x=10;
    int *ptr;
    ptr=&x;
    printf("value of x is %d",*ptr);
}
```

Output: value of x is 10.

In above program ptr is the name of pointer variable. The ‘*’ operator informs the compiler that ptr is a pointer variable. **int** specifies that **ptr** will store address of an integer variable.

In statement, ptr=&x;

& operator retrieves the address of **x** and copies that to the contents of the pointer **ptr**. Hence **ptr** is assigned with the address of **x**.

‘*’ is called dereference operator or indirection operator.

Using pointers

Following steps are needed to use pointer

- 1) Declaring a pointer variable
- 2) Assigning an address to the pointer (initializing)
- 3) Dereferencing the pointer to address data i.e to access data.

Declaring a pointer variable

Syntax:

`datatype *pointer_variable;`

Here,

- **datatype** specifies the datatype of the pointer to be declared
- **pointer** variable is the name of the pointer
- ‘*’ is the dereferencing operator.

Note:

- Initially pointer variable contains garbage value this means that the pointer doesn't point to a specific variable, such pointers are also called as stray pointer or dangling pointer.
- If the pointer is initialized with zero value, the pointer doesn't point anywhere.
- Such pointers are called null pointers.

To declare multiple pointers.

Syntax:

```
datatype *ptr_var1,*ptr_var2,...,*ptr_varn;
```

Example:

```
char *p1,*p2,*p3;
int *pntr,*pm;
```

Pointer assignment and initialization

Syntax:

```
datatype *ptr=expression;
```

Example:

- | | | |
|------------------|---------------------|---------------------|
| 1. int a=10; | 2. float x=0.5; | 3. int *p=NULL; |
| int *p; | float *px=&x; | |
| p=&a; | | |

Accessing variable through pointers :

Consider simple expression involving pointer,

```
double d=10;
double *pd;
pd=&d;
```

Access the contents of variable by dereferencing the pointer.

In the above example **pd** is a pointer pointing to variable **d**. The value of variable **d** can be accessed by using ***p**. Here ***** operator acts as a dereference operator.

Example 1:

- 1) `*pd=2.0;` // assigns 2.0 to variable d
- 2) `x=*pd;` //value of d is assigned to x
- 3) `y=*pd +10.0` //This is equivalent to `y=d+10.0`
- 4) `*pd=*pd+5.0` //Adds 5.0 to d

Example 2:

- 1) `int x=1;`
- 2) `int *p = &x;`
- 3) `++*p;` // Increment x
- 4) `a=*p++;` //Assign value of x to a, then increment a

Some Common Mistakes

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

Here p is not declared as a pointer

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

Here p is declared as pointer but it is not assigned with address

Difference between Pointer Variable and Normal Variable

Pointer variable	Normal variable
A pointer variable holds address.	A normal variable holds data.
Syntax for pointer declaration is datatype *ptr_name;	General syntax is datatype var_name;
Dereferencing a pointer variable should be done to access data.	No dereferencing, Can access data by simply using the variable.

Pointers and Functions (Call by Reference)

- Values should be passed to the function or address can be passed.
- Passing the address is known as pass by reference (Call by reference).
- Here the address of the variable is passed during the function call in the actual parameter list.
- The pointers are declared in the formal parameter list. These pointers receive the address from the actual parameters.
- Pass by reference is used when the function needs be called with actual parameters.

/* Write a C Program using functions to swap the values of two variables */

```
#include<stdio.h>

void swap(int*,int*);

main()
{
    int x,y;
    x=10;
    y=20;
    printf("The numbers before swaping : ");
    printf("%d and %d\n",x,y);
    swap(&x,&y);
    printf("The numbers after swaping : %d and %d\n",x,y);
}

void swap(int *px,int *py)
{
    int temp;
    temp = *px;
    *px=*py;
    *py=temp;
}
```

Output:

The numbers before swapping: 10 and 20

The numbers after swapping: 20 and 10

Pointers and Arrays

Any operation that can be achieved by array subscripting can also be done with pointers.

Example:

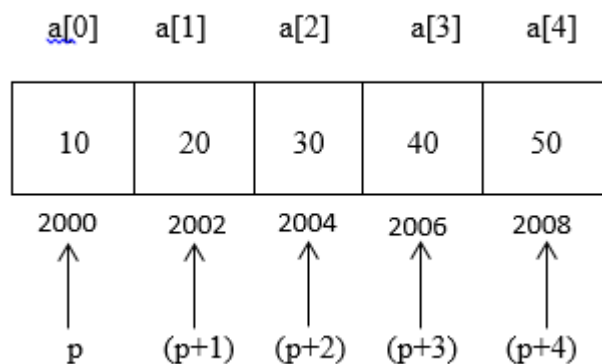
```
int a[5]={ 10,20,30,40,50};
```

From the diagram alongside, we can see that the starting address of the array is 2000.

This address is called as a Base address of the array. Pointer to the Array can be created as:

