

CLASS
12

Computer Technology and Programming-II)

Class
12

Computer Technology and Programming-II)

Computer Technology and Programming-II)

Class -XII



**BOARD OF SECONDARY EDUCATION
RAJASTHAN, AJMER**

Textbook Development Committ
Computer Technology and Programming-II)
Class -XII

H R Choudhary

Assistant Professor, Government Engineering College

Ajmer, Rajasthan

Authors:

Vishnu Prakash Sharma

Assistant Professor

Government Engineering College

Ajmer, Rajasthan

Anil Kumar Tailor

Assistant Professor

Government Engineering College

Ajmer, Rajasthan

SYLLABUS DEVELOPMENT COMMITTEE

Class - XII

Computer Technology and Programming-II)

Convenor - Dr. Vishnu Goel, Director
Centre for E-Governance,
Govt. Khetan Polytechnic College, Jaipur

- Members -
1. Dr. Anil Gupta, Assistant Professor
Computer Science and Electronics Department
MBM Engineering College, Jodhpur
 2. Mr. Harji Ram Choudhary, Assistant Professor
Govt. Engineering College, Ajmer
 3. Mr. Dalpat Singh Songara, Assistant Professor
Govt. Girls Engineering College, Ajmer
 4. Mr. Amarjeet Punia, Assistant Professor
Govt. Girls Engineering College, Ajmer
 5. Mr. Vishnu Prakash Sharma, Assistant Professor
Govt. Engineering College, Ajmer
 6. Mr. Rajesh Kumar Tiwari, Principal
Govt. Senior Secondary School, Jotaya, Sarwar (Ajmer)

Preface

This book covers essential concepts of Computer Technology and Programming. The overall aim of the book is to introduce about Data Structure, C++, DBMS and their applications.

The topics covered in this book are according to the recently revised syllabus of Board of Secondary Education, Rajasthan. This book comprises of 15 chapters and each chapter has its own significance.

Chapters 1 to 5, are part of first unit and cover introduction to data structure which include arrays, sorting, stacks, queues and linked lists.

Chapter 6 to 12, are part of second unit and cover introduction to C++ programming language which include simple C++ programmes & their process of linking and compiling, operators, expressions, control structures, functions in C++, classes, objects, constructors, destructors, operator overloading and inheritance.

Chapter 13 to 15, are part of third unit and discuss about introduction to DBMS which include **DBMS concepts**, Relational database concepts and Basics of PL/SQL.

We take this opportunity to extend our heartfelt thanks to everyone who supported us in this endeavour. Our thanks to the people who helped in data collection, organisation of topics, editing and reviewing of contents and much more. Last but not least, we extend our thanks to our family members without whose support this dream could not have been converted into a REALITY.

Suggestions for improvement in the book are welcome.

Authors

Index

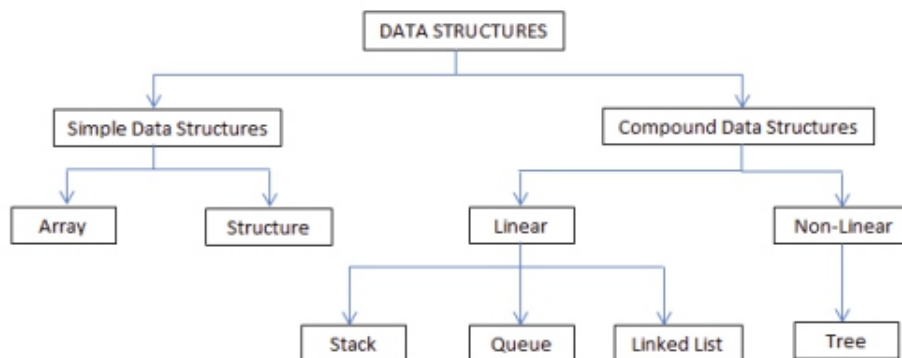
Chapter No.	Title	Page No.
Chapter 1	Introduction to Data Structures	1 - 4
Chapter 2	Array	5 - 31
Chapter 3	Sorting	32-53
Chapter 4	Stack	54-65
Chapter 5	Linked List	66-75
Chapter 6	Beginning with C++	76-85
Chapter 7	Operators, Expressions and Control Structures	86-93
Chapter 8	Functions in C++	94-98
Chapter 9	Classes and Objects	99-112
Chapter 10	Constructors and Destructors	113-123
Chapter 11	Operator Overloading	124-131
Chapter 12	Inheritance	132-148
Chapter 13	DBMS concepts	149-175
Chapter 14	SQL	176-217
Chapter 15	PL/SQL	218-233

Chapter 1

Introduction to Data Structures

Data Structure is a way of collecting and organising data in such a way that we can perform operations on these data in an effective way. Data Structures is about rendering data elements in terms of some relationship, for better organization and storage. Data structure is a logical and mathematical view of any organisations data. For example, we have data player's name "Virat" and age 26. Here "Virat" is of String data type and 26 is of integer data type. We can organize this data as a record like Player record. Now we can collect and store player's records in a file or database as a data structure. For example: "Dhoni" 30, "Gambhir" 31, "Sehwag" 33

Classification of Data Structures:



Simple Data Structures:

These are normally built from primitive data types like integers, real, character, Boolean etc. There are following two types of simple data structures

1. Array
2. Structure

Compound Data Structures:

Simple data structures can be combined in various ways to form more complex structures called compound data structures.

They are classified into the following two types:

1. Linear data structures

These data structures are single level data structures. A data structure is said to be linear if its elements form a sequence. There are the following types:

- a. Stack
- b. Queue
- c. Linked List

2. Non-linear data structures

These are multilevel data structures. Examples of non-linear data structure are Tree

and Graph.

Operations on Data Structures: The basic operations that are performed on data structures are as follows:

Insertion: Insertion means addition of a new data element in a data structure.

Deletion: Deletion means removal of a data element from a data structure if it is found.

Searching: Searching involves searching for the specified data element in a data structure.

Traversal: Traversal of a data structure means processing all the data elements present in it.

Sorting: Arranging data elements of a data structure in a specified order is called sorting.

Merging: Combining elements of two similar data structures to form a new data structure of the same type, is called merging.

Algorithm: An algorithm is a finite set of instructions or logic, written in order, to accomplish a certain predefined task. Algorithm is not the complete code or program, it is just the core logic (solution) of a problem, which can be expressed either as an informal high level description as pseudocode or using a flowchart.

An algorithm is said to be efficient and fast, if it takes less time to execute and consumes less memory space. The performance of an algorithm is measured on the basis of following properties:

Space Complexity

Time Complexity

Space Complexity

It's the amount of memory space required by the algorithm, during the course of its execution. Space complexity must be taken seriously for multi-user systems and in situations where limited memory is available. An algorithm generally requires space for following components:

Instruction Space: It's the space required to store the executable version of the program. This space is fixed, but varies depending upon the number of lines of code in the program.

Data Space: It's the space required to store all the constants and variables value.

Time Complexity

Time Complexity is a way to represent the amount of time needed by the program to run to completion.

Time Complexity of Algorithms:

Time complexity of an algorithm signifies the total time required by the program to run to completion. The time complexity of algorithms is most commonly expressed using the big O notation.

Time Complexity is most commonly estimated by counting the number of elementary functions performed by the algorithm. And since the algorithm's performance may vary with different types of input data, hence for an algorithm we usually use the worst-case time complexity of an algorithm because that is the maximum time taken for any input size.

Important Points

- Data structure is a logical and mathematical view of any organisations data.
- Simple data structures can be combined in various ways to form more complex structures called compound data structures.
- An algorithm is a finite set of instructions or logic, written in order, to accomplish a certain predefined task.
- Space complexity is the amount of memory space required by the algorithm, during the course of its execution. Space complexity must be taken seriously for multi-user systems and in situations where limited memory is available.

Exercise

Objective type questions.

- Q1. When determining the efficiency of algorithm, the space factor is measured by
- a. Counting the maximum memory needed by the algorithm
 - b. Counting the minimum memory needed by the algorithm
 - c. Counting the average memory needed by the algorithm
 - d. Counting the maximum disk space needed by the algorithm
- Q2. For an algorithm the complexity of the average case is
- a. Much more complicated to analyze than that of worst case
 - b. Much more simpler to analyze than that of worst case
 - c. Sometimes more complicated and some other times simpler than that of worst case
 - d. None or above
- Q3. When determining the efficiency of algorithm the time factor is measured by
- a. Counting microseconds
 - b. Counting the number of key operations
 - c. Counting the number of statements
 - d. Counting the kilobytes of algorithm
- Q4. Which of the following data structure is linear data structure?
- | | |
|-----------|------------------|
| a. Trees | b. Graphs |
| c. Arrays | d. None of above |
- Q5. Which of the following data structure is non linear data structure?
- | | |
|------------------|------------------|
| a. Arrays | b. Linked lists |
| c. Both of above | d. None of above |

Short answer type questions.

- Q1. What is Data Structure ?
- Q2. What are the two main measures for the efficiency of an algorithm ?
- Q3. Why time complexity is important ?
- Q4. Give example of Linear data structure.

Essay type questions.

Q1. How Space Complexity can be calculated ?

Q2. What are the uses of data structure?

Q3. Explain compound data structures ?

Answers

Ans1. a

Ans2. c

Ans3. b

Ans4. c

Ans4. d

Chapter 2

ARRAY

Definition of Array- An array is defined as **finite ordered** collection of **homogenous** data elements which are stored in contiguous memory locations.

Here the words,

finite means data range must be defined.

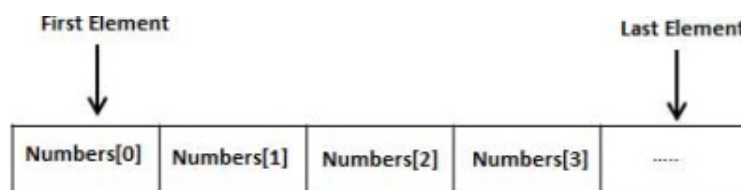
ordered means data must be stored in continuous memory addresses.

homogenous means data must be of similar data type.

There are two types of Array:

1. Single or One Dimensional Array
2. Multi Dimensional Array

Single or One Dimensional array: A list of items can be given one variable name using only one subscript and such a variable is called single sub-scripted variable or one or Single dimensional array.



Declaration of One Dimensional array: Like any other variable, arrays must be declared before they are used so that the compiler can allocate space for them in the memory. The syntax form of array declaration is:
type variable-name[size];

Ex -

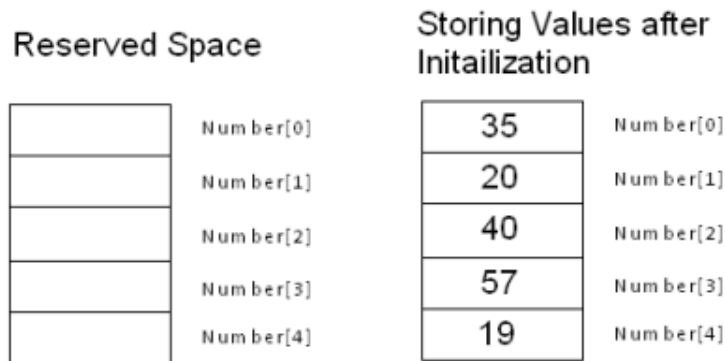
```
float height[50];  
int group[10];  
char name[10];
```

The type specifies the type of the element that will be contained in the array, such as int, float, or char etc. Variable-name specifies the name of array such as height, group and name. The size indicates the maximum number of elements that can be stored inside the array. C programming language also treats character strings simply as arrays of characters.

Now declare an array for five elements

```
int number[5];
```

Then the computer reserves five storage locations as the size of the array as shown below –



Initialization of Single or One Dimensional Array: After an array is declared, its elements must be initialized. In C programming an array can be initialized at either of the following stages:

- At compile time
- At run time

Compile Time initialization: Array can be initialized when it is declared. The general form of initialization of array is:

```
type array-name[size] = {list of values};
```

The values in the list are separated by commas. For example

```
int number[3] = {0,5,4};
```

The above statement will declare the variable “number” as an array of size 3 and will assign the values to each element. If the number of values in the list are less than the number of elements, then only that many elements will be initialized. The remaining elements values will be set to zero automatically.

Remember, if we have more initializers than the declared size, the compiler will produce an error.

Run time Initialization: An array can also be explicitly initialized at run time. For example consider the following segment of a C program.

```
for(i=0;i<10;i++)
{
    scanf("%d",&x[i]);
}
```

Above example will initialize array elements with the values entered through the keyboard. In the run time initialization of the arrays, looping statements are almost compulsory. Looping statements are used to initialize the values of the arrays one by one by using assignment operator or through the keyboard by the user.

Sample One Dimensional Array Program:

```
/* Simple C program to store the elements in the array and to print them from the array */
```

```

#include<stdio.h>
#include<conio.h>
void main()

{

int array[5],i;

printf("Enter 5 numbers to store them in array \n");

for(i=0;i<5;i++)

{

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

}

printf("Element in the array are - \n \n");

for(i=0;i<5;i++)

{

printf("Element stored at a[%d]=%d \n",i,array[i]);

}

getch();

}

```

Input – Enter 5 numbers to store them in array – 23 45 32 25 45

Output – Element in the array are –

Element stored at a[0]-23

Element stored at a[1]-45

Element stored at a[2]-32

Element stored at a[3]-25

Element stored at a[4]-45

Multi Dimensional Array:

Array of an array known as multidimensional array. The general form of a multidimensional array declaration –

type name[size1][size2]...[sizeN];

The simplest form of multidimensional array is the two-dimensional array. Example

```
int x[3][4];
```

Here, x is a two-dimensional (2d) array and can hold 12 elements. You can think the array as table with 3 row and each row has 4 column.

	Column 1	Column 2	Column 3	Column 4
Row 1	x[0][0]	x[0][1]	x[0][2]	x[0][3]
Row 2	x[1][0]	x[1][1]	x[1][2]	x[1][3]
Row 3	x[2][0]	x[2][1]	x[2][2]	x[2][3]

Initialization of Two Dimensional(2D)Array: Like the one dimensional array, 2D arrays can be initialized in two ways; the compile time initialization and the run time initialization.

Compile Time initialization – We can initialize the elements of the 2D array in the same way as the ordinary variables are declared. The best form to initialize 2D array is by using the matrix form. Syntax is as below –

```
int table[2][3]= {
    {0, 2, 5}
    {1, 3, 0}
};
```

Run Time initialization – As in the initialization of 1D array we used the looping statements to set the values of the array one by one. In the similar way 2D array are initialized by using the looping structure. To initialize the 2D array by this way, the nested loop structure will be used; outer for loop for the rows (first sub-script) and the inner for loop for the columns (second sub-script) of the 2D array. Below is the looping section to initialize the 2D array by using the run time initialization method –

```
for(i=0;i<3;i++)
{
    for(j=0;j<3;j++)
    {
        scanf("%d",&ar1[i][j]);
    }
}
```

Sample 2D array Program:

```
/* Sample 2-D array C program */

#include<stdio.h>
#include<conio.h>
void main()
```

```

{
    int array[3][3],i,j,count=0;

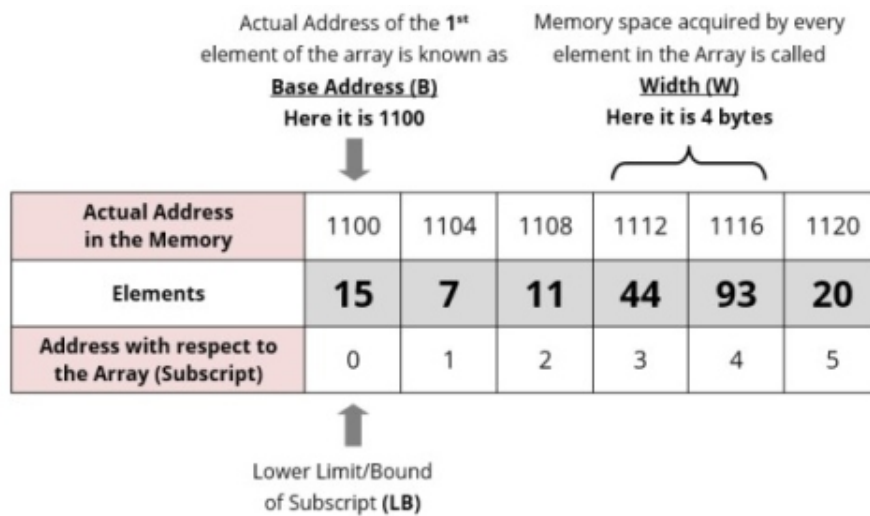
    /* Run time Initialization */
    for(i=1;i<=3;i++)
    {
        for(j=1;j<=3;j++)
        {
            count++;
            array[i][j]=count;
            printf("%d\t",array[i][j]);
        }
        printf("\n");
    }

    getch();
}

```

Output–
1 2 3
4 5 6
7 8 9

Address Calculation in Single (One) Dimensional Array:



Address of an element of an array say “A[I]” is calculated using the following formula:

$$\text{Address of A[I]} = B + W * (I - LB)$$

Where,

B = Base address

W = Storage Size of one element stored in the array (in byte)

I = Subscript of element whose address is to be calculate

LB = Lower limit / Lower Bound of subscript, if not specified assume 0 (zero)

Example:

Given the base address of an array B[1300.....1900] as 1020 and size of each element is 2 bytes in the memory. Find the address of B[1700].

Solution:

The given values are: B = 1020, LB = 1300, W = 2, I = 1700

$$\text{Address of A[I]} = B + W * (I - LB)$$

$$= 1020 + 2 * (1700 - 1300)$$

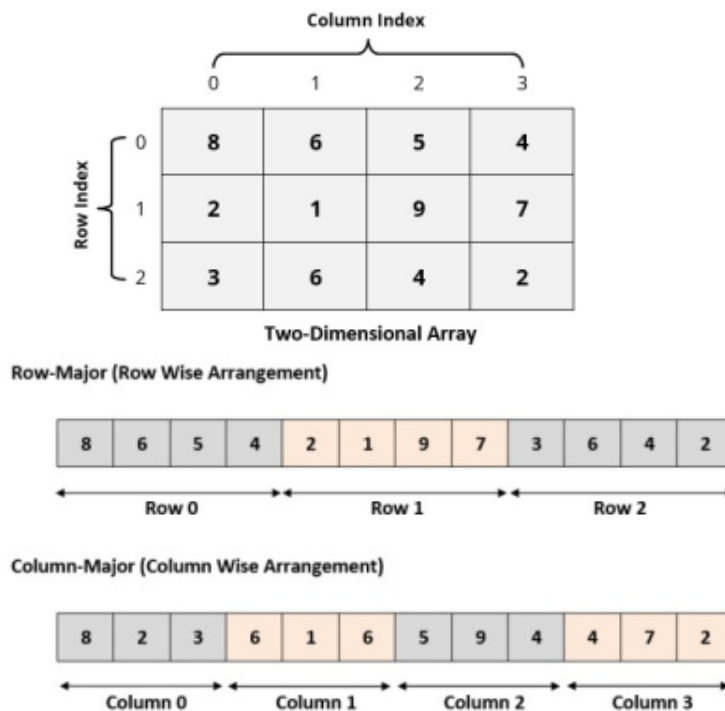
$$= 1020 + 2 * 400$$

$$= 1020 + 800$$

$$= 1820 \text{ [Ans]}$$

Address Calculation in Multi (Two) Dimensional Array:

While storing the elements of a 2-D array in memory, these allocations are contiguous memory locations. Therefore, a 2-D array must be linearized their storage. There are two alternatives to achieve linearization: Row-Major and Column-Major.



Address of an element of any array say “A [I][J]” can be calculated by two types as given below:

- (a) Row Major System
- (b) Column Major System

Row Major System:

The address of a location in Row Major System is calculated using the following formula:

$$\text{Address of A [I][J]} = B + W * [N * (I - L_r) + (J - L_c)]$$

Column Major System:

The address of a location in Column Major System is calculated using the following formula:

$$\text{Address of A [I][J] Column Major Wise} = B + W * [(I - L_r) + M * (J - L_c)]$$

Where,

B = Base address

I = Row subscript of element whose address is to be calculate

J = Column subscript of element whose address is to be calculate

W = Storage Size of one element stored in the array (in byte)

L_r = Lower limit of row/start row index of matrix, if not given assume 0 (zero)

L_c = Lower limit of column/start column index of matrix, if not given assume 0 (zero)

M = Number of row of the given matrix

N = Number of column of the given matrix

Note: Usually number of rows and columns of a matrix are given (like A[20][30] or A[40][60]) but if it is given as A[L_r- ---- - U_r, L_c- ---- - U_c]. In this case number of rows and columns are calculated using the following methods:

Number of rows (M) will be calculated as = (U_r - L_r) + 1

Number of columns (N) will be calculated as = (U_c - L_c) + 1

And rest of the process will remain same as per requirement (Row Major Wise or Column Major Wise).

Examples:

An array X [-15.....10, 15.....40] requires one byte of storage. If beginning location is 1500 determine the location of X [15][20].