```
int *p;
```

```
p=a;
```



Once the pointer is created to the array, the elements present in the array can be accessed using pointers.

Note: In the above example identifier ‘a’ holds the address of the 1st array element.

To point the next element in the array, increment the pointer (p++).

To point the previous element in the array, decrement the pointer (p--).

To point to ith element in the array wrt current position of pointer, use p=p+i;

Example:

```
/*Program to print the array element using pointers */
```

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

```
void main()
```

```
{
```

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

```
int *p;
```

```
p=a;
```

```
for(i=0;i<5;i++)
{
printf("%d\t",*p);
p++;
}
}
```

Passing Arrays to Functions using Pointers

```
/* Program to find the sum of 5 array elements using functions */
#include<stdio.h>
int sumarray(int *p);
void main()
{
int a={ 1,2,3,4,5},i,sum;
sum=sum_array(a);
printf("sum=%d",sum);
}
int sum_array(int*p)
{
int result=0,i;
for(i=0;i<5;i++)
{
result=result+*p;
p++;
}
return result;
}
```

Passing Arrays using Pointers to the Functions

- We can make use of pointers explicitly in the function header.
- i.e., use int* instead of int[] in the formal parameter list.

Example: `int sum_array(int*p)`
 instead of
 `int sum_array(int arr[])`

Example:

`/* Write a program in C using function to find length of a given string */`

```
#include<stdio.h>
int strlenh(char*s);
void main( )
{
char str[20];
int len;
printf("enter the string");
gets(ch);
len=strlength(s);
int strlenh(char *pstr)
{
int n;
for(r=0;*pstr!='\0';s++)
{
n++;
}
return n;
}
```

Returning Pointer from a Function

A function can also return a pointer to a datatype item of any type.

Example:

```
#include<stdio.h>
int * minfunction(int *,int*);
main( )
{
int a=10,b=20;
```

```

int *pa,*pb,*pmin;
pa=&a;
pb=&b;
pmin=minfunction(pa,pb);
printf("minimum=%d",*pmin);
{
int * minfunction(int*ptr a,int*ptr b)
{
if(*ptr a<*ptr b)
return ptr a;
else
return ptr b;
}

```

Pointer Arithmetic

Arithmetic operations can be performed on pointers.

Various operations that can be performed are:

- | | | |
|-------------------|-------------------|-------------|
| 1) Incrementation | 2) Decrementation | 3) Addition |
| 4) Subtraction | 5) Comparison. | |

Incrementation on Pointers

- increment (++) operator increases the value of a pointer by the data size of the datatype.

Example

```

char c;
char *pc=&c;
pc++; //This causes the pointer to increment by 1[since size of(char)=1]

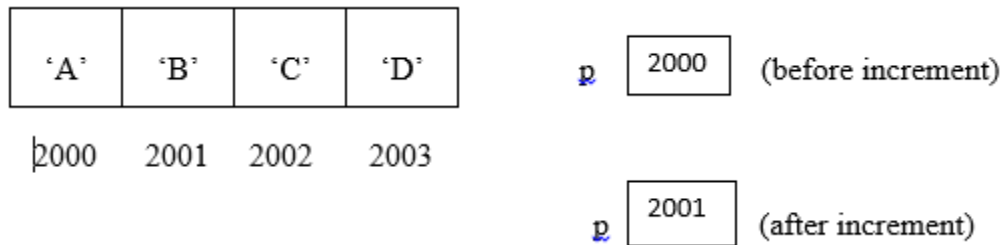
```

Similarly,

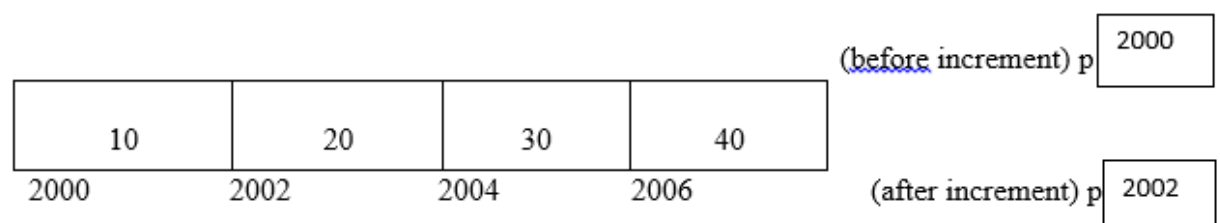
- incrementing pointer for integer causes the pointer to increment by 2 since size of int=2
- incrementing pointer for float causes the pointer to increment by 4 since size of float=4

Consider,

i. char *p;



ii. `int *p;`



Decrementation on Pointers

Decrementation(--) on pointers decreases the value of the pointer by the size of its declared data type, i.e.,

final value of pointer = current pointer value - size of (data type)

- for character pointer
final value = current value - 1
- for integer value
final value = current value - 2
- for float pointer
final value = current value - 4

Pointer Addition

It is possible to add any integer value to the pointer variable.

$p = p + i$ i = integer number

Final value of the pointer variable can be computed using the formula

final value of pointer = current value of pointer + integer number * size of data type

Pointer Subtraction

- It is possible to subtract any integer value to a pointer variable.

$p = p - i$ $i = \text{integer number}$

- final value of the pointer variable can be computed using the formula

final value of pointer = current value of pointer - integer number * size of data type

Comparison of Pointers

- Two pointers can be compared with each other only if both the pointers are pointing to similar type of data

Following operations can be used for comparisons

>	Greater than
<	Lesser than
>=	Greater than or equal to
<=	Lesser than or equal to
==	Equal to
!=	Not equal

/* Write a program in C to illustrate pointer comparison */

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