Solution:

As you see here the number of rows and columns are not given in the question. So they are calculated as:

$$\text{Number of rows say } M = (U_r - L_r) + 1 = [10 - (-15)] + 1 = 26$$

$$\text{Number of columns say } N = (U_c - L_c) + 1 = [40 - 15] + 1 = 26$$

(i) Column Major Wise Calculation of above equation

The given values are: B = 1500, W = 1 byte, I = 15, J = 20, L_r = -15, L_c = 15, M = 26

$$\begin{aligned} \text{Address of A [I][J]} &= B + W * [(I - L_r) + M * (J - L_c)] \\ &= 1500 + 1 * [(15 - (-15)) + 26 * (20 - 15)] = 1500 + 1 * [30 + 26 * 5] = 1500 + 1 * [160] \end{aligned}$$

= 1660 [Ans]

(ii) Row Major Wise Calculation of above equation

The given values are: B = 1500, W = 1 byte, I = 15, J = 20, Lr = -15, Lc = 15, N = 26

Address of A [I][J] = B + W * [N * (I - Lr) + (J - Lc)]
= 1500 + 1 * [26 * (15 - (-15)) + (20 - 15)] = 1500 + 1 * [26 * 30 + 5] = 1500 + 1 * [780 + 5] =
1500 + 785
= 2285 [Ans]

Basic Operations on Array: Following operations can be performed on array

- (a) Traverse – access all the array elements one by one.
- (b) Insertion – Adds an element at the given index.
- (c) Deletion – Deletes an element at the given index.
- (d) Search – Searches an element using the given index or by the value.
- (e) Update – Updates an element at the given index.

Traverse: Traversing means accessing the each and every element of array exactly once. Following is the algorithm for traversing a linear array

Here A is a linear array with lower bound LB and upper bound UB. This algorithm traverses array A and applies the operation PROCESS to each element of the array.

1. Repeat For I = LB to UB
2. Apply PROCESS to A[I]
[End of For Loop]
3. Exit

Insertion: 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. Following is the algorithm for Insertion an element in to a linear array.

Algorithm: Let LA be a Linear Array (unordered) with N elements and K is a positive integer such that $K \leq N$. Following is the algorithm where ITEM is inserted into the Kth position of LA –

1. Start
2. Set J = N
3. Set N = N + 1
4. Repeat steps 5 and 6 while $J \geq K$
5. Set LA[J + 1] = LA[J]
6. Set J = J - 1
7. Set LA[K] = ITEM
8. Stop

C Program for Insertion:

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

```
main() {
```

```

int LA[] = {1,3,5,7,8};
int item = 10, k = 3, n = 5;
int i = 0, j = n;

printf("The original array elements are :\n");

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

n = n + 1;

while(j >= k) {
    LA[j+1] = LA[j];
    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]);
}
}

```

When we compile and execute the above program, it produces the following result –

The original array elements are :

```

LA[0] = 1
LA[1] = 3
LA[2] = 5
LA[3] = 7
LA[4] = 8

```

The array elements after insertion :

```

LA[0] = 1
LA[1] = 3
LA[2] = 5
LA[3] = 10
LA[4] = 7
LA[5] = 8

```

Deletion: Deletion refers to removing an existing element from the array and re-organizing all elements of an array. Following is the algorithm for Deletion an element from a linear array.

Consider LA is a linear array with N elements and K is a positive integer such that $K \leq N$.

Following is the algorithm to delete an element available at the Kth position of LA.

1. Start
2. Set J = K
3. Repeat steps 4 and 5 while J < N
4. Set LA[J-1] = LA[J]
5. Set J = J+1
6. Set N = N-1
7. Stop

C Program for Deletion:

```
#include <stdio.h>
main() {
    int LA[] = {1,3,5,7,8};
    int k = 3, n = 5;
    int i, j;

    printf("The original array elements are :\n");

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

    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]);
    }
}
```

When we compile and execute the above program, it produces the following result –
Output

The original array elements are :
LA[0] = 1
LA[1] = 3
LA[2] = 5
LA[3] = 7
LA[4] = 8
The array elements after deletion :

LA[0]= 1
LA[1]= 3
LA[2]= 7
LA[3]= 8

Search: Searching refers the finding out an element using the given index or by the value. There are two types of searching in linear array

1. Linear Search
2. Binary Search

Linear Search:

A linear search is the basic and simple search algorithm. A linear search searches an element or value from an array till the desired element or value is not found. It searches in a sequence order. It compares the element with all the other elements given in the list and if the element is matched it returns the value index else it return -1. Linear Search is applied on the unsorted or unordered list when there are fewer elements in a list.

Consider LA is a linear array with N elements and K is a positive integer such that $K \leq N$. Following is the algorithm to find an element with a value of ITEM using sequential search

1. Start
2. Set J=0
3. Repeat steps 4 and 5 while $J < N$
4. IF LA[J] is equal ITEM THEN GOTO STEP 6
5. Set J=J+1
6. PRINT J, ITEM
7. Stop

C Program for Searching:

```
#include <stdio.h>
Void main() {
    int LA[] = {1,3,5,7,8};
    int item = 5, n = 5;
    int i = 0, j = 0;

    printf("The original array elements are :\n");

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

    while(j < n){
        if(LA[j] == item) {
            break;
        }

        j = j + 1;
    }
}
```

```

printf("Found element %d at position %d\n", item, j+1);
}

```

When we compile and execute the above program, it produces the following result –
Output

The original array elements are:
LA[0]= 1
LA[1]= 3
LA[2]= 5
LA[3]= 7
LA[4]= 8
Found element 5 at position 3

Binary Search:

Binary Search is applied on the sorted array or list. In binary search, we first compare the value with the elements in the middle position of the array. If the value is matched, then we return the value. If the value is less than the middle element, then it must lie in the lower half of the array and if it's greater than the element then it must lie in the upper half of the array. We repeat this procedure on the lower (or upper) half of the array. Binary Search is useful when there are large numbers of elements in an array.

We basically ignore half of the elements just after one comparison.

Compare x with the middle element.

If x matches with middle element, we return the mid index.

Else If x is greater than the mid element, then x can only lie in right half subarray after the mid element. So we recur for right half.

Else (x is smaller) recur for the left half.

C Program for Binary Search:

```

#include <stdio.h>
#include <conio.h>
#define MAX 20

// array of items on which linear search will be conducted.
int intArray[MAX] = {1,2,3,4,6,7,9,11,12,14,15,16,17,19,33,34,43,45,55,66};

void printline(int count) {
    int i;

    for(i = 0; i < count-1; i++) {
        printf("=");
    }

    printf("\n");
}

```

```

int find(int data) {
    int lowerBound = 0;
    int upperBound = MAX - 1;
    int midPoint = -1;
    int comparisons = 0;
    int index = -1;

    while(lowerBound <= upperBound) {
        printf("Comparison %d\n", (comparisons + 1));
        printf("lowerBound : %d, intArray[%d] = %d\n", lowerBound, lowerBound,
            intArray[lowerBound]);
        printf("upperBound : %d, intArray[%d] = %d\n", upperBound, upperBound,
            intArray[upperBound]);
        comparisons++;

        // compute the mid point
        // midPoint = (lowerBound + upperBound) / 2;
        midPoint = lowerBound + (upperBound - lowerBound) / 2;

        // data found
        if(intArray[midPoint] == data) {
            index = midPoint;
            break;
        } else {
            // if data is larger
            if(intArray[midPoint] < data) {
                // data is in upper half
                lowerBound = midPoint + 1;
            }
            // data is smaller
            else {
                // data is in lower half
                upperBound = midPoint - 1;
            }
        }
    }
    printf("Total comparisons made: %d", comparisons);
    return index;
}

void display() {
    int i;
    printf("[");

    // navigate through all items
    for(i = 0; i < MAX; i++) {
        printf("%d", intArray[i]);
    }
}

```

```

    printf("\n");
}

main() {
    printf("Input Array: ");
    display();
    printline(50);

    //find location of 1
    int location = find(55);

    // if element was found
    if(location != -1)
        printf("\nElement found at location: %d" ,(location+1));
    else
        printf("\nElement not found.");
}

```

If we compile and run the above program then it would produce following result—
Output

Input Array: [1 2 3 4 6 7 9 11 12 14 15 16 17 19 33 34 43 45 55 66]

Comparison 1
lowerBound : 0, intArray[0] = 1
upperBound : 19, intArray[19] = 66
Comparison 2
lowerBound : 10, intArray[10] = 15
upperBound : 19, intArray[19] = 66
Comparison 3
lowerBound : 15, intArray[15] = 34
upperBound : 19, intArray[19] = 66
Comparison 4
lowerBound : 18, intArray[18] = 55
upperBound : 19, intArray[19] = 66
Total comparisons made: 4
Element found at location: 19

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

Consider LA is a linear array with N elements and K is a positive integer such that $K \leq N$. Following is the algorithm to update an element available at the Kth position of LA.

1. Start
2. Set $LA[K-1] = \text{ITEM}$
3. Stop

C Program for Updation:


```

#include <stdio.h>
#include <conio.h>
main() {
    int LA[] = {1,3,5,7,8};
    int k = 3, n = 5, item = 10;
    int i, j;

    printf("The original array elements are :\n");

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

    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]);
    }
}

```

When we compile and execute the above program, it produces the following result –
Output

```

The original array elements are :
LA[0] = 1
LA[1] = 3
LA[2] = 5
LA[3] = 7
LA[4] = 8
The array elements after updation :
LA[0] = 1
LA[1] = 3
LA[2] = 10
LA[3] = 7
LA[4] = 8

```

Character String in C: Strings are actually one-dimensional array of characters terminated by a null character '\0'. Thus a null-terminated string contains the characters that comprise the string followed by a null.

The following declaration and initialization create a string consisting of the word "Hello". To hold the null character at the end of the array, the size of the character array containing the string is one more than the number of characters in the word "Hello."

```
char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

If you follow the rule of array initialization then you can write the above statement as follows –

```
char greeting[] = "Hello";
```

Following is the memory presentation of the above defined string in C/C++ –

Index	0	1	2	3	4	5
Variable	H	e	l	l	o	\0
Address	0x23451	0x23452	0x23453	0x23454	0x23455	0x23456

Actually, you do not place the null character at the end of a string constant. The C compiler automatically places the '\0' at the end of the string when it initializes the array. Let us try to print the above mentioned string –

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

```
int main () {
```

```
    char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
    printf("Greeting message: %s\n", greeting);
    return 0;
```

```
}
```

When the above code is compiled and executed, it produces the following result –

Greeting message: Hello

C supports a wide range of functions that manipulate null-terminated strings –

S.N.	Function	Purpose
1	strcpy(s1, s2);	Copies string s2 into string s1.
2	strcat(s1, s2);	Concatenates string s2 onto the end of string s1.
3	strlen(s1);	Returns the length of string s1.
4	strcmp(s1, s2);	Returns 0 if s1 and s2 are the same; less than 0 if s1<s2; greater than 0 if s1>s2.
5	strchr(s1, ch);	Returns a pointer to the first occurrence of character ch in string s1.
6	strstr(s1, s2);	Returns a pointer to the first occurrence of string s2 in string s1.

The following example uses some of the above-mentioned functions –

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

```
int main () {
```

```

char str1[12]="Hello";
char str2[12]="World";
char str3[12];
int len;

/* copy str1 into str3 */
strcpy(str3, str1);
printf("strcpy( str3, str1): %s\n", str3);

/* concatenates str1 and str2 */
strcat( str1, str2);
printf("strcat( str1, str2): %s\n", str1 );

/* total length of str1 after concatenation */
len = strlen(str1);
printf("strlen(str1): %d\n", len);

return 0;
}

```

When the above code is compiled and executed, it produces the following result –

```

strcpy( str3, str1): Hello
strcat( str1, str2): HelloWorld
strlen(str1): 10

```

Static and Dynamic Memory Allocation: Dynamic memory allocation is at runtime. Static memory allocation is before run time, but the values of variables may be change at run time.

Static memory allocation saves running time, but can't be possible in all cases. Dynamic memory allocation stores it's memory on heap, and the static memory allocation stores it's data in the “data segment” of the memory.

```

#include <stdio.h>
#include <stdlib.h>
int main ()
{
//static allocation example using integer array.
int arr[5]; /* static memory allocation memory allocated before execution, the size of array
should be initialized*/
for ( int j = 0; j < 5; j++) //Waste of memory can be occurred.
{
printf("Enter number for Static Array %d: " ,j);
scanf("%d", &arr[j]);
}
printf("\nThe Static Array is: \n");
for ( int j = 0; j < 5; j++)
{
printf("The value of %d is %d\n", j, arr[j]);
}
}

```

```

}

//dynamic allocation example using integer array
int* array;
int n, i;
printf("\n-----\n\nDynamic Allocation\n");
printf("Enter the number of elements: ");
scanf("%d", &n);
array = (int*) malloc(n*sizeof(int)); //memory is allocated during the execution of the
program
//Less Memory space required.
for (i=0; i<n; i++) {
printf("Enter number %d: ", i);
scanf("%d", &array[i]);
}

printf("\nThe Dynamic Array is: \n");

for (i=0; i<n; i++) {
printf("The value of %d is %d\n", i, array[i]);
}
printf("Size= %d\n", i);

system("PAUSE");
return 0;
}

```

Memory Allocation Functions:

Programming language provides several functions for memory allocation and management. These functions can be found in the <stdlib.h> header file.

S. No.	Function & Description
1	void *calloc(int num, int size); This function allocates an array of num elements each of which size in bytes .
2	void free(void *address); This function releases a block of memory specified by address.
3	void *malloc(int num); This function allocates an array of num bytes and leave them uninitialized.
4	void *realloc(void *address, int newsize); This function re-allocates memory extending it upto newsize.

Following are examples of dynamic memory allocation using functions:

1.

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

int main() {

    char name[100];
    char *description;

    strcpy(name, "Zara Ali");

    /* allocate memory dynamically */
    description = malloc(200 * sizeof(char));

    if( description == NULL ) {
        fprintf(stderr, "Error - unable to allocate required memory\n");
    }
    else {
        strcpy( description, "Zara ali a DPS student in class 10th");
    }

    printf("Name = %s\n", name );
    printf("Description: %s\n", description );
}
```

When the above code is compiled and executed, it produces the following result.

```
Name = Zara Ali
Description: Zara ali a DPS student in class 10th
```

Same program can be written using calloc(); only thing is you need to replace malloc with calloc as follows –

```
calloc(200, sizeof(char));
```

So you have complete control and you can pass any size value while allocating memory, unlike arrays where once the size defined, you cannot change it.

2.

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

```
int main() {

    char name[100];
```

```

char *description;

strcpy(name, "Angad");

/* allocate memory dynamically */
description = malloc( 30 * sizeof(char) );

if( description == NULL ) {
    fprintf(stderr, "Error - unable to allocate required memory\n");
}
else {
    strcpy( description, "Angad is a cute Boy.");
}

/* suppose you want to store bigger description */
description = realloc( description, 100 * sizeof(char) );

if( description == NULL ) {
    fprintf(stderr, "Error - unable to allocate required memory\n");
}
else {
    strcat( description, "He is in 1st Class");
}

printf("Name = %s\n", name );
printf("Description: %s\n", description );

/* release memory using free() function */
free(description);
}

```

When the above code is compiled and executed, it produces the following result.

```

Name = Angad
Description: Angad is a cute Boy.He is in 1st Class.

```

Pointers in 'C': A pointer is a variable whose value is the address of another variable, i.e., direct address of the memory location. Like any variable or constant, you must declare a pointer before using it to store any variable address. The general form of a pointer variable declaration is –
type *var-name;

Here, type is the pointer's base type; it must be a valid C data type and var-name is the name of the pointer variable. The asterisk * used to declare a pointer is the same asterisk used for multiplication. Take a look at some of the valid pointer declarations –

```

int *ip; /* pointer to an integer */
double *dp; /* pointer to a double */

```

```
float *fp; /* pointer to a float */
char *ch /* pointer to a character */
```

The actual data type of the value of all pointers, whether integer, float, character, or otherwise, is the same, a long hexadecimal number that represents a memory address. The only difference between pointers of different data types is the data type of the variable or constant that the pointer points to.

The following example shows use of pointer variable –

```
#include <stdio.h>

int main() {

    int var = 20; /* actual variable declaration */
    int *ip; /* pointer variable declaration */

    ip = &var; /* store address of var in pointer variable */

    printf("Address of var variable: %x\n", &var );

    /* address stored in pointer variable */
    printf("Address stored in ip variable: %x\n", ip);

    /* access the value using the pointer */
    printf("Value of *ip variable: %d\n", *ip);

    return 0;
}
```

When the above code is compiled and executed, it produces the following result –

```
Address of var variable: bffd8b3c
Address stored in ip variable: bffd8b3c
Value of *ip variable: 20
```

NULL Pointers

It is always a good practice to assign a NULL value to a pointer variable in case you do not have an exact address to be assigned. This is done at the time of variable declaration. A pointer that is assigned NULL is called a null pointer.

The NULL pointer is a constant with a value of zero defined in several standard libraries. Consider the following program –

```
#include <stdio.h>

int main() {
```

```

int *ptr=NULL;

printf("The value of ptr is : %x\n", ptr );

return 0;
}

```

When the above code is compiled and executed, it produces the following result –

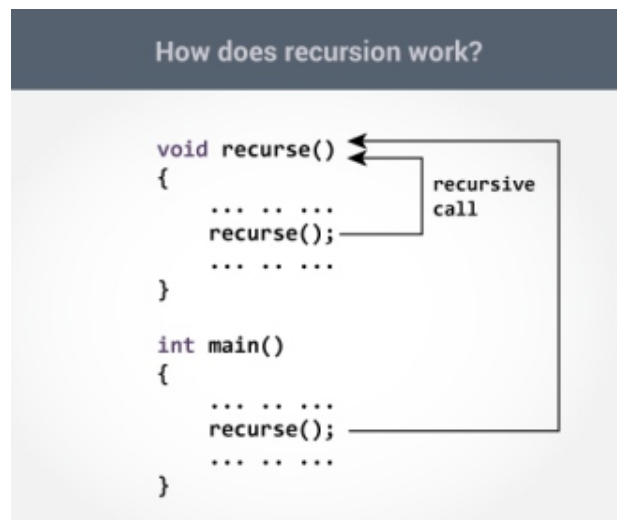
The value of ptr is 0

To check for a null pointer, you can use an 'if' statement as follows –

```

if(ptr) /* succeeds if p is not null */
if(!ptr) /* succeeds if p is null */

```



Recursion in 'C': Recursion is the process of repeating items in a self-similar way. In programming languages, if a program allows you to call a function inside the same function, then it is called a recursive call of the function. Or in other words when a function is calling to itself is known as recursive function.

Recursion works as follows:

```

void recurse()
{
.....
recurse();
.....
}

int main()
{

```



```

.....
recurse();
.....
}

```

Conditions for recursive function:

1. Every function must have a **Base Criteria (Termination condition)** and for that it should not call to itself.
2. Whenever a function is calling to itself it must be closer to the **Base Criteria**.

Following are some examples of recursions-

- (a) Fibonacci Series
- (b) Binomial coefficient
- (c) GCD

(a) Fibonacci Series

Fibonacci series are the numbers in the following integer sequence

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89

the first two numbers in the Fibonacci sequence are 0 and 1, and each subsequent term is the sum of the previous two terms. In mathematical terms, the Nth term of Fibonacci numbers is defined by the recurrence relation:

$\text{fibonacci}(N) = N\text{th term in fibonacci series}$
 $\text{fibonacci}(N) = \text{fibonacci}(N - 1) + \text{fibonacci}(N - 2);$
 whereas, $\text{fibonacci}(0) = 0$ and $\text{fibonacci}(1) = 1$

Below program uses recursion to calculate Nth fibonacci number. To calculate Nth fibonacci number it first calculate (N-1)th and (N-2)th fibonacci number and then add both to get Nth fibonacci number.

For Example : $\text{fibonacci}(4) = \text{fibonacci}(3) + \text{fibonacci}(2);$

C program to print fibonacci series till Nth term using recursion

In below program, we first takes the number of terms of fibonacci series as input from user using scanf function. We are using a user defined recursive function named 'fibonacci' which takes an integer(N) as input and returns the Nth fibonacci number using recursion as discussed above. The recursion will terminate when number of terms are less then 2 because we know the first two terms of fibonacci series are 0 and 1.

```

#include <stdio.h>
#include <conio.h>

int fibonacci(int term);
int main() {
    int terms, counter;
    printf("Enter number of terms in Fibonacci series: ");
    scanf("%d", &terms);
    /*
    Nth term = (N-1)th term + (N-2)th term;

```

```

    */
printf("Fibonacci series till %d terms\n", terms);
for(counter = 0; counter < terms; counter++){
    printf("%d ", fibonacci(counter));
}
getch();
return 0;
}
/*
Function to calculate Nth Fibonacci number
fibonacci(N) = fibonacci(N - 1) + fibonacci(N - 2);
*/
int fibonacci(int term){
    /* Exit condition of recursion*/
    if(term < 2)
        return term;
    return fibonacci(term - 1) + fibonacci(term - 2);
}

```

Program Output

```

Enter number of terms in Fibonacci series: 9
Fibonacci series till 9 terms
0 1 1 2 3 5 8 13 21

```

(b) Binomial Coefficient Program:

```

#include<stdio.h>

int fact(int);
void main()
{
    int n,r,f;
    printf("enter value for n & r\n");
    scanf("%d%d",&n,&r);
    if(n<r)
        printf("invalid input");
    else f=fact(n)/(fact(n-r)*fact(r));
    printf("binomial coefficient=%d",f);
}

int fact(int x)
{
    if(x>1)
        return x*fact(x-1);
    else return 1;
}

```

(c) GCD of Two numbers:

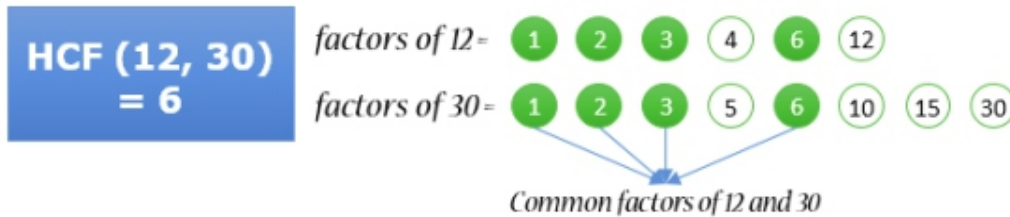
```

Input first number: 10

```

Input second number: 15
Output GCD: 5

Logic to find GCD using recursion



Euclidean algorithm to find GCD:

Begin:

```
function gcd(a, b)
```

```
  If (b = 0) then
```

```
    return a
```

```
  End if
```

```
  Else
```

```
    return gcd(b, a mod b);
```

```
  End if
```

```
End function
```

```
End
```

Program to find GCD using recursion:

```
/**  
 * C program to find GCD (HCF) of two numbers using recursion  
 */
```

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

```
/* Function declaration */
```

```
int gcd(int a, int b);
```

```
int main()
```

```
{
```

```
  int num1, num2, hcf;
```

```
  /* Reads two numbers from user */
```

```
  printf("Enter any two numbers to find GCD: ");
```

```
  scanf("%d%d", &num1, &num2);
```

```
  hcf = gcd(num1, num2);
```

```

printf("GCD of %d and %d = %d\n", num1, num2, hcf);

return 0;
}

/**
 * Recursive approach of euclidean algorithm to find GCD of two numbers
 */
int gcd(int a, int b)
{
    if(b == 0)
        return a;
    else
        return gcd(b, a%b);
}

```

Output:

Enter any two numbers to find GCD: 12

30

GCD of 12 and 30 = 6

Important Points

- An array is defined as finite ordered collection of homogenous data elements which are stored in contiguous memory locations.
- While storing the elements of a 2-D array in memory, these are allocated contiguous memory locations.
- Traversing means accessing the each and every element of array exactly once.
- A pointer is a variable whose value is the address of another variable.

Exercise

Objective type questions.

- Q1. In linear search algorithm worst case occurs when
- The item is somewhere in the middle of the array
 - The item is not in the array at all
 - The item is the last element in the array
 - The item is the last element in the array or is not there at all
- Q2. The complexity of linear search algorithm is
- $O(n)$
 - $O(\log n)$
 - $O(n^2)$
 - $O(n \log n)$
- Q3. Average case occur in linear search algorithm
- When item is somewhere in the middle of the array
 - When item is not in the array at all

- c. When item is the last element in the array
 - d. When item is the last element in the array or is not there at all
- Q4. Finding the location of the element with a given value is:
- a. Traversal
 - b. Search
 - c. Sort
 - d. None of above
- Q5. Which of the following case does not exist in complexity theory
- a. Best case
 - b. Worst case
 - c. Average case
 - d. Null case

Short answer type questions.

- Q1. What is the time complexity of binary search ?
- Q2. What do you mean by Array ?
- Q3. What is string ?
- Q4. What do you mean by pointer ?
- Q5. What is dynamic memory allocation ?

Essay type questions.

- Q1. Explain two Dimensional array with example ?
- Q2. Explain Malloc function in detail ?
- Q3. Which data structure is used to perform recursion ?
- Q4. Why binary search is better then linear search ?
- Q5. Explain character string ?

Answers

Ans1. d
Ans4. b

Ans2. d
Ans4. d

Ans3. a

Chapter 3

Sorting:

Sorting refers to arranging data in a particular format. Sorting algorithm specifies the way to arrange data in a particular order. Most common orders are in numerical or lexicographical order. The importance of sorting lies in the fact that data searching can be optimized to a very high level, if data is stored in a sorted manner. Sorting is also used to represent data in more readable formats. Following are some of the examples of sorting in real-life scenarios –

Telephone Directory – The telephone directory stores the telephone numbers of people sorted by their names, so that the names can be searched easily.

Dictionary – The dictionary stores words in an alphabetical order so that searching of any word becomes easy.

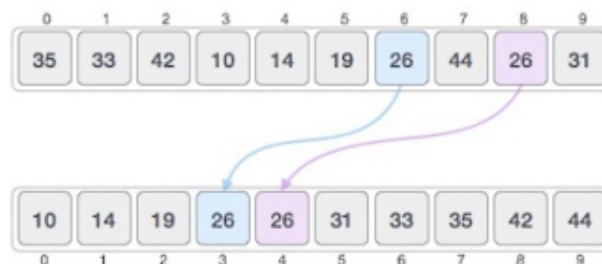
In-place Sorting and Not-in-place Sorting:

Sorting algorithms may require some extra space for comparison and temporary storage of few data elements. In-place sorting algorithms do not require any extra space and sorting is said to happen in-place within the array itself. Bubble sort is an example of in-place sorting.

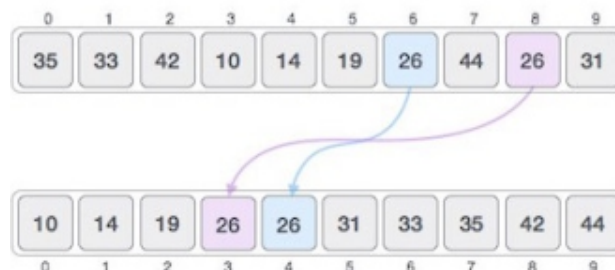
However, in some sorting algorithms, the program requires space which is more than or equal to the elements being sorted. Sorting which uses equal or more space is called not-in-place sorting. Merge-sort is an example of not-in-place sorting.

Stable and Unstable Sorting:

If a sorting algorithm, after sorting the contents, does not change the sequence of similar content in which they appear, it is called stable sorting.



If a sorting algorithm, after sorting the contents, changes the sequence of similar content in which they appear, it is called unstable sorting.



Stability of an algorithm matters when we wish to maintain the sequence of original elements, like in a tuple for example.

Adaptive and Non-Adaptive Sorting Algorithm:

A sorting algorithm is said to be adaptive, if it takes advantage of already 'sorted' elements in the list that is to be sorted. That is, while sorting if the source list has some element already sorted, adaptive algorithms will take this into account and will try not to re-order them.

A non-adaptive algorithm is one which does not take into account the elements which are already sorted. They try to force every single element to be re-ordered to confirm their sortedness.

Important Terms

Some terms are generally coined while discussing sorting techniques, here is a brief introduction to them –

Increasing Order:

A sequence of values is said to be in increasing order, if the successive element is greater than the previous one. For example, 1, 3, 4, 6, 8, 9 are in increasing order, as every next element is greater than the previous element.

Decreasing Order:

A sequence of values is said to be in decreasing order, if the successive element is less than the current one. For example, 9, 8, 6, 4, 3, 1 are in decreasing order, as every next element is less than the previous element.

Non-Increasing Order:

A sequence of values is said to be in non-increasing order, if the successive element is less than or equal to its previous element in the sequence. This order occurs when the sequence contains duplicate values. For example, 9, 8, 6, 3, 3, 1 are in non-increasing order, as every next element is less than or equal to (in case of 3) but not greater than any previous element.

Non-Decreasing Order:

A sequence of values is said to be in non-decreasing order, if the successive element is greater than or equal to its previous element in the sequence. This order occurs when the sequence contains duplicate values. For example, 1, 3, 3, 6, 8, 9 are in non-decreasing order, as every next element is greater than or equal to (in case of 3) but not less than the previous one.

Bubble Sort:

Bubble sort is a simple sorting algorithm. This sorting algorithm is comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order. This algorithm is not suitable for large data sets as its average and worst case complexity are of $O(n^2)$ where n is the number of items.

How Bubble Sort Works?

We take an unsorted array for our example. Bubble sort takes $O(n^2)$ time so we're keeping it short and precise.



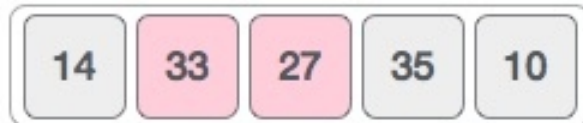
Bubble sort starts with very first two elements, comparing them to check which one is greater.



In this case, value 33 is greater than 14, so it is already in sorted locations. Next, we compare 33 with 27.



We find that 27 is smaller than 33 and these two values must be swapped.



The new array should look like this –



Next we compare 33 and 35. We find that both are in already sorted positions.



Then we move to the next two values, 35 and 10.



We know then that 10 is smaller 35. Hence they are not sorted.



We swap these values. We find that we have reached the end of the array. After one iteration, the array should look like this –



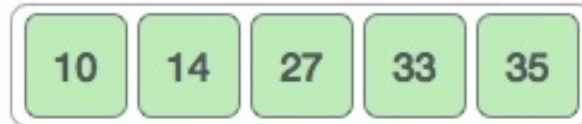
To be precise, we are now showing how an array should look like after each iteration. After the second iteration, it should look like this –



Notice that after each iteration, at least one value moves at the end.



And when there's no swap required, bubble sort learns that an array is completely sorted.



Algorithm:

We assume list is an array of n elements. We further assume that swap function swaps the values of the given array elements.

```
begin BubbleSort(list)
```

```
  for all elements of list
    if list[i] > list[i+1]
      swap(list[i], list[i+1])
    end if
  end for
```

```
  return list
```

```
end BubbleSort
```

C program for Bubble Sort:

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

```
#define MAX 10
```

```
int list[MAX] = {1,8,4,6,0,3,5,2,7,9};
```

```

void display() {
    int i;
    printf("[");

    // navigate through all items
    for(i = 0; i < MAX; i++) {
        printf("%d ",list[i]);
    }

    printf("]\n");
}

void bubbleSort() {
    int temp;
    int i,j;

    bool swapped = false;

    // loop through all numbers
    for(i = 0; i < MAX-1; i++) {
        swapped = false;

        // loop through numbers falling ahead
        for(j = 0; j < MAX-1-i; j++) {
            printf("  Items compared: [ %d, %d ] ", list[j],list[j+1]);

            // check if next number is lesser than current no
            // swap the numbers.
            // (Bubble up the highest number)

            if(list[j] > list[j+1]) {
                temp = list[j];
                list[j] = list[j+1];
                list[j+1] = temp;

                swapped = true;
                printf(" => swapped [%d, %d]\n",list[j],list[j+1]);
            }else {
                printf(" => not swapped\n");
            }
        }

        // if no number was swapped that means
        // array is sorted now, break the loop.
        if(!swapped) {
            break;
        }
    }
}

```

```

    }

    printf("Iteration %d#: ",(i+1));
    display();
}

}

main() {
    printf("Input Array: ");
    display();
    printf("\n");

    bubbleSort();
    printf("\nOutput Array: ");
    display();
}

```

If we compile and run the above program, it will produce the following result –

Input Array: [1 8 4 6 0 3 5 2 7 9]

```

Items compared: [ 1, 8 ] => not swapped
Items compared: [ 8, 4 ] => swapped [4, 8]
Items compared: [ 8, 6 ] => swapped [6, 8]
Items compared: [ 8, 0 ] => swapped [0, 8]
Items compared: [ 8, 3 ] => swapped [3, 8]
Items compared: [ 8, 5 ] => swapped [5, 8]
Items compared: [ 8, 2 ] => swapped [2, 8]
Items compared: [ 8, 7 ] => swapped [7, 8]
Items compared: [ 8, 9 ] => not swapped

```

Iteration 1#: [1 4 6 0 3 5 2 7 8 9]

```

Items compared: [ 1, 4 ] => not swapped
Items compared: [ 4, 6 ] => not swapped
Items compared: [ 6, 0 ] => swapped [0, 6]
Items compared: [ 6, 3 ] => swapped [3, 6]
Items compared: [ 6, 5 ] => swapped [5, 6]
Items compared: [ 6, 2 ] => swapped [2, 6]
Items compared: [ 6, 7 ] => not swapped
Items compared: [ 7, 8 ] => not swapped

```

Iteration 2#: [1 4 0 3 5 2 6 7 8 9]

```

Items compared: [ 1, 4 ] => not swapped
Items compared: [ 4, 0 ] => swapped [0, 4]
Items compared: [ 4, 3 ] => swapped [3, 4]
Items compared: [ 4, 5 ] => not swapped
Items compared: [ 5, 2 ] => swapped [2, 5]
Items compared: [ 5, 6 ] => not swapped
Items compared: [ 6, 7 ] => not swapped

```

Iteration 3#: [1 0 3 4 2 5 6 7 8 9]
 Items compared: [1, 0] => swapped [0, 1]
 Items compared: [1, 3] => not swapped
 Items compared: [3, 4] => not swapped
 Items compared: [4, 2] => swapped [2, 4]
 Items compared: [4, 5] => not swapped
 Items compared: [5, 6] => not swapped
 Iteration 4#: [0 1 3 2 4 5 6 7 8 9]
 Items compared: [0, 1] => not swapped
 Items compared: [1, 3] => not swapped
 Items compared: [3, 2] => swapped [2, 3]
 Items compared: [3, 4] => not swapped
 Items compared: [4, 5] => not swapped
 Iteration 5#: [0 1 2 3 4 5 6 7 8 9]
 Items compared: [0, 1] => not swapped
 Items compared: [1, 2] => not swapped
 Items compared: [2, 3] => not swapped
 Items compared: [3, 4] => not swapped

Output Array: [0 1 2 3 4 5 6 7 8 9]

Selection Sort: Selection sort is a simple sorting algorithm. This sorting algorithm is an in-place comparison-based algorithm in which the list is divided into two parts, the sorted part at the left end and the unsorted part at the right end. Initially, the sorted part is empty and the unsorted part is the entire list. The smallest element is selected from the unsorted array and swapped with the leftmost element, and that element becomes a part of the sorted array. This process continues moving unsorted array boundary by one element to the right. This algorithm is not suitable for large data sets as its average and worst case complexities are of $O(n^2)$, where n is the number of items.

How Selection Sort Works?

Consider the following depicted array as an example.



For the first position in the sorted list, the whole list is scanned sequentially. The first position where 14 is stored presently, we search the whole list and find that 10 is the lowest value.



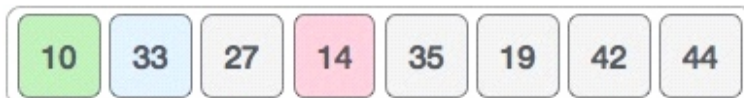
So we replace 14 with 10. After one iteration 10, which happens to be the minimum value in the list, appears in the first position of the sorted list.



For the second position, where 33 is residing, we start scanning the rest of the list in a linear manner.



We find that 14 is the second lowest value in the list and it should appear at the second place. We swap these values.



After two iterations, two least values are positioned at the beginning in a sorted manner.



The same process is applied to the rest of the items in the array.

Following is a pictorial depiction of the entire sorting process –



Algorithm:

- Step 1 – Set MIN to location 0
- Step 2 – Search the minimum element in the list
- Step 3 – Swap with value at location MIN
- Step 4 – Increment MIN to point to next element
- Step 5 – Repeat until list is sorted

C program for Selection Sort:

```
#include <stdio.h>
#include <stdbool.h>
#define MAX 7

int intArray[MAX] = {4,6,3,2,1,9,7};

void printline(int count) {
    int i;

    for(i = 0; i < count-1; i++) {
        printf("=");
    }

    printf("\n");
}

void display() {
    int i;
    printf("[");

    // navigate through all items
    for(i = 0; i < MAX; i++) {
        printf("%d ", intArray[i]);
    }

    printf("]\n");
}

void selectionSort() {

    int indexMin, i, j;

    // loop through all numbers
    for(i = 0; i < MAX-1; i++) {

        // set current element as minimum
        indexMin = i;

        // check the element to be minimum
        for(j = i+1; j < MAX; j++) {
```

```

        if(intArray[j] < intArray[indexMin]) {
            indexMin = j;
        }
    }

    if(indexMin != i) {
        printf("Items swapped: [ %d, %d ]\n" , intArray[i], intArray[indexMin]);

        // swap the numbers
        int temp = intArray[indexMin];
        intArray[indexMin] = intArray[i];
        intArray[i] = temp;
    }

    printf("Iteration %d#:",(i+1));
    display();
}

main() {
    printf("Input Array: ");
    display();
    printline(50);
    selectionSort();
    printf("Output Array: ");
    display();
    printline(50);
}

```

If we compile and run the above program, it will produce the following result –
Input Array: [4 6 3 2 1 9 7]

```

=====
Items swapped: [ 4, 1 ]
Iteration 1#:[1 6 3 2 4 9 7 ]
Items swapped: [ 6, 2 ]
Iteration 2#:[1 2 3 6 4 9 7 ]
Iteration 3#:[1 2 3 6 4 9 7 ]
Items swapped: [ 6, 4 ]
Iteration 4#:[1 2 3 4 6 9 7 ]
Iteration 5#:[1 2 3 4 6 9 7 ]
Items swapped: [ 9, 7 ]
Iteration 6#:[1 2 3 4 6 7 9 ]
Output Array: [1 2 3 4 6 7 9 ]

```

Merge Sort: Merge sort is a sorting technique based on divide and conquer technique. With worst-case time complexity being $O(n \log n)$, it is one of the most respected algorithms. Merge sort first divides the array into equal halves and then combines them in a sorted manner.

How Merge Sort Works?

To understand merge sort, we take an unsorted array as the following –



We know that merge sort first divides the whole array iteratively into equal halves unless the atomic values are achieved. We see here that an array of 8 items is divided into two arrays of size 4.



This does not change the sequence of appearance of items in the original. Now we divide these two arrays into halves.



We further divide these arrays and we achieve atomic value which can no more be divided



Atul starts from here

Now, we combine them in exactly the same manner as they were broken down. Please note the color codes given to these lists.

We first compare the element for each list and then combine them into another list in a sorted manner. We see that 14 and 33 are in sorted positions. We compare 27 and 10 and in the target list of 2 values we put 10 first, followed by 27. We change the order of 19 and 35 whereas 42 and 44 are placed sequentially.



In the next iteration of the combining phase, we compare lists of two data values, and merge them into a list of found data values placing all in a sorted order.



After the final merging, the list should look like this –



Now we should learn some programming aspects of merge sorting.

Algorithm:

Merge sort keeps on dividing the list into equal halves until it can no more be divided. By definition, if it is only one element in the list, it is sorted. Then, merge sort combines the smaller sorted lists keeping the new list sorted too.

Step 1 – if it is only one element in the list it is already sorted, return.

Step 2 – divide the list recursively into two halves until it can no more be divided.

Step 3 – merge the smaller lists into new list in sorted order.

C Program for Merge Sort:

```
#include <stdio.h>
#define max 10

int a[10] = {10, 14, 19, 26, 27, 31, 33, 35, 42, 44 };
int b[10];

void merging(int low, int mid, int high) {
    int l1, l2, i;

    for(l1 = low, l2 = mid + 1, i = low; l1 <= mid && l2 <= high; i++) {
        if(a[l1] <= a[l2])
            b[i] = a[l1++];
        else
            b[i] = a[l2++];
    }

    while(l1 <= mid)
        b[i++] = a[l1++];

    while(l2 <= high)
        b[i++] = a[l2++];

    for(i = low; i <= high; i++)
        a[i] = b[i];
}

void sort(int low, int high) {
    int mid;

    if(low < high) {
```

```

        mid = (low + high) / 2;
        sort(low, mid);
        sort(mid+1, high);
        merging(low, mid, high);
    }else {
        return;
    }
}

int main() {
    int i;

    printf("List before sorting\n");

    for(i = 0; i <= max; i++)
        printf("%d ", a[i]);

    sort(0, max);

    printf("\nList after sorting\n");

    for(i = 0; i <= max; i++)
        printf("%d ", a[i]);
}

```

If we compile and run the above program, it will produce the following result –

```

Output
List before sorting
10 14 19 26 27 31 33 35 42 44 0
List after sorting
0 10 14 19 26 27 31 33 35 42 44

```

Insertion Sort: This is an in-place comparison-based sorting algorithm. Here, a sub-list is maintained which is always sorted. For example, the lower part of an array is maintained to be sorted. An element which is to be inserted in this sorted sub-list, has to find its appropriate place and then it has to be inserted there. Hence the name, insertion sort.

The array is searched sequentially and unsorted items are moved and inserted into the sorted sub-list (in the same array). This algorithm is not suitable for large data sets as its average and worst case complexity are of $O(n^2)$, where n is the number of items.

How Insertion Sort Works?

We take an unsorted array for our example.



Insertion sort compares the first two elements.



It finds that both 14 and 33 are already in ascending order. For now, 14 is in sorted sub-list.



Insertion sort moves ahead and compares 33 with 27.



And finds that 33 is not in the correct position.



It swaps 33 with 27. It also checks with all the elements of sorted sub-list. Here we see that the sorted sub-list has only one element 14, and 27 is greater than 14. Hence, the sorted sub-list remains sorted after swapping.



By now we have 14 and 27 in the sorted sub-list. Next, it compares 33 with 10.



These values are not in a sorted order.



So we swap them.



However, swapping makes 27 and 10 unsorted.



Hence, we swap them too.



Again we find 14 and 10 in an unsorted order.



We swap them again. By the end of third iteration, we have a sorted sub-list of 4 items.



This process goes on until all the unsorted values are covered in a sorted sub-list.

Algorithm:

Now we have a bigger picture of how this sorting technique works, so we can derive simple steps by which we can achieve insertion sort.