```
void main()
```

```
{
```

```
int *ptr1,*ptr2;
```

```
int a;
```

```
ptr1=&a;
```

```
ptr2=&a;
```

```
if(ptr1==ptr2)
```

```
    printf("The two pointers are pointing to same data");
```

```
else
```

```
    printf("The two pointers are pointing to different data");
```

```
}
```

Character Pointer

Declaration:

```
char *ptr_var;
```

- A character pointer can be used to access strings

Example:

```
char *a ="HELLO";
```

or

```
char ch[] ="HELLO";
```

```
char *a;
```

```
a=ch;
```

Using pointer **a**, the string can be printed as,

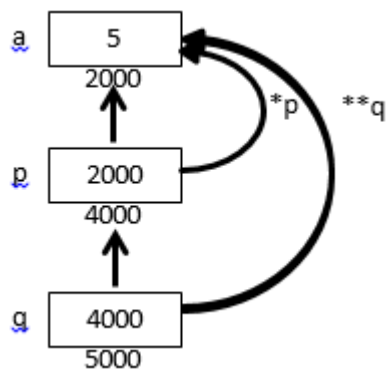
```
printf("%s\n",a);
```

Pointer to Pointer

- address of normal variable are stored in single pointer
- Address of pointer is stored in double pointer. they are called as pointers to pointers

Example:

```
datatype **doubleptr;
```



```
int a=5;
int *p;
int **q;
p=&a;
q=&p;
```

To access the value of the variable using the double pointer **q**, we need to use ****q**

Pointer to Arrays

The pointer can themselves be stored in an array.

Syntax:

`datatype pointerarray[size];`

In the above syntax,

`data_type` → type of data the `pointer_array` dereferences

`pointerarray` → name of the array consisting of the pointers

`size` → size of the array

Example:

```
int *ptr[100];
```

This declares pointer as an array of 10 integer pointers. Thus each element in `ptr` holds a pointer to integer value

```
/*Program to illustrate array of pointers */
```

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

```
void main()
```

```
{
```

```
int var[]={ 10,20,30};
```

```
int i, *ptr[3];
```

```
for(i=0;i<3;i++)
```

```
{
```

```
ptr[i] = &var[i];
```

```
printf("%d\n",*ptr[i]);
```

```
}
```

```
}
```

Output:

10

20

30

Note: The above `for()` loop can also be written using pointer arrays

We can also use pointers to store the list of strings as follows:

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

```
main()
```

```
{
```

```
int i;
char *names[]={“abc”,“def”,“ghi”};
for(i=0;i<3;i++)
printf(“%s\n”names[i]);
}
```

Double Pointers

Declaring double pointers

Syntax:

```
datatype variable;
datatype *pointer;
pointer=&variable;
datatype **doublepointer; //double pointer declaration
doublepointer=&pointer;
```

// C program to demonstrate pointer to pointer

```
int main()
{
    int var = 789;
    // pointer for var
    int *ptr2;
    // double pointer for ptr2
    int **ptr1;
    // storing address of var in ptr2
    ptr2 = &var;
    // Storing address of ptr2 in ptr1
    ptr1 = &ptr2;
    // Displaying value of var using
    // both single and double pointers
    printf("Value of var = %d\n", var );
```

Output:

Value of var = 789

Value of var using single pointer = 789

Value of var using double pointer = 789