- Step 1 – If it is the first element, it is already sorted. return 1
- Step 2 – Pick next element
- Step 3 – Compare with all elements in the sorted sub-list
- Step 4 – Shift all the elements in the sorted sub-list that is greater than the value to be sorted
- Step 5 – Insert the value
- Step 6 – Repeat until list is sorted

C Program for Insertion Sort:

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 7

int intArray[MAX] = {4,6,3,2,1,9,7};

void printline(int count) {
    int i;
```

```

for(i = 0;i <count-1;i++) {
    printf("=");
}

printf("\n");
}

void display() {
    int i;
    printf("[");

    // navigate through all items
    for(i = 0;i<MAX;i++) {
        printf("%d ",intArray[i]);
    }

    printf("]\n");
}

void insertionSort() {

    int valueToInsert;
    int holePosition;
    int i;

    // loop through all numbers
    for(i = 1; i < MAX; i++) {

        // select a value to be inserted.
        valueToInsert = intArray[i];

        // select the hole position where number is to be inserted
        holePosition = i;

        // check if previous no. is larger than value to be inserted
        while (holePosition > 0 && intArray[holePosition-1] > valueToInsert) {
            intArray[holePosition] = intArray[holePosition-1];
            holePosition--;
            printf(" item moved : %d\n" , intArray[holePosition]);
        }

        if(holePosition != i) {
            printf(" item inserted : %d, at position : %d\n" , valueToInsert,holePosition);
            // insert the number at hole position
            intArray[holePosition] = valueToInsert;
        }

        printf("Iteration %d#:",i);

```

```

        display();
    }
}

main() {
    printf("Input Array: ");
    display();
    printline(50);
    insertionSort();
    printf("Output Array: ");
    display();
    printline(50);
}

```

If we compile and run the above program, it will produce the following result –

```

Input Array: [4 6 3 2 1 9 7 ]
=====
Iteration 1#[4 6 3 2 1 9 7 ]
item moved : 6
item moved : 4
item inserted : 3, at position : 0
Iteration 2#[3 4 6 2 1 9 7 ]
item moved : 6
item moved : 4
item moved : 3
item inserted : 2, at position : 0
Iteration 3#[2 3 4 6 1 9 7 ]
item moved : 6
item moved : 4
item moved : 3
item moved : 2
item inserted : 1, at position : 0
Iteration 4#[1 2 3 4 6 9 7 ]
Iteration 5#[1 2 3 4 6 9 7 ]
item moved : 9
item inserted : 7, at position : 5
Iteration 6#[1 2 3 4 6 7 9 ]
Output Array: [1 2 3 4 6 7 9 ]

```

Quick Sort: Quick sort is a highly efficient sorting algorithm and is based on partitioning of array of data into smaller arrays. A large array is partitioned into two arrays one of which holds values smaller than the specified value, say pivot, based on which the partition is made and another array holds values greater than the pivot value. Quick sort partitions an array and then calls itself recursively twice to sort the two resulting subarrays. This algorithm is quite efficient for large-sized data sets as its average and worst case complexity are of $O(n \log n)$, where n is the number of items.

Partition in Quick Sort:

Following example explains how to find the pivot value in an array.

The pivot value divides the list into two parts. And recursively, we find the pivot for each sub-lists until all lists contains only one element.

A[6]	A[7]	A[8]	A[9]	A[10]	A[11]	A[12]
98	84	65	108	60	96	72
72	84	65	108	60	96	98
72	84	65	98	60	96	108
72	84	65	96	60	98	108

Pivot

- Step 1 – Choose the highest index value as pivot
- Step 2 – Take two variables to point left and right of the list excluding pivot
- Step 3 – left points to the low index
- Step 4 – right points to the high index
- Step 5 – while value at left is less than pivot move right
- Step 6 – while value at right is greater than pivot move left
- Step 7 – if both step 5 and step 6 does not match swap left and right
- Step 8 – if left \geq right, the point where they met is new pivot

Quick Sort Algorithm:

Using pivot algorithm recursively, we end up with smaller possible partitions. Each partition is then processed for quick sort. We define recursive algorithm for quicksort as follows –

- Step 1 – Make the right-most index value pivot
- Step 2 – partition the array using pivot value
- Step 3 – quicksort left partition recursively
- Step 4 – quicksort right partition recursively

C Program for Quick Sort:

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 7

int intArray[MAX] = {4,6,3,2,1,9,7};

void printline(int count) {
```

```

int i;

for(i = 0;i <count-1;i++) {
    printf("=");
}

printf("\n");
}

void display() {
    int i;
    printf("[");

    // navigate through all items
    for(i = 0;i<MAX;i++) {
        printf("%d ",intArray[i]);
    }

    printf("]\n");
}

void swap(int num1, int num2) {
    int temp = intArray[num1];
    intArray[num1] = intArray[num2];
    intArray[num2] = temp;
}

int partition(int left, int right, int pivot) {
    int leftPointer = left -1;
    int rightPointer = right;

    while(true) {
        while(intArray[++leftPointer] < pivot) {
            //do nothing
        }

        while(rightPointer > 0 && intArray[--rightPointer] > pivot) {
            //do nothing
        }

        if(leftPointer >= rightPointer) {
            break;
        }else {
            printf(" item swapped :%d,%d\n", intArray[leftPointer],intArray[rightPointer]);
            swap(leftPointer,rightPointer);
        }
    }
}

```



```

printf(" pivot swapped :%d,%d\n", intArray[leftPointer],intArray[right]);
swap(leftPointer,right);
printf("Updated Array: ");
display();
return leftPointer;
}

void quickSort(int left, int right) {
if(right-left <= 0) {
return;
}else {
int pivot = intArray[right];
int partitionPoint = partition(left, right, pivot);
quickSort(left,partitionPoint-1);
quickSort(partitionPoint+1,right);
}
}

main() {
printf("Input Array: ");
display();
printline(50);
quickSort(0,MAX-1);
printf("Output Array: ");
display();
printline(50);
}

```

If we compile and run the above program, it will produce the following result –
Output

Input Array: [4 6 3 2 1 9 7]

```

=====
pivot swapped :9,7
Updated Array: [4 6 3 2 1 7 9 ]
pivot swapped :4,1
Updated Array: [1 6 3 2 4 7 9 ]
item swapped :6,2
pivot swapped :6,4
Updated Array: [1 2 3 4 6 7 9 ]
pivot swapped :3,3
Updated Array: [1 2 3 4 6 7 9 ]
Output Array: [1 2 3 4 6 7 9 ]

```

Important Points

- Sorting algorithm specifies the way to arrange data in a particular order. Most common orders are in numerical or lexicographical order.
- Bubble sort is a simple sorting algorithm. This sorting algorithm is comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order.
- Merge sort is a sorting technique based on divide and conquer technique. With worst-case time complexity being $O(n \log n)$, it is one of the most respected algorithms.
- Quick sort is a highly efficient sorting algorithm and is based on partitioning of array of data into smaller arrays.

Exercise

Objective type questions.

- Q1. The complexity of Bubble sort algorithm is
- a. $O(n)$
 - b. $O(\log n)$
 - c. $O(n^2)$
 - d. $O(n \log n)$
- Q2. The complexity of merge sort algorithm is
- a. $O(n)$
 - b. $O(\log n)$
 - c. $O(n^2)$
 - d. $O(n \log n)$
- Q3. The complexity of selection sort algorithm is
- a. $O(n)$
 - b. $O(\log n)$
 - c. $O(n^2)$
 - d. $O(n \log n)$
- Q4. Which is the good sorting algorithm.
- a. Selection sort
 - b. Insertion sort
 - c. Quick sort
 - d. None
- Q5. The complexity of quick sort algorithm is.
- a. $O(n)$
 - b. $O(\log n)$
 - c. $O(n^2)$
 - d. $O(n \log n)$

Short answer type questions.

- Q1. What is sorting ?
- Q2. What is Stable sort ?
- Q3. What is in-place sorting ?
- Q4. What is worst case running time of Quick sort ?

Q5. When is the worst case for quick sort ?

Essay type questions.

Q1. Explain merge sort in detail ?

Q2. Which is the best sorting algorithm and Why ?

Q3. Explain Quick sort ?

Q4. Differentiate between selection and insertion sort ?

Q5. What is the difference between stable and unstable sorting ?

Answers

Ans1. C

Ans4. c

Ans2. d

Ans5. d

Ans3. c

Chapter 4

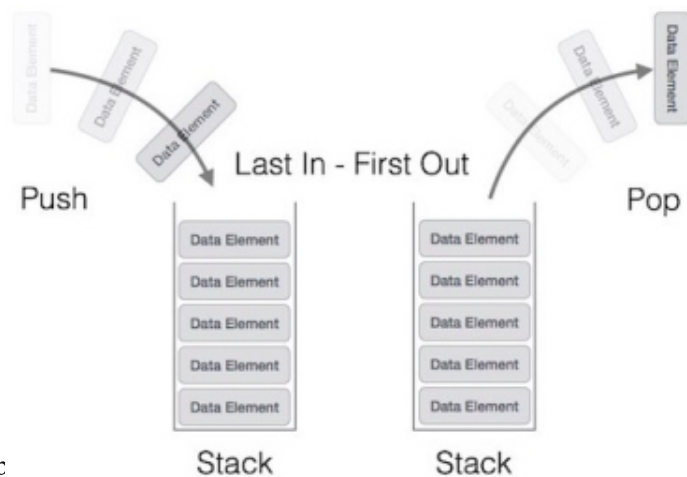
Stack:

A stack is an Abstract Data Type (ADT), commonly used in most programming languages. It is named stack as it behaves like a real-world stack, for example – a deck of cards or a pile of plates, etc.



A real-world stack allows operations at one end only. For example, we can place or remove a card or plate from the top of the stack only. Likewise, Stack ADT allows all data operations at one end only. At any given time, we can only access the top element of a stack. This feature makes it LIFO data structure. LIFO stands for Last-in-first-out. Here, the element which is placed (inserted or added) last, is accessed first. In stack terminology, insertion operation is called PUSH operation and removal operation is called POP operation.

Stack Presentation: The following diagram depicts a stack and its operations –



A stack can be implemented as a linked List. Stack can either be a fixed size one or it may have a sense of dynamic resizing. Here, we are going to implement stack using arrays, which makes it a fixed size stack implementation.

Basic Operations:

Stack operations may involve initializing the stack, using it and then de-initializing it. Apart from these basic stuffs, a stack is used for the following two primary operations –

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

pop() – Removing (accessing) an element from the stack.

To use a stack efficiently, we need to check the status of stack as well. For the same purpose, the following functionality is added to stacks –

peek() – get the top data element of the stack, without removing it.

isFull() – check if stack is full.

isEmpty() – check if stack is empty.

At all times, we maintain a pointer to the last Pushed data on the stack. As this pointer always represents the top of the stack, hence named TOP. The TOP pointer provides top value of the stack without actually removing it.

Procedures to support stack functions –

peek():

Algorithm of peek() function –

```
begin procedure peek
```

```
    return stack[top]
```

```
end procedure
```

Implementation of peek() function in C programming language –

```
int peek() {  
    return stack[top];  
}
```

isfull():

Algorithm of isfull() function –

```
begin procedure isfull
```

```
    if top equals to MAXSIZE
```

```
        return true
```

```
    else
```

```
        return false
```

```
    endif
```

```
end procedure
```

Implementation of isfull() function in C programming language –

```
bool isfull() {  
    if(top == MAXSIZE)  
        return true;
```

```

else
    return false;
}

```

isempty():

Algorithm of isempty() function –

begin procedure isempty

```

if top less than 1
    return true
else
    return false
endif

```

end procedure

Implementation of isempty() function in C programming language is slightly different. We initialize top at -1, as the index in array starts from 0. So we check if the top is below zero or -1 to determine if the stack is empty. Here's the code –

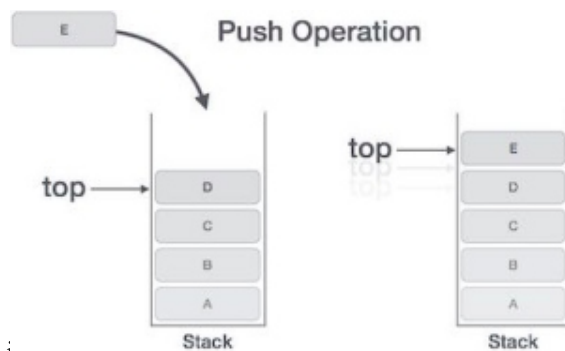
```

bool isempty() {
    if(top == -1)
        return true;
    else
        return false;
}

```

Push Operation: The process of putting a new data element onto stack is known as a Push Operation. Push operation involves a series of steps –

- Step 1 – Checks if the stack is full.
- Step 2 – If the stack is full, produces an error and exit.
- Step 3 – If the stack is not full, increments top to point next empty space.
- Step 4 – Adds data element to the stack location, where top is pointing.
- Step 5 – Returns success.



If the linked list is used to implement the stack, then in step 3, we need to allocate space dynamically.

Algorithm for Push Operation:

```
begin procedure push: stack, data
  if stack is full
    return null
  endif

  top ← top + 1

  stack[top] ← data
```

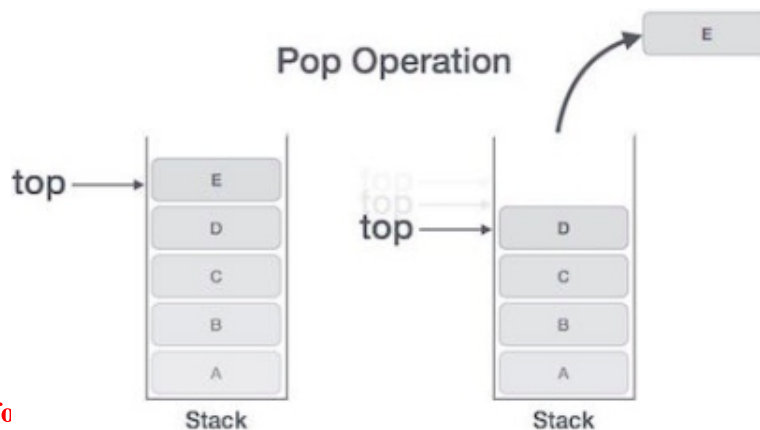
end procedure
Implementation of algorithm in C –

```
void push(int data) {
  if(!isFull()) {
    top = top + 1;
    stack[top] = data;
  } else {
    printf("Could not insert data, Stack is full.\n");
  }
}
```

Pop Operation: Accessing the content while removing it from the stack, is known as a Pop Operation. In an array implementation of pop() operation, the data element is not actually removed, instead top is decremented to a lower position in the stack to point to the next value. But in linked-list implementation, pop() actually removes data element and deallocates memory space.

A Pop operation may involve the following steps –

- Step 1 – Checks if the stack is empty.
- Step 2 – If the stack is empty, produces an error and exit.
- Step 3 – If the stack is not empty, accesses the data element at which top is pointing.
- Step 4 – Decreases the value of top by 1.
- Step 5 – Returns success.



Algorithm fo

```
begin procedure pop: stack
```

```
  if stack is empty  
    return null  
  endif
```

```
  data ← stack[top]
```

```
  top ← top - 1
```

```
  return data
```

```
end procedure
```

Implementation of algorithm in C –

```
int pop(int data) {  
  if(!isempty()) {  
    data = stack[top];  
    top = top - 1;  
    return data;  
  } else {  
    printf("Could not retrieve data, Stack is empty.\n");  
  }  
}
```

Application of Stack: Stack can be used for following purpose-

- (a) Arithmetic expression evaluation
- (b) Backtracking
- (c) Memory Management

(a) Arithmetic expression evaluation: The way to write arithmetic expression is known as a notation. An arithmetic expression can be written in three different but equivalent notations, i.e., without changing the essence or output of an expression. These notations are –

- Infix Notation
- Prefix (Polish) Notation
- Postfix (Reverse-Polish) Notation

These notations are named as how they use operator in expression.

Infix Notation

We write expression in infix notation, e.g. $a - b + c$, where operators are used in-between operands. It is easy for us humans to read, write, and speak in infix notation but the same does not go well with computing devices. An algorithm to process infix notation could be difficult and costly in terms of time and space consumption.

Prefix Notation

In this notation, operator is prefixed to operands, i.e. operator is written ahead of operands. For example, $+ab$. This is equivalent to its infix notation $a + b$. Prefix notation is also known as Polish Notation.

Postfix Notation

This notation style is known as Reversed Polish Notation. In this notation style, the operator is postfixed to the operands i.e., the operator is written after the operands. For example, $ab+$. This is equivalent to its infix notation $a + b$.

So the stack is used for conversion an expression from one notation to another notation.

(b) Backtracking: Backtracking is used in algorithms in which there are steps along some path (state) from some starting point to some goal.

Find your way through a maze.

Find a path from one point in a graph (roadmap) to another point.

In all of these cases, there are choices to be made among a number of options. We need some way to remember these decision points in case we want/need to come back and try the other alternative

Consider the maze. At a point where a choice is made, we may discover that the choice leads to a dead-end. We want to retrace back to that decision point and then try the other (next) alternative.

Again, stacks can be used as part of the solution. Recursion is another, typically more favored, solution, which is actually implemented by a stack.

(c) Memory Management: Any modern computer environment uses a stack as the primary memory management model for a running program. Whether it's native code (x86, Sun, VAX) or JVM, a stack is at the center of the run-time environment for Java, C++, Ada, FORTRAN, etc.

Queue:

Queue is an abstract data structure, somewhat similar to Stacks. Unlike stacks, a queue is open at both its ends. One end is always used to insert data (enqueue) and the other is used to remove data (dequeue). Queue follows First-In-First-Out methodology, i.e., the data item stored first will be accessed first.



first, exits first. More real-world examples can be seen as queues at the ticket windows of bus-stops and others.

Queue presentation:

As we now understand that in queue, we access both ends for different reasons. The following diagram given below tries to explain queue representation as data structure –



As in stacks, a queue can also be implemented using arrays, linked lists, pointers and Structures. For the sake of simplicity, we shall implement queues using one-dimensional array.

Basic Operations:

Queue operations may involve initializing or defining the queue, utilizing it, and then completely erasing it from the memory. Here we shall try to understand the basic operations associated with queues –

enqueue() – add (store) an item to the queue.

dequeue() – remove (access) an item from the queue.

Few more functions are required to make the above-mentioned queue operation efficient. These are –

peek() – Gets the element at the front of the queue without removing it.

isfull() – Checks if the queue is full.

isempty() – Checks if the queue is empty.

supportive functions of a queue –

peek()

The algorithm of peek() function is as follows –

```
begin procedure peek
```

```
    return queue[front]
```

```
end procedure
```

Implementation of peek() function in C programming language –

Example

```
int peek() {  
    return queue[front];  
}
```

isfull():

As we are using single dimension array to implement queue, we just check for the rear pointer to reach at MAXSIZE to determine that the queue is full. In case we maintain the queue in a circular linked-list, the algorithm will differ. Algorithm of isfull() function –

```
begin procedure isfull
```

```
    if rear equals to MAXSIZE  
        return true  
    else  
        return false  
    endif
```

```
end procedure
```

Implementation of isfull() function in C programming language –

```
bool isfull() {  
    if(rear == MAXSIZE - 1)  
        return true;  
    else  
        return false;  
}
```

isempty():

Algorithm of isempty() function –

```
begin procedure isempty
```

```
    if front is less than MIN or front is greater than rear  
        return true  
    else  
        return false  
    endif
```

end procedure

If the value of front is less than MIN or 0, it tells that the queue is not yet initialized, hence empty.

Here's the C programming code –

```
bool isempty() {  
    if(front < 0 || front > rear)  
        return true;  
    else  
        return false;  
}
```

Enqueue Operation:

Queues maintain two data pointers, front and rear. Therefore, its operations are comparatively difficult to implement than that of stacks.

The following steps should be taken to enqueue (insert) data into a queue –

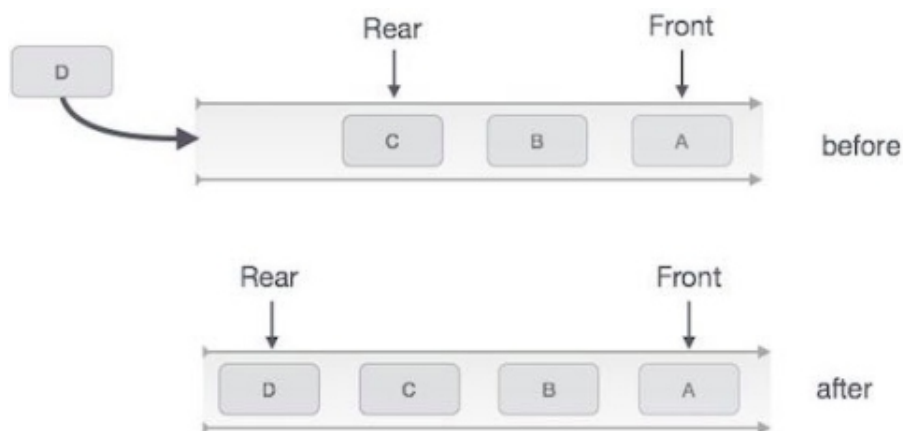
Step 1 – Check if the queue is full.

Step 2 – If the queue is full, produce overflow error and exit.

Step 3 – If the queue is not full, increment rear pointer to point the next empty space.

Step 4 – Add data element to the queue location, where the rear is pointing.

Step 5 – Return success.



Algor

Queue Enqueue

```

procedure enqueue(data)
  if queue is full
    return overflow
  endif
  rear ← rear + 1
  queue[rear] ← data

  return true
end procedure

```

Implementation of enqueue() in C programming language –

```

int enqueue(int data)
  if(isfull())
    return 0;

  rear = rear + 1;
  queue[rear] = data;

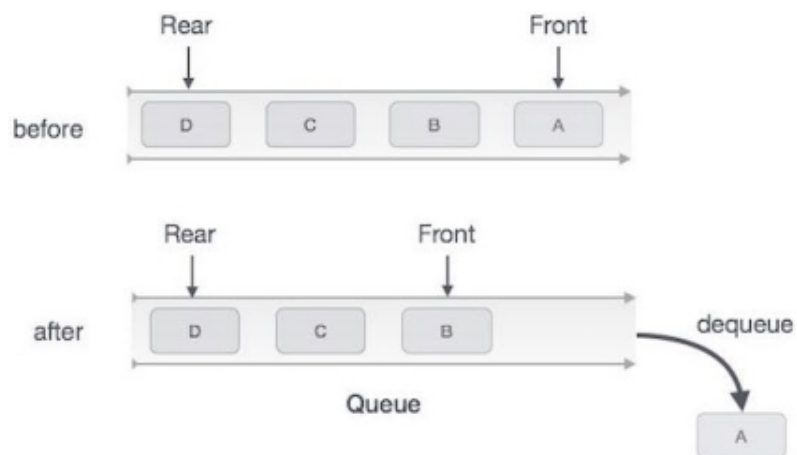
  return 1;
end procedure

```

Dequeue Operation:

Accessing data from the queue is a process of two tasks – access the data where front is pointing and remove the data after access. The following steps are taken to perform dequeue operation –

- Step 1 – Check if the queue is empty.
- Step 2 – If the queue is empty, produce underflow error and exit.
- Step 3 – If the queue is not empty, access the data where front is pointing.
- Step 3 – Increment front pointer to point to the next available data element.
- Step 5 – Return success.



Algorithm

Queue Dequeue

```

procedure dequeue
  if queue is empty
    return underflow
  end if

  data = queue[front]
  front ← front + 1

  return true
end procedure

```

Implementation of dequeue() in C programming language –

```

int dequeue() {

  if(isempty())
    return 0;

  int data = queue[front];
  front = front + 1;

  return data;
}

```

Important Points

- A stack is an Abstract Data Type (ADT), commonly used in most programming languages.
- A stack can be implemented by means of Array, Structure, Pointer, and Linked List. Stack can either be a fixed size one or it may have a sense of dynamic resizing.
- Queue is an abstract data structure, somewhat similar to Stacks. Unlike stacks, a queue is open at both its ends.
- Any modern computer environment uses a stack as the primary memory management model for a running program.

Exercise

Objective type questions.

- Q1. Which of the following name does not relate to stacks.
- FIFO lists
 - LIFO list
 - Pop
 - Push-down lists
- Q2. The term "push" and "pop" is related to the
- array

- b. lists
- c. stacks
- d. all of above

Q3. A data structure where elements can be added or removed at either end but not in the middle.

- a. Linked lists
- b. Stacks
- c. Queues
- d. Dequeue

Q4. The data structure required for Breadth First Traversal on a graph is.

- a. Stack
- b. Array
- c. Queue
- d. Tree

Q5. A queue is a.

- a. FIFO (First In First Out) list
- b. LIFO (Last In First Out) list.
- c. Ordered array
- d. Linear tree

Short answer type questions.

Q1. Define stack ?

Q2. Define Queue ?

Q3. What is Push operation ?

Q4. What is Pop operation ?

Essay type questions.

Q1. What are some of the applications for the stack data structure ?

Q2. Explain stack operations in detail ?

Q3. Explain circular queue in detail ?

Q4. Explain dequeue ?

Answers

Ans1. a

Ans2. c

Ans3. d

Ans4. c

Ans5. a

Chapter 5

Linked List:

Linked list is a linear data structure that contains sequence of elements such that each element links to its next element in the sequence. Each element in a linked list is called as "Node".

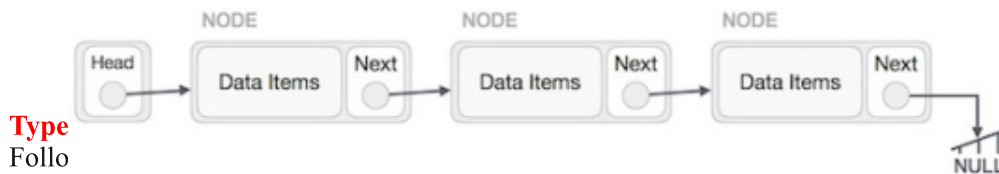
Simply a list is a sequence of data, and linked list is a sequence of data linked with each other. linked list is the second most-used data structure after array. Following are the important terms to understand the concept of Linked List.

Node – Each node contains data item and a pointer which is address of next node in list

Next – A pointer field which contains address of next node in list

Linked List Representation:

Linked list can be visualized as a chain of nodes, where every node points to the next node.



Single Linked List – Item navigation is forward only.

Doubly Linked List – Items can be navigated forward and backward.

Circular Linked List – Last item contains link of the first element as next and the first element has a link to the last element as previous.

Advantages of Linked list: Following are advantages of linked list-

- Linked List is Dynamic Data Structure.
- Linked List can grow and shrink during run time.
- Insertion and Deletion Operations are easier
- Efficient Memory Utilization, i.e, no need to pre-allocate memory
- Faster Access time can be expanded in constant time without memory overhead
- Linear Data Structures such as Stack, Queue can be easily implemented using Linked list

Disadvantages of Linked list:

- Memory wastage if required space is known
- Searching operations is difficult.

Basic Operations:

Basic operations supported by a list are

Insertion – Adds an element at the beginning of the list.

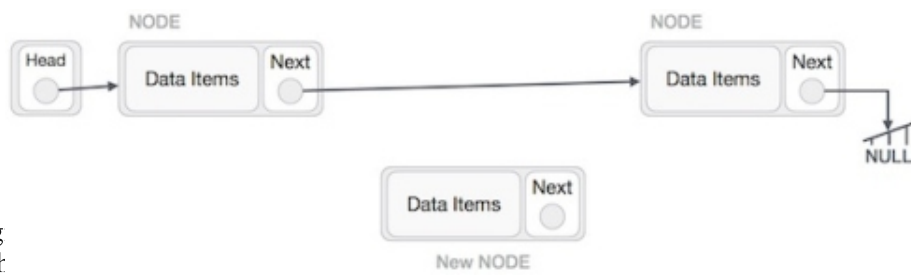
Deletion – Deletes an element at the beginning of the list.

Display – Displays the complete list.

Search – Searches an element using the given key.

Insertion Operation:

Adding a new node in linked list is a more than one step activity. We shall learn this with diagrams here. First, create a node using the same structure and find the location where it has to be inserted.

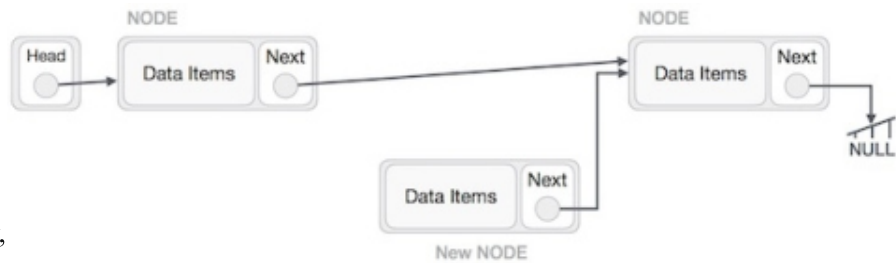


Imag
(Right

id C

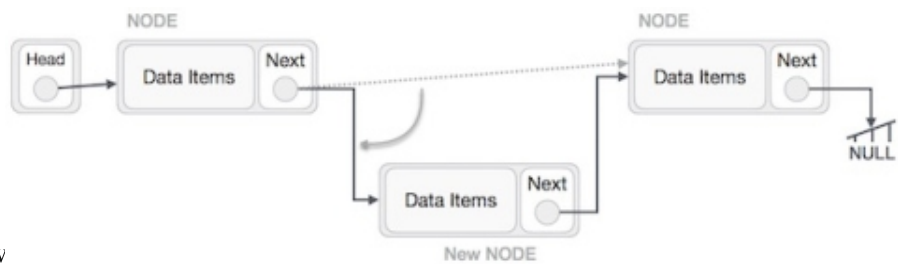
point B.next to C and NewNode.next -> RightNode;

It should look like this –

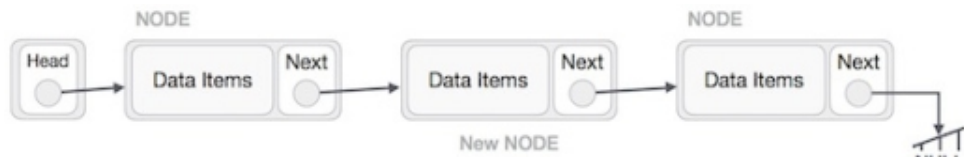


Now,

LeftNode.next -> NewNode;



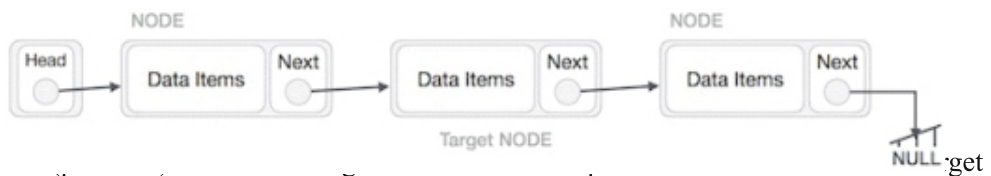
This w



Similar steps should be taken if the node is being inserted at the beginning of the list. While inserting it at the end, the second last node of the list should point to the new node and the new node will point to NULL.

Deletion Operation:

Deletion is also a more than one step process. We shall learn with pictorial representation. First, locate the target node to be removed, by using searching algorithms.



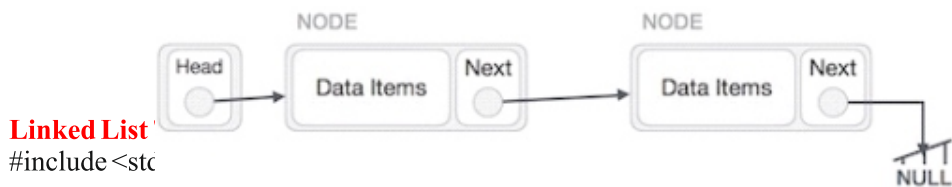
The node –
LeftNode.next -> TargetNode.next;



This we w
TargetNode.next -> NULL;



We deallocate memory and wipe off the target node completely.



Linked List
#include <stc

```

#include <string.h>
#include <stdlib.h>
#include <stdbool.h>

struct node {
    int data;
    int key;
    struct node *next;
};

struct node *head = NULL;
struct node *current = NULL;

//display the list
void printList() {
    struct node *ptr = head;
    printf("\n[ ");

    //start from the beginning
    while(ptr != NULL) {
        printf("%d,%d", ptr->key, ptr->data);
        ptr = ptr->next;
    }

    printf(" ]");
}

//insert link at the first location
void insertFirst(int key, int data) {
    //create a link
    struct node *link = (struct node*) malloc(sizeof(struct node));

    link->key = key;
    link->data = data;

    //point it to old first node
    link->next = head;

    //point first to new first node
    head = link;
}

//delete first item
struct node* deleteFirst() {

    //save reference to first link
    struct node *tempLink = head;

```

```

//mark next to first link as first
head = head->next;

//return the deleted link
return tempLink;
}

//is list empty
bool isEmpty() {
    return head == NULL;
}

int length() {
    int length = 0;
    struct node *current;

    for(current = head; current != NULL; current = current->next) {
        length++;
    }

    return length;
}

//find a link with given key
struct node* find(int key) {

    //start from the first link
    struct node* current = head;

    //if list is empty
    if(head == NULL) {
        return NULL;
    }

    //navigate through list
    while(current->key != key) {

        //if it is last node
        if(current->next == NULL) {
            return NULL;
        }else {
            //go to next link
            current = current->next;
        }
    }

    //if data found, return the current Link
    return current;
}

```

```

}

//delete a link with given key
struct node* delete(int key) {

    //start from the first link
    struct node* current = head;
    struct node* previous = NULL;

    //if list is empty
    if(head == NULL) {
        return NULL;
    }

    //navigate through list
    while(current->key != key) {

        //if it is last node
        if(current->next == NULL) {
            return NULL;
        } else {
            //store reference to current link
            previous = current;
            //move to next link
            current = current->next;
        }
    }

    //found a match, update the link
    if(current == head) {
        //change first to point to next link
        head = head->next;
    } else {
        //bypass the current link
        previous->next = current->next;
    }

    return current;
}

void sort() {

    int i, j, k, tempKey, tempData;
    struct node *current;
    struct node *next;

    int size = length();
    k = size ;

```

```

for (i = 0; i < size - 1; i++, k--) {
    current = head;
    next = head->next;

    for (j = 1; j < k; j++) {

        if (current->data > next->data) {
            tempData = current->data;
            current->data = next->data;
            next->data = tempData;

            tempKey = current->key;
            current->key = next->key;
            next->key = tempKey;
        }

        current = current->next;
        next = next->next;
    }
}

void reverse(struct node** head_ref) {
    struct node* prev = NULL;
    struct node* current = *head_ref;
    struct node* next;

    while (current != NULL) {
        next = current->next;
        current->next = prev;
        prev = current;
        current = next;
    }

    *head_ref = prev;
}

main() {
    insertFirst(1,10);
    insertFirst(2,20);
    insertFirst(3,30);
    insertFirst(4,1);
    insertFirst(5,40);
    insertFirst(6,56);

    printf("Original List: ");
}

```

```

//print list
printList();

while(!isEmpty()) {
    struct node *temp = deleteFirst();
    printf("\nDeleted value:");
    printf("(%d,%d)",temp->key,temp->data);
}

printf("\nList after deleting all items: ");
printList();
insertFirst(1,10);
insertFirst(2,20);
insertFirst(3,30);
insertFirst(4,1);
insertFirst(5,40);
insertFirst(6,56);

printf("\nRestored List: ");
printList();
printf("\n");

struct node *foundLink = find(4);

if(foundLink != NULL) {
    printf("Element found: ");
    printf("(%d,%d)",foundLink->key,foundLink->data);
    printf("\n");
} else {
    printf("Element not found.");
}

delete(4);
printf("List after deleting an item: ");
printList();
printf("\n");
foundLink = find(4);

if(foundLink != NULL) {
    printf("Element found: ");
    printf("(%d,%d)",foundLink->key,foundLink->data);
    printf("\n");
} else {
    printf("Element not found.");
}

printf("\n");
sort();

```

```

printf("List after sorting the data: ");
printList();

reverse(&head);
printf("\nList after reversing the data: ");
printList();
}

```

If we compile and run the above program, it will produce the following result –

```

Original List:
[(6,56) (5,40) (4,1) (3,30) (2,20) (1,10) ]
Deleted value:(6,56)
Deleted value:(5,40)
Deleted value:(4,1)
Deleted value:(3,30)
Deleted value:(2,20)
Deleted value:(1,10)
List after deleting all items:
[]
Restored List:
[(6,56) (5,40) (4,1) (3,30) (2,20) (1,10) ]
Element found: (4,1)
List after deleting an item:
[(6,56) (5,40) (3,30) (2,20) (1,10) ]
Element not found.
List after sorting the data:
[(1,10) (2,20) (3,30) (5,40) (6,56) ]
List after reversing the data:
[(6,56) (5,40) (3,30) (2,20) (1,10) ]

```

Important Points

- Linked list is a linear data structure that contains sequence of elements such that each element links to its next element in the sequence. Each element in a linked list is called as "Node".
- Linear Data Structures such as Stack, Queue can be easily implemented using Linked list.
- Linked List is Dynamic data Structure.

Exercise

Objective type questions.

- Q1. Linked lists are best suited
- for relatively permanent collections of data
 - for the size of the structure and the data in the structure are constantly changing
 - for both of above situation
 - for none of above situation
- Q2. Generally collection of Nodes is called as _____.
- Stack
 - Linked List
 - Heap
 - Pointer
- Q3. Which of the following is not a type of Linked List
- Doubly Linked List
 - Singly Linked List
 - Circular Linked List
 - Hybrid Linked List
- Q4. Linked list is generally considered as an example of _____ type of memory allocation.
- Static
 - Dynamic
 - Compile Time
 - None of these
- Q5. In a circular linked list
- Components are all linked together in some sequential manner.
 - There is no beginning and no end.
 - Components are arranged hierarchically.
 - Forward and backward traversal within the list is permitted.

Short answer type questions.

- Q1. Define linked list ?
 Q2. What is header linked list ?
 Q3. Which is better In array and linked list ?
 Q4. Define circular linked list ?

Essay type questions.

- Q1. Explain doubly linked list ?
 Q2. Differentiate between singly and doubly lined list ?
 Q3. Which types of memory allocates linked list ?
 Q4. Explain the uses of linked list ?

Answers

Ans1. b
 Ans4. b

Ans2. b
 Ans5. b

Ans3. d

Chapter 6

Beginning with C++

6.1 Structure of C++ Program

A C++ program contains four sections as shown in figure 6.1. These sections may be placed in different source files and then compiled independently or jointly.

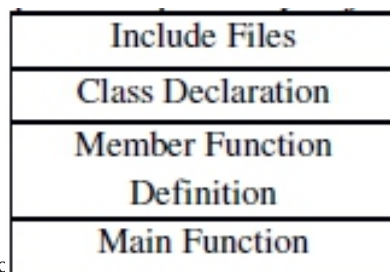


Fig 6.1 Structure of C++ Program

6.2 A Simple C++ Program

The following C++ program prints “Hello World” on the output screen.

Program 6.1 A Simple C++ Program

```
#include<iostream>          //include header file
using namespace std;
int main()
{
    cout<<“Hello World”;    //print “Hello World”
    return 0;
}
```

The output of the program 6.1 would be:

Hello World

Program features

- Like C, the C++ program is a collection of functions.
- Every C++ program must have a main function.
- Like C, the C++ statements terminate with semicolon(;).

Comments

- //(double slash) is used to comment a single line.
For example-
// This is my first C++ program.
- /*, */ used to comment multiple lines

For example-
/* This is my
first C++ program.*/

The iostream file

The statements preceded with # symbol are called pre-processor directive statements. They are placed at the beginning of the C++ program. The pre-processor processes these type of statements before the program is handed to the compiler. The #include<iostream> statement adds the contents of the iostream file to the program. It contains the declaration of the identifier cout and the insertion operator (<<).

Namespace

It defines scope of the identifiers used in the program. To use namespace scope we write using namespace std; std is the namespace where C++ standard class libraries are defined.

6.3 Compiling and Linking

The process of compiling and linking depends on the operating system.

- Linux OS
The command g++ is used to compiling and linking a C++ program.
For example-
g++ abc.cpp

g++ command compile the program written in abc.cpp file. The compiler produce an object file abc.o and then automatically link with the library functions to produce an executable file. The default executable file name is a.out.

- MS DOS
Turbo C++ and Borland C++ compilers provide integrated development environment under MS DOS. They provide a built-in editor with File, Edit, Compile and Run options.
-File option: To create and save the source file.
-Edit option: To edit the source file
-Compile option: To compile the program.
-Run option: To compile, link and run the program in one step.

6.4 Tokens

The smallest individual unit in a program is called token. C++ has the following types of tokens:

- Keywords

- Identifiers
- Constants
- Operators
- Strings

6.5 Keywords

These are the reserved words whose meaning can't be changed by the programmer. These words can't be used as a name of the variable, constants or other user-defined program elements. The complete list of C++ keywords are shown in table 6.1. Many of them are common to both C and C++.

Table 6.1 C++ Keywords

asm	double	new	switch
auto	else	operator	template
break	enum	private	this
case	extern	protected	throw
catch	float	public	try
char	for	register	typedef
class	friend	return	union
const	goto	short	unsigned
continue	if	signed	virtual
default	inline	sizeof	void
delete	int	static	volatile
do	long	struct	while

6.6

Identifiers

The names of variables, functions, arrays, classes etc. that are created by the programmer are called identifiers. Each language has its own rules for naming these identifiers. The following rules are common to both C and C++:

- Only alphabetic characters, digits and underscore are allowed.
- The name can't begin with a digit.
- Uppercase and lowercase letters are distinct.
- Keywords can't be used as a variable name.

Constants

The fixed values that can't be changed during the execution of program

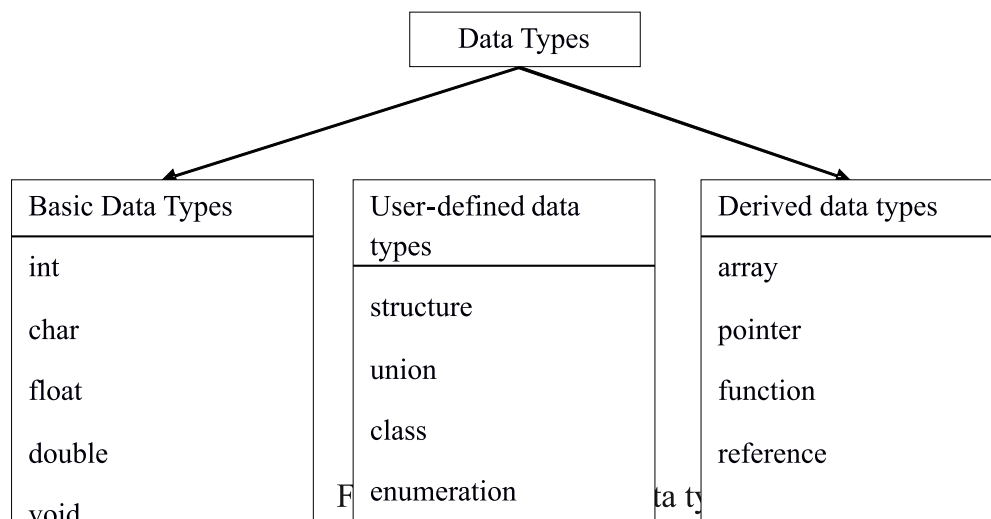
are called constants.

Examples:

```
65 // decimal integer
34.14 //floating point number
025 // octal integer
0x36 //hexadecimal integer
"Hello" //string constant
'Z' //character constant
```

6.7 Basic Data Types

Data types can be classified as shown in figure 6.2



The basic data types may have various modifiers which are placed before them, except the void data type. The modifiers may be applied to characters and integer basic data types. The modifier long can also be applied to double. The list of modifiers is as follows:

- signed
- unsigned
- short
- long

The list of all combinations of the basic data types and the modifiers along with their size and range is shown in table 6.2.