```

printf("Value of var using single pointer = %d\n", *ptr2 );
printf("Value of var using double pointer = %d\n", **ptr1);
return 0;
}

```

Dynamic Memory Allocation (Allocating memory at runtime)

When C program is compiled, the compiler allocates memory to store different data elements such as constants, variables, arrays, structures. This is called compile time allocation (or static memory allocation).

Disadvantages of static memory allocations:

- i. The static allocation is done in the memory allocation to the program. This memory is limited in size.
- ii. In case of fixed arrays, the array size cannot be increased if we need to include more elements also there are few elements, array size cannot be to same memory.
- iii. Advanced data structures like linked list, trees and graphs cannot be created. These structures are essential in most real life programming.

Dynamic Memory Allocation

- In this memory is allocated at runtime instead of compile time.
- In C, there are 4 library routines called memory management functions for the purpose of memory management.

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

Note:

- Global variable and program code is stored in permanent storage.
- Local variables are stored in memory called stack.
- Dynamic allocation is done in memory called heap. Size of heap varies throughout the program execution.

i. malloc():

Syntax:

```
datatype *ptr=(datatype *) malloc(size in bytes);
```


malloc() is used to allocate contiguous block of memory from heap.

This function returns pointer to the allocated memory block of successful allocation otherwise it returns NULL.

Example: To allocate memory for 50 integers,

```
int * ptr;  
ptr =(int *) malloc(50*sizeof(int));
```

ii. calloc():

calloc() is used to allocate memory for the array elements and to initialize them with 0. It has two parameters.

Syntax:

```
datatype *ptr=(datatype*)calloc(n blocks,size of each block);
```

Here **n blocks** - number of blocks to be allocated

calloc() initializes all the bytes to zero and returns a pointer to the first byte of the allocated region.

Example: `p=(int*)calloc(3,2);` → 3 blocks of 2 bytes each will be allocated.

iii. realloc():

The realloc() function is used to modify the size of the previously allocated memory block.

Syntax:

```
datatype *ptr = realloc(ptr,newsize);
```

New memory of size mentioned in new size is allocated to the pointer ptr.

This function returns the pointer to the first byte of the new memory block.

New size may be smaller or larger than the format size.

The new memory block may not be allocated at the same location as the original block. The original block is lost.

Consider the code snippet below,

```
int *arr = malloc(2 * sizeof(int));  
arr[0] = 1;  
arr[1] = 2;
```

```
arr = realloc(arr, 3 * sizeof(int));
arr[2] = 3;
```

iv. **free():**

This is used to deallocate/free the memory block which has been allocated using malloc(), calloc() or realloc()

Once memory is deallocated, the pointer is set to NULL.

Syntax:

```
free(ptr);
```

/* Write a program in C to compute the sum of 2 arrays and store the result in third array.

Allocate the memory for three arrays at runtime */

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

```
#include<alloc.h>
```

```
void main()
```

```
{
```

```
    int *x,*y,*z;
```

```
    int i,num;
```

```
    printf("enter the size of arrays");
```

```
    scanf("%d",&num);
```

```
    x = (int *)malloc(num*size of (int));
```

```
    y = (int*)malloc(num*size of(int));
```

```
    z = (int*)malloc(num*size of(int));
```

```
    printf("Enter 1st array elements");
```

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

```
        scanf("%d",x+i);
```

```
    printf("Enter 2nd array elements");
```

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

```

scanf("%d",y+1);

printf("sum of 2 arrays is");
for(i=0;i<num;i++)
{
    *(z+1)=*(x+i)+*(y+i);
    printf("%d\n",*(z+1));
}

free(x);
free(y);
free(z);
}

/* Note: After each malloc( ), calloc( ) statement we can check if the pointer is pointing to NULL
*/
x=(int*)malloc()
if(x==NULL)
{
    printf("error");
    exit(0);
}

/* Program to illustrate calloc function */
#include <stdio.h>    /* printf, scanf, NULL */
#include <stdlib.h>    /* calloc, exit, free */
int main ()
{
    int i,n;
    int * pData;
    printf ("Amount of numbers to be entered: ");
    scanf ("%d",&n);
    pData = (int*) calloc (n,sizeof(int));

```

```

if (pData==NULL) exit (1);
for (i=0;i<n;i++)
{
    printf ("Enter number #%d: ",i+1);
    scanf ("%d",&pData[i]);
}
printf ("You have entered: ");
for (i=0;i<n;i++)
printf ("%d ",pData[i]);
free (pData);
return 0;
}

```

Dynamically Allocated Arrays

One Dimensional Arrays

Consider the code below,