Table 6.2 Size and range of basic data types

Data type	Size in bytes	Range
char	1	-128 to 127
unsigned char	1	0 to 255
signed char	1	-128 to 127
int	2	-32768 to 32767
unsigned int	2	0 to 65535
signed int	2	-32768 to 32767
short int	1	-128 to 127
long int	4	-2147483648 to 2147483647
float	4	3.4E-38 to 3.4E+38
double	8	1.7E-308 to 1.7E+308
long double	10	3.4E-4932 to 1.1E+4932

Structure

Basic data types are not sufficient to handle real world problems. A structure is a group of basic data types and other data types. The syntax of structure is:

```
struct name_of_structure
{
    data_type member1;
    data_type member2;
    -----
    -----
};
```

Let us take an example of a student which has several attributes such as name, age, percentage etc.

```
struct student
{
    char name[20];
    int age;
    float percentage;
};
struct student student1, student2;
```

Here student1 and student2 are variables of user-defined data type 'student'.

Union

Union is similar to structure, but there is a difference between structure and union. The size of structure type is equal to the sum of sizes of individual member types while the size of union type is equal to the size of its largest member's type. For example:

```
union item
{
    int m;
    float x;
    char c;
}item1;
```

This declares a variable item1 of type 'item'. This union contains three members each with different data type. However, only one of them can be used at a time. The variable item1 will occupy four bytes in memory as its largest size member is of floating type variable x. If we define item as a structure then the variable item1 will occupy seven bytes in memory. We can say union is memory efficient alternative of structure.

Class

Class is an important feature of C++. Just like any other basic data types class type variable can be declared. The class type variables are called objects. We will discuss classes in detail in chapter 9.

Enumerated Data Type

It is way of attaching names to numbers. The enum keyword assign the numbers 0,1,2, and so on to the list of names.

For example:

```
enum color{red, green, blue};
```

By default red is assigned 0, green is assigned 1 and blue is assigned 2. We can over-ride the default values by explicitly assigning integer values to the enumerators.

For example:

```
enum color{red, green=3, blue=8};
```

Here, red is assigned 0 by default.

6.9 Derived Data Types

Arrays

An array is collection of elements of same type.

For example:

```
int numbers[5]={2, 7, 8, 9, 11};
```

Here, the 'numbers' is an array of size five and containing the five

integer type elements.

Functions

A function is part of a program which is used to perform a task. Dividing a program in functions is one of the major principles of structured programming. It reduces the size of program by calling and using them at different places in the program. We will discuss functions in more detail in chapter 8.

Pointers

A pointer is variable which is used to store the address of another variable.

For example:

```
int x=5;      //integer variable
int *ptr;    // integer pointer variable
ptr= &x;     //address of x assigned to ptr
*ptr=10;     //the value of x is changed from 5 to 10
```

Reference Type

A variable of reference type is called reference variable. It provides an alternative name for the previously defined variable.

For example:

```
int x=10;
int & y=x;
```

Here, x is an integer variable and y is a reference type variable and it is an alternative name for already declared variable x.

```
cout<<x;
```

and

```
cout<<y;
```

both statements will print the value 10. The statement

```
x=x+5;
```

will change the values of both x and y to 15.

6.10 Type Compatibility

C++ is very strict to type compatibility as compared to C. int, short int and long int are three different types. They must be type cast when their values are assigned to one another. For an operation the type of operands must be type compatible with the operation. There are two mechanisms to achieve type compatibility.

Explicit type conversion

By using the type cast operator, explicit type conversion of variables and expressions can be performed.

Program 6.2: Explicit type conversion

```
#include<iostream>
using namespace std;
int main()
{
    int i=5;
    float f=30.57;
    cout<<"i="<<i;
    cout<<"\nf="<<f;
    cout<<"\nfloat(i)="<<float(i);
    cout<<"\nint(f)="<<int(f);
    return 0;
}
```

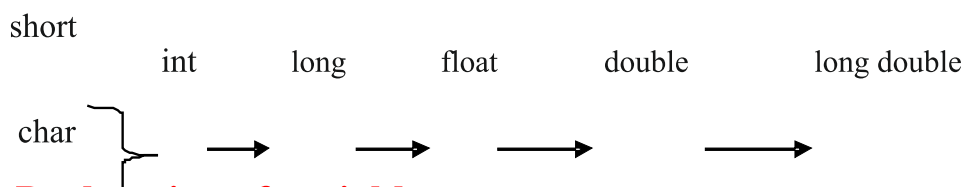
The output of the program 6.2 would be:

```
i=5
f=30.57
float(i)=5
int(f)=30
```

Implicit type conversion

When an expression consists of mixed data types, the compiler performs the automatic type conversion by using the rule that smaller type is converted to the wider type. Whenever a char or short int appears in an expression, it is converted to int. This is called integral widening conversion.

The following diagram shows the implicit conversion rule:



6.11 Declaration of variables

In C, all variables are declared at the beginning of the program. In C++, this is true but it also allows the declaration of variables anywhere in the program. For example

```
int main()
{
```

```

        int x,y;                //variable declaration
        cin>>x>>y;
        int sum=x+y; //variable declaration
        cout<<sum;
    }

```

Important Points

- C++ program is a collection of functions.
- Every C++ program must have a main function.
- C++ program statements terminate with semicolon.
- Statements preceded with # symbol are called pre-processor directive statements.
- The smallest individual unit in a program is called token.
- Reserved words whose meaning can't be changed by the programmer are called keywords.
- Names of variables, functions, arrays, classes etc. that are created by the programmer are called identifiers.
- Fixed values that can't be changed during the execution of program are called constants.
- A structure is a group of basic data types and other data types.
- An array is collection of elements of same type.
- A pointer is variable which is used to store the address of another variable.
- C++ is very strict to type compatibility as compared to C.

Practice Questions

Objective type questions:

- Q.1 Structure of a C++ program consist of
 A. Class Declaration B. Member Function Definition
 C. Main Function D. All of these
- Q.2 Symbol used to comment a single line is
 A. \\
 B. //
 C. ||
 D. !!
- Q.3 Pre-processor directive statements are preceded with the symbol
 A. \$ B. #
 C. & D. *
- Q.4 In Linux OS, the command used to compiling and linking a C++ program is
 A. g++ B. a++
 C. y++ D. z++
- Q.5 Which is a token?

- A. Keywords
- B. Identifiers
- C. Operators
- D. All of these

Q.6 Which is NOT a basic data type

- A. int
- B. char
- C. float
- D. class

Very Short Answer Type Questions:

- Q.1 What are tokens?
- Q.2 What are keywords?
- Q.3 What are identifiers?
- Q.4 What are constants?
- Q.5 What is difference between structure and union?

Short Answer Type Questions:

- Q. 1 Explain the classification of data types.
- Q.2 What is enumerated data type?
- Q.3 What is reference type?

Essay Type Questions:

- Q.1 Explain compiling and linking of a C++ program on Linux OS.
- Q.2 Explain the implicit and explicit type conversions with suitable examples.

Answer Key

- 1. D
- 2. A
- 3. B
- 4. A
- 5. D
- 6. D

Chapter 7

Operators, Expressions and Control Structures

7.1 Introduction

All operators in C are also valid in C++. In addition, C++ introduces some new operators. They are following:

- **<< Insertion operator:** It prints the contents of the variable on its right to the output screen.
- **>> Extraction operator:** It takes the value from the keyboard and assign it to the variable on its right.
- **:: Scope resolution operator:** C++ is a block-structured language. Same variable name can be used in different blocks. The scope of variable is in between the point of its declaration and end of the block containing the declaration. A variable declared inside a block is local to that block. The scope resolution operator is used to access global version of a variable.

Program 7.1: Scope resolution operator

```
#include<iostream>
using namespace std;
int x=10;      //global variable
int main()
{
    int x=20;   //x re-declared , local to main
    {
        cout<<"Inner block\n";
        int x=30;   //x declared again, local to inner block
        cout<<"x="<<x<<"\n";
        cout<<"::x="<<::x<<"\n";
    }
    cout<<"Outer block\n";
    cout<<"x="<<x<<"\n";
    cout<<"::x="<<::x<<"\n";
    return 0;
}
```

The output the program 7.1 would be:

```
Inner block
x=30
::x=10
Outer block
```

```
x=20
::x=10
```

- **new operator:** The operator allocates sufficient amount of memory to data object at run time. For example

```
int *p = new int;
```

The above statement allocates sufficient amount of memory to integer data object at run time.

- **delete operator:** The operator de-allocates the memory when the data object is no longer needed. So that the released memory can be reused by the other programs.

For example

```
delete p;
```

The above statement de-allocates the memory pointed by the pointer variable p.

7.2 Expressions and their types

An expression is a combination of operators, constants and variables arranged as per the rule of the language. There are following types of expressions:

- **Constant expressions:** It consists of only constant values. For example $20+10*5.2$
- **Integral expressions:** Those expressions which produce integer results after implementing implicit and explicit type conversions. Examples:

```
x+y*10
```

```
x+'a'
```

```
5+int(7.5)
```

where x and y are integer variables.

- **Float expressions:** Those expressions which produce floating-point results after implementing implicit and explicit type conversions. Examples:

```
a+b/5
```

```
7+float(10)
```

where a and b are float type variables.

- **Pointer expressions:** Pointer expressions produce address values.

Examples:

```
ptr=&x;
```

```
ptr+1
```

where x is a variable and ptr is a pointer.

- **Relational expressions:** Those expressions which produce Boolean type results that is either true or false. Examples:

```
x<=y
```

```
a==b
```

- **Logical expressions:** Those expressions which combines two or more relational expressions and produce Boolean type result. Examples:

```
x>y && x==5
```

```
a==20 || y==10
```

- **Bitwise expressions:** These type of expressions are used to manipulate data at bit level. They are used for testing or shifting bits. Examples:

```
a<<3 // shift three bits position to left
```

```
x>>1 // shift one bit position to right
```

- **Special Assignment Expressions:**

Chained assignment

```
a=b=10;
```

First 10 is assigned to b then to a.

Embedded assignment

```
a=(b=20)+5;
```

(b=20) is an assignment expression called embedded assignment. Here, the value 20 is assigned to b and then the result is assigned to a.

Compound assignment

It is a combination of the assignment operator and a binary arithmetic operator.

For example:

The expression

```
a=a+5;
```

can be written as

```
a+=5;
```

The operator += is called compound assignment operator or short-hand operator.

7.3 Operator precedence and associativity

If more than operators are involved in an expression, C++ language has predefined rules of priority for the operators. The operator with higher priority will execute before the operators with lower priority. This rule is called operator precedence.

If two or more operators with same precedence are present in an expression, the order in which they execute is called associativity of operators. The complete list of C++ operators with their precedence from highest to lowest and associativity is given in table 7.1.

Table 7.1 Operator precedence and associativity

Operator precedence	Associativity
::	Left to right
->, ., (), [], ++, --, ~, !, unary+, unary-, unary*	Left to right
Unary &, (type), sizeof, new, delete	Right to left
*, /, %	Left to right
+, -	Left to right
<<, >>	Left to right
<, <=, >, >=	Left to right
==, !=	Left to right
&	Left to right
^	Left to right
	Left to right
&&	Left to right
	Left to right
?:	Left to right
=, *=, /=, %=, +=	Right to left
<<=, >>=, &=, ^=, =, ,(comma)	Left to right

There are three types of control structures:

- (i) Sequence structure
- (ii) Selection structure
- (iii) Loop structure

C++ supports all the three basic control structures and implements them using various control statements.

- (i) **Sequence structure:** Statements are executed sequentially as they are written in program. Example:

```
-----  
statement1;  
statement2;  
statement3;  
-----
```

- (ii) **Selection structure:** Two or more paths of execution out of which one is selected based on a condition.
Examples:

The if statement

```
if(expression is true)  
{  
    statements;  
}
```

The if-else statement

```
if(expression is true)  
{  
    statements;  
}  
else  
{  
    statements;  
}
```

The switch statement

```
switch(expression)  
{  
    case 1: statements;  
        break;  
    case 2: statements;  
        break;
```



```

        case 3: statements;
            break;
        default : statements;
    }

```

(iii) Loop structure: Statements are executed zero or more times.

Examples:

The for statement

The for loop is used when an action is to be repeated for a predefined number of times.

```

for(initial value; test condition; increment/decrement)
{
    statements;
}

```

The while statement

The statements within the while are executed till the condition is true. It is a pre-test condition loop.

```

while(condition is true)
{
    statements;
}

```

The do-while statement

The loop is executed at least one time. It is a post-test condition loop.

```

do
{
    statements;
}while(condition is true);

```

Important Points

- All operators in C are also valid in C++.
- C++ is a block-structured language.
- An expression is a combination of operators, constants and variables arranged as per the rule of the language.

- The operator with higher precedence will execute before the operators with lower precedence.
- C++ supports all the three basic control structures and implements them using various control statements.

Practice Questions

Objective type questions:

Q.1 Operator that prints the contents of the variable on its right to the output screen is

- | | |
|-------|-------|
| A. << | B. >> |
| C. :: | D. & |

Q.2 Operator that allocates sufficient amount of memory to data object at run time is

- | | |
|-----------------------|------------------------|
| A. Insertion operator | B. Extraction operator |
| C. new operator | D. delete operator |

Q.3 In the expression $a=(b=20)+5$; the value of variable 'a' will be

- | | |
|-------|-------|
| A. 20 | B. 25 |
| C. 5 | D. 30 |

Q.4 Which is a short-hand assignment operator?

- | | |
|-------|-----------------|
| A. += | B. -= |
| C. *= | D. All of these |

Q.5 Selection structure is implemented by which control statement?

- | | |
|---------------------|----------------------|
| A. if statement | B. if-else statement |
| C. switch statement | D. All of these |

Q.6 Loop structure is implemented by which control statement?

- | | |
|-----------------------|--------------------|
| A. for statement | B. while statement |
| C. do-while statement | D. All of these |

Very Short Answer Type Questions

Q.1 Define operator precedence.

Q.2 Define associativity of operators.

Q.3 What are the different types of control structures?

Q.4 What are expressions?

Short Answer Type Questions

- Q.1 What are the uses of scope resolution operator?
- Q.2 What are the uses of new and delete operators?
- Q.3 Explain how selection control structure is implemented in C++.

Essay Type Questions

- Q.1 Explain different types of expressions with examples.
- Q.2 Explain various types of looping statements.

Answer Key

- | | | |
|------|------|------|
| 1. A | 2. C | 3. B |
| 4. D | 5. D | 6. D |

Chapter 8

Functions in C++

8.1 Introduction

A function is a part of the program which is used to perform a task. Dividing a program into functions is one of the major principles of structured programming. The advantage of using functions is that it reduces the size of program by calling and using them at different places in the program.

In C++, many new features are added to functions to make them more reliable and flexible.

8.2 Function Prototype

The function prototype gives the information to the compiler about the function like the number and type of arguments and the return type. The function prototype is a declaration statement in the calling program and its syntax is as follows:

```
type function_name(arguments-list);
```

Example:

```
int sum(int a, int b);
```

In function declaration name of arguments are dummy variables and therefore they are optional. The following statement

```
int sum(int, int);
```

is a valid function declaration.

8.3 Call-by-reference

In call-by-value parameter passing method the actual parameters in calling program are copied to formal parameters in called function. The changes made by the called function on formal parameters are not reflect in calling program.

To make the changes in actual parameters in calling program, we use call-by-reference parameter passing method.

For example

Program 8.1: Call-by reference

```
#include<iostream>
using namespace std;
int main()
{
    int count=0;
    void update(int &);
    cout<<"count="<<count<<"\n";
```

```

        update(count);
        cout<<"count="<<count;
        return 0;
    }
    void update(int &x)
    {
        x=x+1;
    }

```

The output of the program 8.1 would be:

```

count=0
count=1

```

In the above program, the variable x in update function becomes the alias of the variable count in main function.

8.4 Return by reference

A function can also return a reference. For example

Program 8.2: Return by reference

```

#include<iostream>
using namespace std;
int main()
{
    int x=6,y=9;
    int & min(int &, int &);
    min(x,y)=-1;
    cout<<"x="<<x<<"\n";
    cout<<"y="<<y;
    return 0;
}

```

```

int & min(int &a, int &b)
{
    if(a<b)
        return a;
    else
        return b;
}

```

The output of the program 8.2 would be:

```

x=-1
y=9

```

In the above program, the return type of the function min is int &, the function

returns the reference to a or b. The function calling statement `min(x,y)`; is a reference to either x or y depending on their values.

8.5 Function overloading

Same function name but different argument list can be used to perform different tasks, is known as function overloading. The correct function is to be called depends on the number and type of arguments but not on the return type of the function.

Program 8.3 Function overloading

```
#include<iostream>
using namespace std;
int sum(int, int);
int sum(int, int, int);
int main()
{
    cout<<"Sum of two numbers is "<<sum(5,10);
    cout<<"\n";
    cout<<"Sum of three numbers is "<<sum(10,20,30);
    return 0;
}
int sum(int x, int y)
{
    return(x+y);
}
int sum(int a, int b, int c)
{
    return(a+b+c);
}
```

The output of the program 8.3 would be:

Sum of two numbers is 15

Sum of three numbers is 60

In the above program, the function `sum()` is overloaded two times. When we pass two arguments to the function `sum()`, the function with two arguments is invoked and when we pass three arguments to the function `sum()`, the function with three arguments is invoked.

8.6 Inline function

When a function is called, control of execution is transferred from calling function to called function then again go back to the calling function. This is an overhead in program execution time. If the function body is small a

lot of time is spent in such overhead. The solution of this problem is inline function. An inline function is expanded in line when it invoked. The compiler replaces the function call statement with corresponding function body. To make a function inline , write inline keyword with the function declaration/definition statement.

For example

```
inline int sum(int x, int y)
{
return(x+y);
}
```

Important Points

- A function is a part of the program which is used to perform a task.
- The function prototype gives the information to the compiler about the function.
- A function can also return a reference.
- Same function name but different argument list can be used to perform different tasks, is called function overloading.
- An inline function is expanded in line when it invoked in which the compiler replaces the function call statement with corresponding function body.

Practice Questions

Objective type questions:

Q.1 Which is a valid function declaration

- A. int fun(int a, int b); B. int fun(int, int);
C. Both A and B D. None of these

Q.2 In which parameter passing method the actual parameters are copied to formal parameters of function?

- A. Call- by- reference B. Call-by-value
C. Call- by- address D. None of these

Q.3 In function overloading, the correct function is to be called is NOT depends on

- A. Number of arguments B. Type of arguments
C. Return type of the function D. None of these

Q.4 Same function name but different argument list can be used to perform different tasks, is called

- A. Function overloading B. Operator overloading
C. Class overloading D. None of these

Very Short Answer Type Questions

- Q.1 What is function?
- Q.2 What is function overloading?
- Q.3 What is inline function?

Short Answer Type Questions

- Q.1 What is function prototype?
- Q.2 What is difference between call-by-value and call-by-reference?
- Q.3 What are advantages of functions in structured programming?

Essay Type Questions

- Q.1 Write a program to swap two values by using call-by-reference mechanism.
- Q.2 Write a program to overload 'area()' function to compute the area of circle and the area of rectangle.

Answer Key

1. C 2. B 3. C 4. A

Chapter 9

Classes and Objects

9.1 Introduction

Class is the most important feature of object-oriented programming language. The concept of class is taken from the structure in C. It is a new way of creating and implementing user-defined type.

In this chapter, we shall discuss the various concepts of classes and objects.

9.2 Defining class

A class is a user-defined data type that binds data and function together. The class declaration includes the declaration of its data members and member functions.

The syntax of the class declaration is :

```
class class_name
{
    private:
        variable declaration;
        function declaration;
    public:
        variable declaration;
        function declaration;
};
```

The class members that are declared in private section can be accessed only by the members of that class. The class members that are declared in public section can be accessed from outside the class also. By default, the members of a class are private. The data hiding by using private declaration is the important feature of object-oriented programming.

A simple class example

```
class point
{
    int x,y;           // private by default
public:
    void input(int a, int b);
    void output(void);
};
```

Creating objects

Like basic data type, we can create the variables of class type. These variables are called objects.

For example

```
point p, q;
```

In the above statement, two objects p and q of class type point are created.

Accessing class members

The public members of the class can be accessed from outside the class by using the object of that class.

The syntax for accessing a public member function is:

```
object_name.function_name(arguments list);
```

For example, the function call statement

```
p.input(10,20);
```

assign the value 10 to x and 20 to y of the object p by defining the input() function.

The statement `p.x=10;` is illegal, since x is declared private and it can only be accessed by the member functions directly and not by the object from outside the class.

9.3 Defining member functions

The member function of a class can be defined within the class and outside the class also.

Inside the class

The function declaration is replaced by the actual definition of the member function inside the class. The function defined inside the class are treated as an inline function.

For example

```
class point
{
    int x,y;
public:
    void input(int a, int b)
    {
        x=a;
        y=b;
    }
    void output(void)
    {
        cout<<"x="<<x<<"\n";
        cout<<"y="<<y;
    }
};
```

Outside the class

The member functions declared in a class must be defined separately outside the class. The format for member function definition is:

```

return_type class_name:: function_name(arguments)
{
    function body
}

```

The class name indicates the function belongs to this particular class.