```

int i, n, *list;
printf("Enter the number of numbers to generate:");
scanf("%d", &n);
if(n < 1)
{
    fprintf(stderr, "Improper value of n\n");
    exit(EXIT_FAILURE);
}
MALLOC(list, n * sizeof(int));

```

In the above code snippet MALLOC is a macro as defined below,

```

#define MALLOC(p,s) \
    If(!((p) = malloc (s))) {\
        fprintf(stderr, "Insufficient memory");\
        exit(EXIT_FAILURE);\
    }

```

```
}
```

calloc()

The function **calloc** allocates a user specified amount of memory and initializes the allocated memory to 0 (i.e., all allocated bits are set to 0); a pointer to the start of the allocated memory is returned. In case there is insufficient memory to make the allocation, the returned value is NULL. Consider the statements below,

```
int *x;
x = calloc(n, sizeof(int));
```

The above statements can be used to define a one dimensional array of integers, capacity of this array is n, and x[0:n-1] are initially 0. A macro CALLOC as to be defined as follows,

```
#define CALLOC(p,n,s) \
    If(!((p) = calloc (n,s))) {\
        fprintf(stderr, "Insufficient memory");\
        exit(EXIT_FAILURE);\
    }
```

realloc()

The function **realloc** resizes memory previously allocated by either malloc or calloc.

Consider the statement,

```
realloc(p, s);
```

The above statement changes the size of the memory block pointed at by p to s. The contents of the first min{s, oldsize} bytes of the block are unchanged as a result of this resizing.

- When $s > \text{oldsize}$, the additional $s - \text{oldsize}$ have a new unspecified value and,
- When $s < \text{oldsize}$, the rightmost $\text{oldsize} - s$ bytes of the old block are freed.

When realloc is able to do the resizing, it returns a pointer to the start of the new block and when it is unable to do the resizing, the old block is unchanged and the function returns the value NULL.

The below code snippet defines a macro for calloc,

```
#define REALLOC(p, s) \
    If(!((p) = realloc (p,s))) {\
        fprintf(stderr, "Insufficient memory");\
    }
```

```

        exit(EXIT_FAILURE);\
    }

```

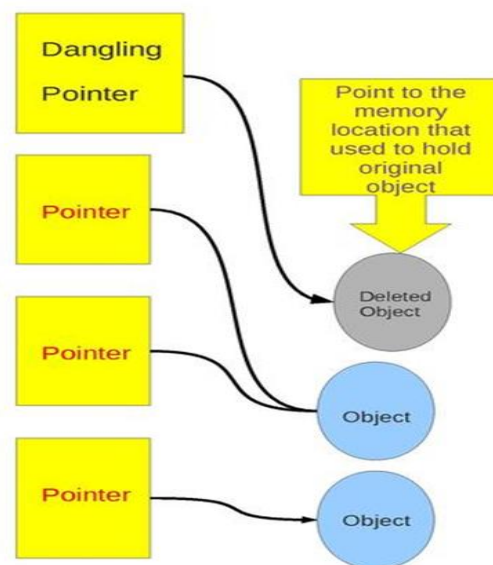
Dangling Pointer

Problem

Dangling pointers arise when an object that has an incoming reference is deleted or de-allocated, without modifying the value of the pointer, so that the pointer still points to the memory location of the deallocated memory.

Solution

After de-allocating memory, initialize the pointer to NULL



```

void function( )
{
    int *ptr = (int *)malloc(SIZE);
    .....
    .....
    free(ptr); //ptr now becomes dangling pointer which is pointing to dangling reference
}

```

In above example we first allocated a memory and stored its address in ptr. After executing few statements we deallocated the memory. Now still ptr is pointing to same memory address so it becomes dangling pointer.

To solve this problem just assign NULL to the pointer after the deallocation of memory that it was pointing. It means now pointer is not pointing to any memory address.

```
void function( )
{
    int *ptr = (int *)malloc(SIZE);
    .....
    .....
    free(ptr); //ptr now becomes dangling pointer which is pointing to dangling reference
    ptr=NULL; //now ptr is not dangling pointer
}
```

Two Dimensional Arrays

Consider the code snippet below for creating a two dimensional array at run time,

```
int** make2dArray(int rows, int cols)
{
    /* Create a two dimensional rows and cols array */
    int **x, i;
    MALLOC(x, rows * sizeof(*x)); //get memory for row pointers
    /*get memory for each row*/
    for(i=0; i < rows; i++)
        MALLOC(x[i], cols * sizeof(**x));
    return x;
}
```

The above function may be used in the following way,

```
int **myArray;
myArray = make2dArray(5,10); //allocates memory for a 5 by 10 two dimensional array of integers
myArray[2][4]=6; //assigns the value 6 to the [2][4] element of this array
```

Polynomials and Sparse Matrices

A polynomial is a sum of terms, where each term has a form $\mathbf{ax^e}$, where \mathbf{x} is the variable, \mathbf{a} is the coefficient, and \mathbf{e} is the exponent.

Example:

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

The largest (or leading) exponent of a polynomial is called its **degree**.

Consider the two polynomials,

$$A(x) = \sum a_i x^i \text{ and } B(x) = \sum b_i x^i$$

$$A(x) + B(x) = \sum (a_i + b_i) x^i$$

$$A(x) \cdot B(x) = \sum (a_i x^i \cdot \sum b_j x^j)$$

Polynomial Representation

Polynomials can be represented in C using typedef as follows,

```
#define MAX_DEGREE 101

typedef struct
{
    int degree;
    float coef[MAX_DEGREE];
} polynomial;

polynomial a; // a is object of polynomial
```

Now if \mathbf{a} is of type `polynomial` and $\mathbf{n} < \mathbf{MAX_DEGREE}$, the polynomial $A(x) = \sum_{i=0}^n x^i a_i$ would be represented as,

$\mathbf{a.degree = n}$;

$\mathbf{a.coef[i] = a_{n-i}, \quad 0 \leq i \leq n}$

An array is used to store the different coefficients of the polynomial. The size of the array is equal to the **degree+1** where **degree** is the degree of the polynomial,

Example: $A(x) = x^6 + 10x^3 + 3x^2 + 1$

	1	0	3	10	0	0	1
index →	0	1	2	3	4	5	6

ADT Polynomial is

objects: $p(x) = a_1x^{e_1} + \dots + a_nx^{e_n}$; a set of sorted pairs of $\langle e_i, a_i \rangle$ where a_i in Coefficients and e_i in Exponents, e_i are integers ≥ 0

functions:

for every poly, poly1, poly2 \in Polynomial, coef \in Coefficient, expon \in Exponents

Polynomial Zero() ::= return the polynomial $p(x)=0$

Boolean isZero(poly) ::= return (poly == 0)

Coefficient Coeff(poly , expon) ::= **If** (expon \in poly) return its corresponding coefficient **else** return 0

Exponent LeadExp(poly) ::= return the degree of poly

Polynomial Remove(poly , expon) ::= **If** (expon \in poly) remove the corresponding term and return the new poly **else** return ERROR

Polynomial SingleMult(poly , coef , expon) ::= return poly \times coef \times ~~expon~~

Polynomial Add(poly1 , poly2) ::= return poly1 + poly2

Polynomial Mult(poly1 , poly2) ::= return poly1 \times poly2

end Polynomial

We store the coefficients in order of decreasing exponents, such that **a.coef [i]** is the coefficient of x^{n-i} provided a term with exponent $n-i$ exists, otherwise, **a.coef [i] = 0**.

For instance, if **a.degree** is much less than **MAX_DEGREE**, then most of the positions of in **a.coef [MAX_DEGREE]** is not needed.

/* 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

As illustrated above the same argument applies if the polynomial is **sparse**, that is, the number of terms with nonzero coefficients is small relative to the degree of the polynomial. To **preserve space** an alternative representation is illustrated below which uses only one global array, **terms**, to store all the polynomials.

The C declarations are,

```

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

The polynomials are stored in array **terms** as follows,

The index of first term of A and B is given by **startA** and **startB**, respectively, 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**. Example: startA = 0, finishA = 1, startB = 2, finish = 5, and avail = 6.

Consider the array representation of two polynomials,

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

The above representation does not impose any limit on the number of polynomials that can be place on **terms**. The only constraint is that the total number of nonzero terms must be no more that **MAX_TERMS**. To refer to a polynomial, **A(x)** we need to pass in **startA** and **finishA** (<start, finish> pair). Any polynomial **A** that has **n** nonzero terms has **startA** and **finishA** such that **finishA = startA + n - 1**.