For example

```

class point
{
    int x,y;
public:
    void input(int a, int b);
    void output(void);
};
void point :: input(int a, int b)
{
    x=a;
    y=b;
}
void point :: output(void)
{
    cout<<"x="<<x<<"\n";
    cout<<"y="<<y;
}

```

Program 9.1 A simple program with class

```

#include<iostream>
using namespace std;
class point
{
    int x,y;
public:
    void input(int a, int b);
    void output(void);
};
void point :: input(int a, int b)
{
    x=a;
    y=b;
}
void point :: output(void)
{
    cout<<"x="<<x<<"\n";

```

```

        cout<<"y="<<y;
    }
int main()
{
    point p;
    p.input(5,10);
    p.output();
return 0;
}

```

The output of the program 9.1 would be:

```

x=5
y=10

```

9.4 Access Modifiers

public and private keywords are called access modifiers. Since, they control the access mechanism of members of a class.

- The public member of a class can be accessible from outside the class. Generally, member function of a class are kept in public section of the class.
- The private members of a class can't be accessible from outside the class even with the object of that class. Generally, variables are kept in private section of the class.

9.5 Arrays within class

Arrays can act as a data member of a class.

Program 9.2 Array within class

```

#include<iostream>
using namespace std;
class data
{
    int a[5];
public:
    void getdata(void);
    void showdata(void);

```

```

};
void data :: getdata(void)
{
    cout<<"Enter the elements of array\n";
    for(int i=0; i<5; i++)
    {
        cin>>a[i];
    }
}
void data :: showdata(void)
{
    cout<<"Array elements are\n";
    for(int i=0; i<5; i++)
        cout<<a[i]<<"\t";
}
int main()
{
    data d;
    d.getdata();
    d.showdata();
return 0;
}

```

The output of the program 9.2 would be:

```

Enter the elements of array
6    5    9    8    1
Array elements are
6    5    9    8    1

```

9.6 Static Data Members

The data members of a class can be declared as a static. The characteristics of static data members are:

- Its initial value is set to zero, when first object of its class is created.
- Only single copy of the data member is created and it is shared by all the objects of the class.
- Since it is associated with the entire class, it is also called class

variables.

Program 9.3 Static Data Members

```
#include<iostream>
using namespace std;
class data
{
    static int x;
    int y;
public:
    void getdata(int a)
    {
        y=a;
        x++;
    }
    void show_x(void)
    {
        cout<<"x="<<x<<"\n";
    }
};
int data :: x; // static member definition
int main()
{
    data d1, d2; // x is initialized to zero
    d1.show_x();
    d2.show_x();
    d1.getdata(10);
    d2.getdata(20);
    cout<<"After reading data"<<"\n";
    d1.show_x();
    d2.show_x();
    return 0;
}
```

The output of the program 9.3 would be:

x=0

x=0

After reading data

x=2

x=2

The static data member x is initialized to zero when objects are created. The value of x is incremented by one each time the function getdata() is called.

Since the static variable x is shared between the two objects, the value of x is printed 2 each time. The initial value can be assigned to static data member, when it defined outside the class. For example:

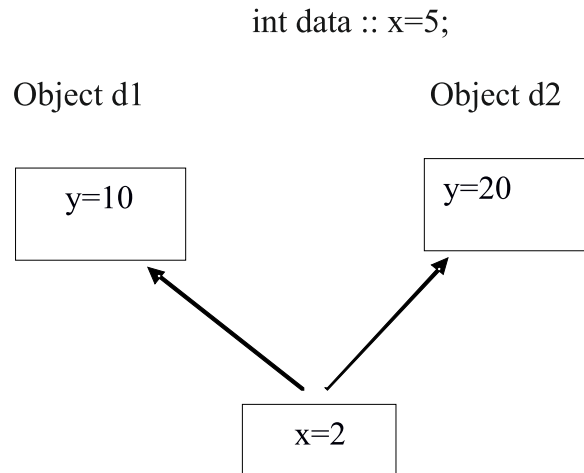


Fig. 9.1 Sharing of a static data member

9.7 Static Member Function

A member function declared with static is called static member function. The properties of static member functions are:

- It can access only other static data members and member functions in the same class.
- They are invoked using the class name.

Program 9.4: Static Member Function

```
#include<iostream>
using namespace std;
class test
{
    int x;
    static int y;
public:
    void set_xy(int a)
    {
        x=a;
        y++;
    }
    void show_x(void)
    {
        cout<<"x="<<x<<"\n";
    }
}
```

```

    }
    static void show_y(void)
    {
        cout<<"y="<<y;
    }
};
int test::y;
int main()
{
    test t1, t2;
    t1.set_xy(10);
    t2.set_xy(20);
    t1.show_x();
    t2.show_x();
    test::show_y();    // calling static function
    return 0;
}

```

The output of the program 9.4 would be:

```

x=10
x=20
y=2

```

9.8 Friend function

As we know that the private members of a class can't be accessed from outside the class. A friend function can access the private data of a class through the object of that class. A function that is common to two classes, we can make this as a friend to these two classes. The function is declared with keyword friend. A friend function has following characteristics:

- It is invoked like a normal function, not with the any object of the class.
- It can only access the members of the class by using the object of that class.
- It can be declared anywhere in the class.
- Generally, it has objects as arguments.

Program 9.5: Friend function

```

#include<iostream>
using namespace std;
class test
{
    int x,y;
    public:
    void getdata(int a, int b)

```



```

        {
            x=a;
            y=b;
        }
    friend int sum(test t);
};

int sum(test t)
{
    return(t.x+t.y);
}
int main()
{
    test q;
    q.getdata(10,20);
    cout<<"Sum="<<sum(q);
    return 0;
}

```

The output of the program 9.5 would be:
Sum=30

Friend class

A member function of a class can be friend function of another class.

For example

```

class A
{
void fun(); // member function of A
};
class B
{
-----
friend void A::fun();
-----
};

```

Note that the friend function in class B is declared with the class name and scope resolution operator.

The function fun() is a member function of class A and friend function of class B.

If all the member function of one class are declared as friend functions in another class, then the class is called friend class. For example

```

class C
{

```

```

-----
friend class A;           //All member functions of class A are friend to C
-----
};

```

Program 9.6: Using friend function to find maximum between data members of two classes

```

#include<iostream>
using namespace std;
class second; //forward declaration
class first
{
    int x;
public:
    void set_value(int a)
    {
        x=a;
    }
    friend void max(first, second);
};
class second
{
    int y;
public:
    void set_value(int b)
    {
        y=b;
    }
    friend void max(first, second);
};
void max(first f, second s)
{
    if(f.x>s.y)
        cout<<"Maximum is "<<f.x;
    else
        cout<<"Maximum is "<<s.y;
}
int main()
{
    first A;
    second B;
    A.set_value(10);
}

```

```

        B.set_value(20);
        max(A,B); // calling friend function
        return 0;
    }

```

The output of the program 9.6 would be:
Maximum is 20

9.9 Returning Objects

In the previous section, we saw that the friend functions receive objects as arguments. A friend function can return object also.

Program 9.7: Returning object

```

#include<iostream>
using namespace std;
class vector
{
    int V[3];
public:
    void set_vector(void)
    {
        cout<<"Enter three numbers\n";
        for(int i=0; i<3; i++)
            cin>>V[i];
    }
    void display(void)
    {
        for(int i=0; i<3; i++)
            cout<<V[i]<<" ";
    }
    friend vector sum(vector, vector);
};
vector sum(vector p, vector q)
{
    vector r;
    for(int j=0; j<3; j++)
        r.V[j]=p.V[j]+q.V[j];
    return r;
}
int main()
{
    vector v1, v2, v3;
    v1.set_vector();

```

```

        v2.set_vector();
        v3=sum(v1,v2);
        cout<<"First vector is:";
        v1.display();
        cout<<"\n";
        cout<<"Second vector is:";
        v2.display();
        cout<<"\n";
        cout<<"Resultant vector is:";
        v3.display();
        return 0;
    }

```

The output of the program 9.7 would be:

```

Enter three numbers
3    -2    5
Enter three numbers
-8    6    7
First vector is: 3,-2,5,
Second vector is: -8,6,7,
Resultant vector is: -5,4,12

```

9.10 Pointers to members

We can assign the address of a class member to a pointer

For example

```

class X
{
    int a;
    public void show();
};

```

We can define pointer to member as:

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

X::* means "pointer-to-member of X".

&X::a means "address of the member a of the class X"

The statement `int *p=&a;` will not work;

The pointer `p` can be used to access the member `a` inside member function or friend function.

For example

```

void show()
{
    X x;           // object created
}

```

```

        cout<<x.*p; //display value of a
        cout<<x.a; //same as above
    }

```

We can also set a pointer to member function of a class. The member function can be invoked using dereferencing operator (.*).

For example

```

X x;           // object created
void (X::*pf)()=&X::show;
(x.*pf());    //invoke show()

```

Here, pf is a pointer to member function show().

Important Points

- A class is a user-defined data type that binds data and function together.
- By default, the members of a class are private.
- The member function of a class can be defined within the class and outside the class also.
- public and private keywords are called access modifiers.
- The data members of a class can be declared as a static.
- A member function declared with static is called static member function.
- A friend function can access the private data of a class through the object of that class.
- We can assign the address of a class member to a pointer.

Practice Questions

Objective type questions:

Q.1 User-defined data type that binds data and function together is called

- | | |
|-----------|------------|
| A. Object | B. Class |
| C. Array | D. Pointer |

Q.2 By default, the members of a class are

- | | |
|--------------|------------------|
| A. public | B. private |
| C. protected | D. None of these |

Q.3 Which is an Access Modifier?

- | | |
|-----------------|------------------|
| A. public | B. private |
| C. Both A and B | D. None of these |

Q.4 Which is true with respect to static data members?

- | |
|--|
| A. Its initial value is set to zero, when first object of its class is created |
| B. Only single copy of the data member is created. |
| C. Also called class variables. |

- D. All of these
- Q.5 Which is true with respect to static member functions?
- A. Declared with static keyword
 - B. Access only other static data members and member functions in the class.
 - C. Invoked using the class name.
 - D. All of these
- Q.6 Which is true with respect to friend functions?
- A. Invoked like a normal function.
 - B. Declared anywhere in the class.
 - C. Generally, it has objects as arguments.
 - D. All of these

Very Short Answer Type Questions

- Q.1 What is class?
- Q.2 What is object?
- Q.3 What is friend class?

Short Answer Type Questions

- Q.1 Differentiate between private and public access modifiers.
- Q.2 What are the characteristics of static data members?
- Q.3 What are the properties of static member functions?

Essay Type Questions

- Q.1 What is friend function? Write its characteristics.
- Q.2 Write a program to create a class 'complex', that represents a complex number and define the member functions to compute addition and subtraction of two complex numbers.
- Q.3 Write a program to swap data members of two classes using friend function.

Answer Key

- | | | |
|------|------|------|
| 1. B | 2. B | 3. C |
| 4. D | 5. D | 6. D |

Chapter 10

Constructors and Destructors

10.1 Introduction

In the examples of classes, we have used member functions like `input()`, `getdata()` etc. to initialize the private data members of a class. The function call statements are used with the objects that have already been created. The functions are unable to initialize data members at the time of creation of their objects.

The aim of C++ is that the class behaves like a basic data type. A class type variable (object) should be initialized when it is declared in the same way as basic data type variables.

In this chapter, we will discuss a special member function called constructor which is used to initialize objects when they are created. An another member function destructor that destroys the object when they are no longer required.

10.2 Constructors

A constructor is a special member function that is used to initialize the objects of its class. It is called automatically when the objects of its class are created.

Special Characteristics of constructor functions are:

- Its name is same as class name.
- It must be declared in public section.
- Invoked automatically when objects are created.
- Do not have any return type, not even void and hence, it can't return any value.
- It can't be inherited.
- We can't access their addresses.

For example

```
class point
{
    int x,y;
public:
point(void); //constructor declared
-----
};
point :: point(void) //constructor defined
{
    x=0;
    y=0;
}
```

When we declare the object of the class point. For example

```
point p;
```

The constructor in the class is automatically called and initialize the private data members x and y to zero for the object p.

A constructor with no arguments is called default constructor. If no such constructor is defined in a class, then compiler provides a default constructor to create the object of the class.

10.3 Parameterized constructors

The constructors that receive arguments are called parameterized constructors. For example

```
class point
{
    int x, y;
public:
    point(int a, int b);    // parameterized constructor
    {
        -----
        -----
    }
};
point::point(int a, int b)
{
    x=a;
    y=b;
}
```

The parameterized constructors can be called in two ways:

```
point p= point(10,20); //explicit call
```

This statement create an object p and passes the values 10 and 20 to it.

```
point p(10,20); //implicit call
```

This statement works same as above statement.

The constructor functions can also be defined as inline functions. For example

```
class point
{
    int x, y;
public:
    point(int a, int b)
    {
        x=a;
        y=b;
    }
}
```



```

    }
-----
-----
};

```

The arguments of a constructor can be of any type except the class to which it belongs. For example

```

class X
{
-----
-----
public:
X(X);
};

```

is illegal.

But a constructor can accept a reference to its own class as an argument. For example

```

class X
{
-----
-----
public:
X(X&);
};

```

is valid and the constructor is called copy constructor.

Program 10.1: Parameterized constructor

```

#include<iostream>
using namespace std;
class rectangle
{
    int length;
    int breadth;
public:
    rectangle(int a, int b)
    {
        length=a;
        breadth=b;
    }
    void area()
    {
        cout<<"Area="<<length*breadth;
    }
}

```

```
};
int main()
{
    rectangle r(5,10);
    return 0;
}
```

The output of the program 10.1 would be:
Area=50

10.4 Multiple constructors in a class

A class can have more than one constructors and it is called constructor overloading.

Program 10.2: Overloaded constructors

```
#include<iostream>
using namespace std;
class point
{
    int x,y;
public:
    point()// no argument constructor
    {
        x=0;
        y=0;
    }
    point(int a) //one argument constructor
    {x=y=a;}
    point(int m, int n) //two arguments constructor
    {
        x=m;
        y=n;
    }
    void show()
    {
        cout<<"x="<<x<<"\n";
        cout<<"y="<<y<<"\n";
    }
};
int main()
{
    point p1;
    point p2(5);
    point p3(7,11);
}
```

```

        cout<<"Coordinates of p1 are\n";
        p1.show();
        cout<<"Coordinates of p2 are\n";
        p2.show();
        cout<<"Coordinates of p3 are\n";
        p3.show();
        return 0;
    }

```

The output of the program 10.2 would be:

Coordinates of p1 are

x=0

y=0

Coordinates of p2 are

x=5

y=5

Coordinates of p3 are

x=7

y=11

In the above program the point has three constructors. First is no argument constructor and it initializes the object with zero values. Second constructor receives one value as an argument and initialize the object with this value. Third constructor receives two arguments and initialize the object with these two values.

10.5 Constructor with default arguments

The constructor can take default arguments. For example

```
point(int a, int b=0);
```

Note that default arguments are given from right to left. The default value of argument b is zero. Then, the statement

```
point p(5);
```

assign the value 5 to a and 0 to b(by default). But the statement

```
point(7,11);
```

assign the value 7 to a and 11 to b because when actual parameters are given, they overrides the default arguments.

If one argument constructor is also present with this constructor , then the calling statement

```
point p(5);
```

is unable to decide which constructor is to be called and an ambiguity is created. The compiler will generate an error message.

10.6 Dynamic initialization of objects

The initial value of an object can be provided at the run time. The advantage of dynamic initialization is that we can give different input formats by using constructor overloading.

Program 10.3: Dynamic initialization of objects

```
#include<iostream>
using namespace std;
class shape
{
    float length, breadth;
    float radius;
    float area;
public:
    shape() {}
    shape(float r)
    {
        radius=r;
        area=3.14*r*r;
    }
    shape(float l, float b)
    {
        length=l;
        breadth=b;
        area=length*breadth;
    }
    void display()
    {
        cout<<"Area="<<area<<"\n";
    }
};
int main()
{
    shape circle, rectangle;
    float r, l, b;
    cout<<"Enter the radius of circle\n";
    cin>>r;
    circle=shape(r);
    cout<<"Enter the length and breadth of rectangle\n";
    cin>>l>>b;
    rectangle=shape(l,b);
    cout<<"Area of circle\n";
    circle.display();
    cout<<"Area of rectangle\n";
```

```

        rectangle.display();
        return 0;
    }

```

The output of the program 10.3 would be:

```

Enter the radius of circle
5
Enter the length and breadth of rectangle
17    8
Area of circle
78.5
Area of rectangle
136

```

10.7 Copy constructor

A constructor that is used to declare and initialize object from another object of the same class is known as copy constructor. A copy constructor takes a reference to an object of the same class as an argument.

Program 10.4: Copy constructor

```

#include<iostream>
using namespace std;
class product
{
    int code;
public:
    product(){} // default constructor
    product(int x)//parameterized constructor
    {
        code=x;
    }
    product(product &y) //copy constructor
    {
        code=y.code; //copy the value
    }
    void display(void)
    {
        cout<<code;
    }
};
int main()
{
    product p1(10);

```

```

        product p2(p1);    //copy constructor called
        product p3=p1;    //again copy constructor called
        cout<<"Code of p1:";
        p1.display();
        cout<<"\nCode of p2:";
        p2.display();
        cout<<"\nCode of p3:";
        p3.display();
        return 0;
    }

```

The output of the program 10.4 would be:

```

Code of p1:10
Code of p2:10
Code of p3:10

```

Note: When no copy constructor is defined in the program, the compiler supplies its own copy constructor.

10.8 Destructors

A special member function of the class that is used to destroy the objects that have been created by constructor.

Special characteristics of destructors:

- Its name is same as class name but preceded by tilde(~).
- It never takes any argument and does not return any value.
- It is invoked implicitly by the compiler upon exit from the program or block or function.

The following program shows the destructor is invoked implicitly by the compiler.

Program 10.5: Implementation of Destructor

```

#include<iostream>
using namespace std;
class sample
{
    sample()            //Constructor
    {
        cout<<"Object created\n";
    }
    ~sample()          //Destructor
    {
        cout<<"Object destroyed";
    }
};

```

```

int main()
{
    sample s;
    return 0;
}

```

The output of the program 10.5 would be:
Object created
Object destroyed

The use of destructors are to free the allocated memory to objects at run time. So, that the freed memory can be reuse for another program or objects. The memory is allocated to an object by using new operator in constructor function and de-allocated by using delete operator in destructor function.

Program 10.6: Memory de-allocation of an object using destructor.

```

#include<iostream>
using namespace std;
class sample
{
    char *t;
public:
    sample(int length)
    {
        t=new char[length];
        cout<<"Character array of length"<<length<<"created";
    }
    ~sample()
    {
        delete t;
        cout<<"\n Memory de-allocated for the character array";
    }
};

int main()
{
    sample s(10);
    return 0;
}

```

The output of the program 10.6 would be:
Character array of length 10 created

Memory de-allocated for the character array

Important Points

- A constructor is a special member function that is used to initialize the objects of its class.
- The arguments of a constructor can be any type except the class to which it belongs.
- A constructor can accept a reference to its own class as an argument.
- A constructor that is used to declare and initialize object from another object of the same class is known as copy constructor.
- When no copy constructor is defined in the program, the compiler supplies its own copy constructor.
- A special member function of the class that is used to destroy the objects that have been created by constructor is called destructor.

Practice Questions

Objective type questions:

- Q.1 Which is true with respect to constructor?
- A. Its name is same as class name.
 - B. It must be declared in public section.
 - C. Invoked automatically when objects are created.
 - D. All of these
- Q.2 Constructors that take arguments are called
- A. Default constructors
 - B. No argument constructors
 - C. Parameterized constructors
 - D. None of these
- Q.3 A constructor that is used to declare and initialize object from another object of the same class is known as
- A. Default constructor
 - B. Copy constructor
 - C. Parameterized constructor
 - D. None of these
- Q.4 Which is true with respect to destructors?
- A. Its name is same as class name but preceded by tilde(~).
 - B. It never takes any argument and does not return any value.
 - C. It is invoked implicitly by the compiler upon exit from the program or block or function.
 - D. All of these

Very Short Answer Type Questions

- Q.1 What are constructors?
- Q.2 What are parameterized constructors?
- Q.3 What is constructor overloading?
- Q.4 What is copy constructor?

Short Answer Type Questions

Q.1 What are the characteristics of constructors?

Q.2 What are destructors? Write its properties.

Q.3 What are the uses of destructors?

Essay Type Questions

Q.1 Explain constructor with default arguments.

Answer Key

1. D

2. C

3. B

4. D

Chapter 11

Operator Overloading

11.1 Introduction

Operator overloading is an important feature of C++ language. Using this feature, we can add two variables of user-defined types with the same way as we do with basic data types.

The mechanism of giving special meanings to an operator for a data type is known as operator overloading.

All the operators in C++ can be overload except the following:

- Class member access operators (.,.*)
- Scope resolution operator (::)
- Size operator (sizeof)
- Conditional operator (?:)

When we overload an operator, its original meaning remains same. For example, if we overload + operator to add two matrices, can still be used to add two numbers.

11.2 The operator function

To give additional meaning to an operator, we use a special function called operator function. The prototype of the operator function is :

```
return_type class_name :: operator op(arguments list)
{
    function body
}
```

Where operator is a keyword and op is an operator to be overload.

Operator function must be either member function or friend function of a class. The basic difference between them is that member function takes no argument for unary operators and one argument for binary operators while friend function takes one argument for unary operators and two arguments for binary operators.

11.3 Overloading unary operators using member function

Let us consider the postfix increment operator (++). It takes just one operand and increment its value by one , when applied to basic data types. We will overload this operator so that it can be applied to an object and it increments each data item of that object by one.