Polynomial Addition

C function to add two polynomials A and B, to obtain $D = A + B$. To produce $D(x)$, padd function adds $A(x)$ and $B(x)$ term by term. Starting at position **avail**, **attach** places the terms of **D** into the array, **terms**. If there is not enough space in terms to accommodate D, an error message is printed to the standard error device and we exit the program with the error condition.

```
void padd (int starta, int finisha, int startb, int finishb, int * startd, int *finishd)
{
    /* add A(x) and B(x) to obtain D(x) */
    float coefficient;
    *startd = avail;
    while (starta <= finisha && startb <= finishb)
        switch (COMPARE(terms[starta].expon, terms[startb].expon))
```

```

{
    case -1: /* a expon < b expon */
        attach(terms[startb].coef, terms[startb].expon);
        startb++;
        break;
    case 0: /* equal exponents */
        coefficient = terms[starta].coef + terms[startb].coef;
        if (coefficient)
            attach (coefficient, terms[starta].expon);
        starta++;
        startb++;
        break;
    case 1: /* a expon > b expon */
        attach(terms[starta].coef, terms[starta].expon);
        starta++;
        break;
}

/* add remaining terms of A(x) */
for( ; startA <= finishA; startA++ )
    attach(terms[startA].coef, terms[startA].expon);

/* add remaining terms of B(x) */
for( ; startB <= finishB; startB++ )
    attach(terms[startA].coef, terms[startB].expon);

*finishD = avail - 1;
}

void attach( float coefficient, int exponent)
{ /* add a new term to the polynomial */
    if( avail >= MAX_TERMS)

```

```

{
    fprintf( stderr, "Too many terms in the polynomial\n");
    exit( EXIT_FAILURE);
}

terms[avail].coef = coefficient;

terms[avail++].expon = exponent;
}

```

Sparse Matrix

A matrix is a two-dimensional data object made of m rows and n columns, therefore having total $m \times n$ values. If most of the elements of the matrix have **0** value, then it is called a sparse matrix.

ADT Sparse Matrix is

Objects: a set of triples, $\langle \text{row}, \text{col}, \text{value} \rangle$, where row and column are integers and form a unique combination, and value comes from the set item.

functions:

for all $a, b \in \text{SparseMatrix}$, $x \in \text{item}$, i, j , maxCol , $\text{maxRow} \in \text{index}$

$\text{SparseMatrix Create(maxRow, maxCol) ::=$

return a SparseMatrix that can hold upto $\text{maxItems} = \text{maxRow} \times \text{maxCol}$ and whose maximum row size is maxRow and whose maximum column size is maxCol .

$\text{SparseMatrix Transpose}(a) ::=$

return the matrix produced by interchanging the row and column value of every triple.

$\text{SparseMatrix Add}(a, b) ::=$

if the dimensions of a and b are the same

return the matrix produced by adding corresponding items, namely those with identical row and column values.

else return error

```

SparseMatrix Multily(a, b) ::=
    if number of columns in a equals number of rows in b
    return the matrix d produced by multiplying a by b according to the
    formula:  $d[i][j] = \sum (a[i][k].b[k][j])$  where  $d(i, j)$  is the  $(i, j)^{\text{th}}$  element
    else return error.

```

Sparse Matrix Representation

```

SparseMatrix Create(maxRow, maxCol) ::=
#define MAX_TERMS 101 /* maximum number of terms + 1 */
typedef struct
{
    int col;
    int row;
    int value;
}term;
term a[MAX_TERMS];

```

Consider the sparse matrix below represented in triple < row, col, value >

	col0	col1	col2	col3	col4	col5		row	col	value	
row0	15	0	0	22	0	-15		a[0]	6	6	8
row1	0	11	3	0	0	0		a[1]	0	0	15
row2	0	0	0	-6	0	0		a[2]	0	3	91
row3	0	0	0	0	0	0		a[3]	0	5	11
row4	91	0	0	0	0	0		a[4]	1	1	3
row5	0	0	0	0	0	0		a[5]	1	2	28
								a[6]	2	3	22
								a[7]	4	0	-6
								a[8]	5	2	-15

Above figure shows how a sparse matrix is represented in the array a.

a[0].row contains the number of rows.

a[0].col contains the number of columns.

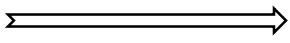
a[0].value contains the total number of nonzero entries.

Positions 1 through 8 store the triples representing the nonzero entries. The row index is in the field **row**, the column index is in the field **col**, and the value is in the field **value**.

Transposing the Sparse Matrix

To transpose a matrix the rows and columns are interchanged. i.e. each element $a[i][j]$ in the original matrix becomes $b[j][i]$ in the transpose matrix.

Example: Transpose of SparseMatrix a stored in b

	row	col	value			row	col	value
a[0]	6	6	8	Transpose of Sparse Matrix a 	b[0]	6	6	8
a[1]	0	0	15		b[1]	0	0	15
a[2]	0	3	22		b[2]	0	4	91
a[3]	0	5	-15		b[3]	1	1	11
a[4]	1	1	11		b[4]	2	1	3
a[5]	1	2	3		b[5]	2	5	28
a[6]	2	3	-6		b[6]	3	0	22
a[7]	4	0	91		b[7]	3	2	-6
a[8]	5	2	28		b[8]	5	0	-15

Consider the algorithm below,

for each row i

take element $\langle i, j, \text{value} \rangle$ and store it in as element $\langle j, i, \text{value} \rangle$ of the transpose.

There is a confusion exactly where to place the element $\langle j, i, \text{value} \rangle$ in the transpose matrix until we have processed all the elements that precede it. For instance,

$(0, 0, 15)$	becomes	$(0, 0, 15)$
$(0, 3, 22)$	becomes	$(3, 0, 22)$
$(0, 5, -15)$	becomes	$(5, 0, -15)$

If we place these triples consecutively in the transpose matrix, then, as we insert new triples, we must move elements to maintain the correct order.

This problem can be solved by using the column indices to determine the placement of elements in the transpose matrix. Consider the below algorithm which adopts this strategy,

for all elements in column j,

place element $\langle i, j, \text{value} \rangle$ in element $\langle j, i, \text{value} \rangle$

Consider the function transpose illustrated below,

The first array, **a**, is the original array, while the second array, **b**, holds the transpose. The variable **currentb**, holds the position in **b** that will contain the next transposed term. The terms in **b** are generated by rows, but since the rows in **b** correspond to the columns in **a**, the nonzero terms for row **i** of **b** are generated by collecting the nonzero terms from column **i** of **a**.

```
void transpose (term a[], term b[])
{
    int n, i, j, currentb;
    n = a[0].value;
    b[0].row = a[0].col;
    b[0].col = a[0].row;
    b[0].value = n;
    if (n > 0)
    {
        currentb = 1;
        for (i = 0; i < a[0].col; i++)
            /* transpose by 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++;
                }
            }
    }
}
```

Note: The above function is not efficient with respect to time its time complexity is more.

A much efficient algorithm (**fast_transpose**) is constructed in order to minimize the time complexity, illustrated as follows,

```
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_term [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;
        }
    }
}
```

Abstract Data Type String

ADT String is

Objects: a finite set of zero or more characters

functions:

for all $s, t \in \text{String}$, $i, j, m \in \text{non-negative integers}$

String Null(m) ::=

return a string whose maximum length is m characters, but is initially set to NULL

We write NULL as “”.

Integer Compare(s, t) ::=

if s equals t **return** 0

else if s precedes t **return** -1

else return +1

Boolean IsNull(s) ::=

if(Compare(s , NULL)) **return** FALSE

else return TRUE

Integer Length(s) ::=

if(Compare(s , NULL))

return the number of characters in s

else return 0

String Concat(s, t) ::=

if(Compare(t , NULL))

return a string whose elements are those of s followed by those of t

else return s

String Substr(s, i, j) ::=

if(($j > 0$) && ($i + j - 1 < \text{Length}(s)$))

return the string containing the characters of s at position $i, i+1, \dots, i+j-1$.

else return NULL.

Pattern Matching

Assume two strings, **string** and **pat**, where **pat** is a pattern to be searched for in **string**. The built-in function **strstr** can be used to perform this operation.

```
char pat[MAX_SIZE], string[MAX_SIZE], *t;
```

The built in function **strstr** is illustrated as follows,

```
if( t = strstr(string, pat))

    printf("The string from strstr is: %s\t",t);

else

    printf("The pattern was not found with strstr\n");
```

The call (**t = strstr(string,pat)**) returns a null pointer if **pat** is not in **string**. If **pat** is in **string**, **t** holds a pointer to the start of **pat** in **string**. The entire string beginning at position **t** is printed out.

Techniques to improve pattern matching,

- By quitting when `strlen(pat)` is greater than the number of remaining characters in the string.
- Checking the first and last characters of `pat` and `string` before we check the remaining characters.

Pattern Matching by Checking End Indices First

```
int nfind(char *string, char *pat)

{ /* match the last character of pattern first, and then match from the beginning */

    int i, j, start=0;

    int lasts =strlen(string)-1;

    int lastp= strlen(pat)-1;

    int endmatch=lastp;

    for(i=0; endmatch <= lasts; endmatch++, start++)

    {

        if(string[endmatch]==pat[lastp])

            for(j=0,i=start;j<lastp && string[i] == pat[j]; i++,j++)

                ;

    }
```

```

        if(j== lastp)
            return start; /*successful */
        }
        return -1;
    }

```

Kruth, Morris, Pratt Pattern Matching 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;
}

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[i];
        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;
    }
}
```