Program 11.1 Overloading postfix increment operator

```
#include<iostream>
using namespace std;
class point
```

```

{
    int x,y;
public:
    void getdata(int a, int b)
    {
        x=a;
        y=b;
    }
    void show(void)
    {
        cout<<"x="<<x;
        cout<<"y="<<y<<"\n";
    }
    void operator++(int)
    {
        x++;
        y++;
    }
};

int main()
{
    point p;
    p.getdata(5,8);
    cout<<"p:";
    p.show();
    p++; // invoke operator function
    cout<<"p++:";
    p.show();
    return 0;
}

```

The output of the program 11.1 would be:

```

p: x=5 y=8
p++: x=6 y=9

```

The int in the operator function is used to indicate that we are overloading postfix increment operator not prefix increment operator.

11.4 Overloading unary operator using friend function

Let us consider prefix decrement operator (--) that takes one operand and decrement the value of operand by one , when applied to basic data types. We

will overload this operator so that it can be applied to an object and decrement the value of each data of that object by one.

Program 11.2: Overloading prefix decrement operator

```
#include<iostream>
using namespace std;
class point
{
    int x,y;
public:
    void getdata(int a, int b)
    {
        x=a;
        y=b;
    }
    void show(void)
    {
        cout<<"x="<<x;
        cout<<"y="<<y<<"\n";
    }
    friend void operator--(point &s)
    {
        s.x=s.x-1;
        s.y=s.y-1;
    }
};

int main()
{
    point p;
    p.getdata(7,10);
    cout<<"p:";
    p.show();
    --p;
    cout<<"--p:";
    p.show();
    return 0;
}
```

The output of the program 11.2 would be:

```
p: x=7 y=10
p--: x=6 y=9
```

Note that the argument in operator function is passed by reference. If we pass argument by value it will not work because the changes made in operator function will not reflect in main function.

We can't overload the following operators using friend function.

- = Assignment operator
- () Function call operator
- [] Subscripting operator
- -> Class member access operator

11.5 Overloading binary operators using member function

Let us consider the binary + operator that takes two operands and add them, when applied to basic data types. We will overload this operator to add two matrices.

Program 11.3: Overloading binary + operator

```
#include<iostream>
using namespace std;
class matrix
{
    int mat[2][2];
public:
    void getmatrix(void);
    matrix operator+(matrix);
    void showmatrix(void);
};
void matrix::getmatrix(void)
{
    for(int i=0; i<2; i++)
        for(int j=0; j<2; j++)
        {
            cout<<"Enter the number:";
            cin>>mat[i][j];
        }
}
matrix matrix::operator+(matrix m)
{
    matrix temp;
    for(int i=0; i<2; i++)
        for(int j=0; j<2; j++)
            temp.mat[i][j]=mat[i][j]+m.mat[i][j];
    return temp;
}
```

```

void matrix::showmatrix(void)
{
    for(int i=0; i<2; i++)
    {
        for(int j=0; j<2; j++)
            cout<<mat[i][j]<<"\t";
        cout<<"\n";
    }
}

```

```

int main()
{
    matrix m1,m2,m3;
    m1.getmatrix();
    m2.getmatrix();
    m3=m1+m2;
    cout<<"matrix m1:\n";
    m1.showmatrix();
    cout<<"matrix m2:\n";
    m2.showmatrix();
    cout<<"Resultant matrix:\n";
    m3.showmatrix();
    return 0;
}

```

The output of the program 11.3 would be:

```

Enetr the number: 2
Enetr the number: 3
Enetr the number: 1
Enetr the number: 4
Enetr the number: 6
Enetr the number: 7
Enetr the number: 8
Enetr the number: 9
matrix m1:
2    3
1    4
matrix m2:
6    7
8    9
Resultant matrix:
8    10

```

9 13

In the above program the operator function takes only one argument of matrix type and that is second operand of binary + operator. The first operand m1 is used to invoking the operator function. So, the data members of m1 are directly accessed by the operator function. The following statement

```
m3=m1+m2;
```

is equivalent to

```
m3 =m1.operator+(m2);
```

For binary operators, the left-side operand is used to invoke the operator function and the right-side operand is passed as an argument.

11.6 Overloading binary operators using friend function

The following program overload the binary + operator to add two complex numbers using friend function.

Program 6.4: Overloading binary + operator using friend function

```
#include<iostream>
using namespace std;
class complex
{
    float real;
    float imag;
public:
    void input(float x, float y)
    {
        real=x;
        imag=y;
    }
    friend complex operator + (complex a, complex b)
    {
        complex c;
        c.real=a.real+b.real;
        c.imag=a.imag+b.imag;
        return c;
    }
    void show(void)
    {
        cout<<real<<"+"<<imag<<"\n";
    }
};
int main()
{
    complex c1,c2,c3;
```

```

    c1.input(1.6,6.2);
    c2.input(2.3,3.4);
    c3=c1+c2;           //invoke operator function
    cout<<"C1=";
    c1.show();
    cout<<"C2=";
    c2.show();
    cout<<"C3=";
    c3.show();
    return 0;
}

```

The output of the program 11.4 would be:

C1=1.6+i6.2;

C2=2.3+i3.4;

C3=3.9+i9.6;

In the above program, the operator function takes two arguments of complex type explicitly and return a resultant complex number. The following statement

`c3=c1+c2;`

is equivalent to

`c3=operator+(c1,c2);`

Important Points

- The mechanism of giving special meanings to an operator for a data type is known as operator overloading.
- When we overload an operator, its original meaning remains same.
- To give additional meaning to an operator, we use a special function called operator function.
- Operator function must be either member function or friend function of a class.

Practice Questions

Objective type questions:

Q.1 Operator that can be overload is

- Scope resolution operator (::)
- Class member access operators (.,.*)
- Binary plus operator (+)
- Conditional operator (?:)

Q.2 To overload a binary operator, operator function as member function will take

- A. Two arguments B. One argument
C. Zero argument D. None of these
- Q.3 To overload a unary operator, operator function as friend function will take
A. Two arguments B. One argument
C. Zero argument D. None of these
- Q.4 Operator that can't be overload using friend function.
A. = Assignment operator B. () Function call operator
C. [] Subscripting operator D. All of these

Very Short Answer Type Questions

- Q.1 What is operator overloading?
Q.2 Write the prototype of the operator function.
Q.3 What are the operators that can't be overloaded?

Short Answer Type Questions

- Q.1 Explain the difference between the operator function implemented as member function and friend function.

Essay Type Questions

- Q.1 Write a program to overload unary minus operator to negate an object of a class using friend function.
Q.2 Write a program to overload binary plus operator to concatenate two strings using member function.

Answer Key

1. C 2. B 3. B 4. D

Chapter 12

Inheritance

12.1 Introduction

Reusability is an important feature of C++. The existing classes are used to create new classes, this mechanism is called inheritance. Using this feature the programmer can save time, money and effort. The existing class is called base class or parent class or super class and the new class is called derived class or child classes or subclass.

12.2 Defining derived classes

The syntax of derived class is:

```
class derived-class-name : visibility-mode base-class-name
{
    members of derived class.
};
```

The visibility-mode can be either private, protected or public. By default, the visibility-mode is private. The visibility-mode specifies whether the features of the base class are privately, protectedly or publically derived .

If the base class is privately inherited by the derived class , the public and protected members of the base class become private members of the derived class. The private members of the base class are never inherited .

If the base class is protectedly inherited by the derived class, the protected and public members of the base class become protected members of the derived class.

If the base class is publically inherited by the derived class, the protected members of the base class become protected members of the derived class and the public members of the base class become public members of the derived class.

12.3 Single Inheritance

In single inheritance, there is one base class and one derived class.

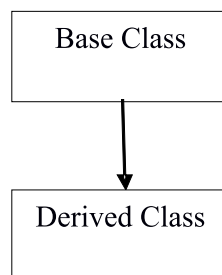


Fig. Inheritance

Program 12.1: Single inheritance

```

#include<iostream>
using namespace std;
class data
{
protected:
    int x,y;
public:
    void getdata(int a, int b)
    {
        x=a;
        y=b;
    }
    void showdata(void)
    {
        cout<<"x="<<x<<"\n";
        cout<<"y="<<y<<"\n";
    }
};
class maximum: public data
{
public:
    void max(void)
    {
        if(x>y)
            cout<<"Maximum is:"<<x;
        else
            cout<<"Maximum is:"<<y;
    }
};

int main()
{
    maximum m;
    m.getdata(4,9);
    m.showdata();
    m.max();
return 0;
}

```

The output of the program 12.1 would be:

x=4

y=9

Maximum is: 9

In the above program, the base class 'data' has two protected data members x and y. These two data members can be accessed by the base class and its immediate derived class, but not outside to these two classes. The derived class 'maximum' compute maximum between these two data members. After public derivation of the base class, the derived class has the following members.

Derived class 'maximum'

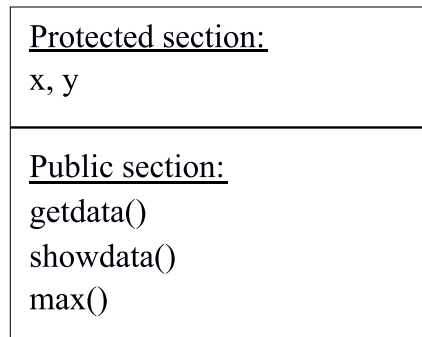


Fig. 12.2 Members of derived class 'maximum'

12.4 Multilevel inheritance

A class can be derived from another derived class.

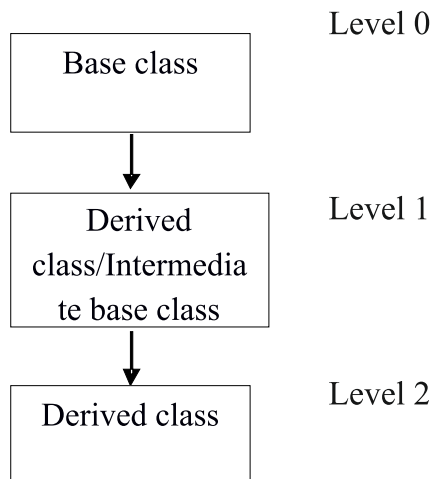


Fig. 12.3 Multilevel inheritance

There can be any number of levels in multilevel inheritance. The following program is an example of multilevel inheritance.

Program 12.2: Multilevel inheritance

```
#include<iostream>
using namespace std;
```

```

class data1
{
protected:
    int x;
public:
    void get_x(int a)
    {
        x=a;
    }
    void show_x(void)
    {
        cout<<"x="<<x<<"\n";
    }
};

class data2:public data1
{
protected:
    int y;
public:
    void get_y(int b)
    {
        y=b;
    }
    void show_y(void)
    {
        cout<<"y="<<y<<"\n";
    }
};

class addition: public data2
{
    int z;
public:
    void sum(void)
    {
        z=x+y;
    }
    void show_z(void)
    {
        cout<<"z="<<z<<"\n";
    }
}

```

```
};

int main()
{
    addition a;
    a.get_x(4);
    a.get_y(7);
    a.sum();
    a.show_x();
    a.show_y();
    a.show_z();
    return 0;
}
```

The output of the program 12.2 would be:

```
x=4
y=7
z=11
```

In the above program, the derived class 'data2' is derived from the base class 'data1' and this is the first level of derivation . The protected data member x of base class 'data1' become protected in derived class 'data2'. After first level of derivation, the derived class 'data2' has the following members.

Derived class 'data2'

<u>Protected section:</u> x, y
<u>Public section:</u> get_x() show_x() get_y() show_y()

Fig. 12.4 Members of derived class 'data2'

The class 'addition' is derived from intermediate base class 'data2'. After this second level of derivation the derived class 'addition' has the following members.

Derived class 'addition'

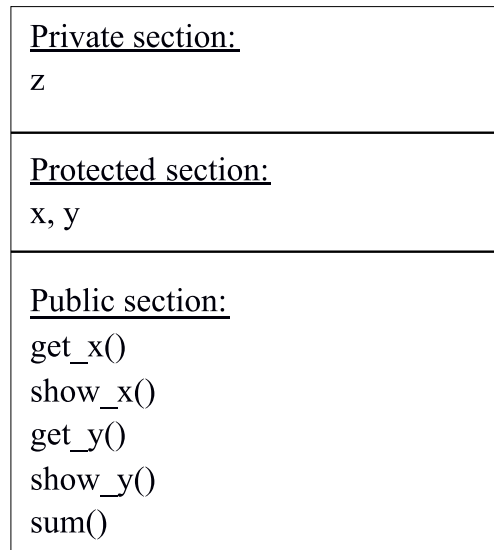


Fig. 12.5 members of derived class addition?

12.5 Multiple Inheritance

When a class inherits the features of two or more classes, is known as multiple inheritance.

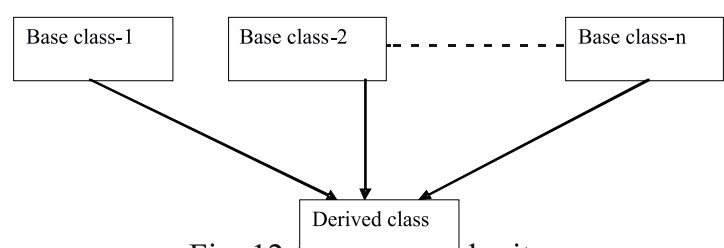


Fig. 12.6 multiple inheritance

The syntax of derived class with multiple base classes is:

```
class derived_class : visibility Base_class-1, visibility Base_class-2, - - - -
- -
{
Members of derived class
};
```

The following program is an example of multiple inheritance.

```
Program 12.3: Multiple inheritance
#include<iostream>
```

```

using namespace std;
class B1
{
protected:
    int x;
public:
    void get_x(int a)
    {
        x=a;
    }
};
class B2
{
protected:
    int y;
public:
    void get_y(int b)
    {
        y=b;
    }
};
class D : public B1, public B2
{
    int z;
public:
    void multiply(void)
    {
        z=x*y;
    }
    void display(void)
    {
        cout<<"x="<<x<<"\n";
        cout<<"y="<<y<<"\n";
        cout<<"z="<<z<<"\n";
    }
};
int main()
{
    D d;
    d.get_x(5);
    d.get_y(3);
    d.multiply();
}

```



```

        d.display();
        return 0;
    }

```

The output of the program 12.3 would be:

```

x=5
y=3
z=15

```

In the above program, the derived class 'D' inherits the members of base classes 'B1' and 'B2'. The derived class 'D' has the following members after this derivation.

Derived class 'D'

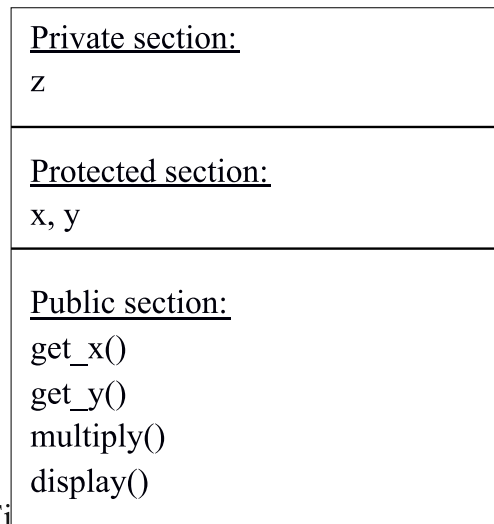


Fig. 12.3

12.6 Hierarchical inheritance

When a base class is inherited by two or more derived classes, it is known as hierarchical inheritance. As an example figure 12.8 shows hierarchical classification of persons in a school.

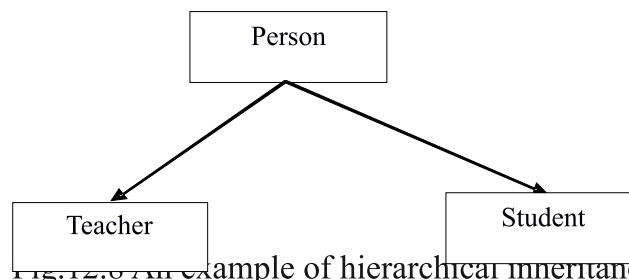


Fig. 12.8 An example of hierarchical inheritance
Program 12.4: Hierarchical inheritance

```

#include<iostream>
#include<string.h>
using namespace std;
class person
{
protected:
    char name[20];
    int age;
public:
    void get_person(const char *n, int a)
    {
        strcpy(name,n);
        age=a;
    }
    void show_person(void)
    {
        cout<<"Name:"<<name<<"\n";
        cout<<"Age:"<<age<<"\n";
    }
};
class teacher : public person
{
    char post[10];
public:
    void get_post(const char *p)
    {
        strcpy(post,p);
    }
    void show_teacher(void)
    {
        show_person();
        cout<<"post:"<<post<<"\n";
    }
};
class student : public person
{
    int standard;

```

```

public:
    void get_standard(int s)
    {
        standard=s;
    }
    void show_student(void)
    {
        show_person();
        cout<<"Standard:"<<standard<<"\n";
    }
};

int main()
{
    teacher t;
    t.get_person("Ram",30);
    t.get_post("TGT");
    student s;
    s.get_person("Shyam",17);
    s.get_standard(12);
    t.show_teacher();
    s.show_student();
    return 0;
}

```

The output of the program 12.4 would be:

Name: Ram

Age: 30

Post: TGT

Name: Shyam

Age: 17

Standard: 12

12.7 Hybrid inheritance

The combination of two or more forms of inheritance is known as hybrid inheritance. As an example figure 12.9 shows hybrid inheritance which is the combination of multilevel inheritance and multiple inheritance.

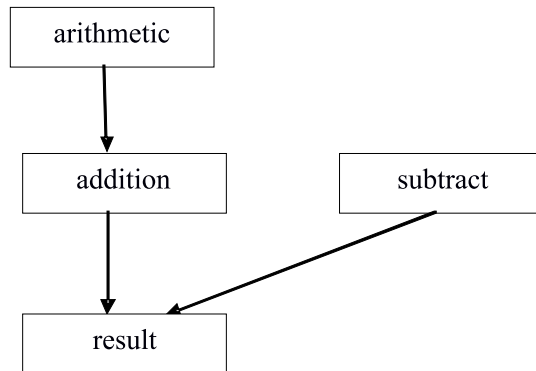


Fig.12.9 An example of hybrid inheritance

Program 12.5: Hybrid inheritance

```

#include<iostream>
using namespace std;
class arithmetic
{
protected:
    int num1, num2;
public:
    void getdata(void)
    {
        cout<<"For Addition:";
        cout<<"\nEnter the first number: ";
        cin>>num1;
        cout<<"\nEnter the second number: ";
        cin>>num2;
    }
};

class addition:public arithmetic
{
protected:
    int sum;
public:
    void add(void)
    {
        sum=num1+num2;
    }
};

class subtract
{

```

```

protected:
    int n1,n2,diff;
public:
    void sub(void)
    {
        cout<<"\nFor Subtraction:";
        cout<<"\nEnter the first number:";
        cin>>n1;
        cout<<"\nEnter the second number:";
        cin>>n2;
        diff=n1-n2;
    }
};
class result:public addition, public subtract
{
public:
    void display(void)
    {
        cout<<"\nSum of "<<num1<<" and "<<num2<<" = "<<sum;
        cout<<"\nDifference of "<<n1<<" and "<<n2<<" = "<<diff;
    }
};

int main()
{
    result z;
    z.getdata();
    z.add();
    z.sub();
    z.display();
    return 0;
}

```

The output of the program 12.5 would be:

For Addition:

Enter the first number: 5

Enter the second number: 7

For Subtraction:

Enter the first number: 10

Enter the second number: 3

Sum of 5 and 7 is 12

Difference of 10 and 3 is 7

12.8 Virtual Base classes

Consider a hybrid inheritance in which three forms of inheritance which are multilevel, multiple and hierarchical inheritance are involved. This is shown in figure 12.10 . The class 'TA'(Teacher Assistant) has two direct base classes 'teacher' and 'student' which have a common base class 'person'. The class 'TA' inherits the features of 'person' via two different paths. This situation creates a problem that all public and protected members of 'person' are inherited into 'TA' twice, first via 'teacher' and second via 'student'. This form of inheritance creates an ambiguity and should be avoided.

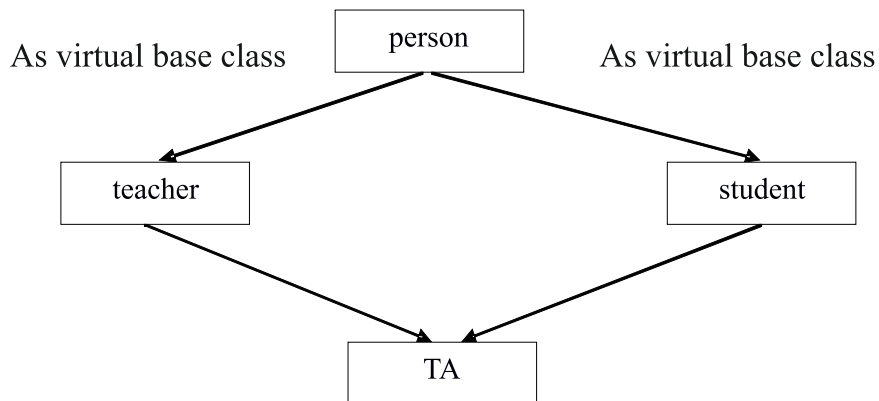


Fig.12.10 Virtual base class

This ambiguity can be resolved by making the common base class as virtual base class while declaring the direct base classes as shown in below.

```

class person
{
  -----
  -----
};
class teacher : virtual public person
{
  -----
  -----
};
class student : virtual public person
{
  -----
  -----
};
class TA: public teacher, public student
  
```

```

{
    -----
    -----
};

```

When a class is declared as a virtual base class, only one copy of the all public and protected members of that class is inherited.

12.9 Abstract classes

If same function name is used in both base and derived classes, the function in base class is declared as virtual that is called a virtual function. A virtual function with empty body is called pure virtual function. A class having at least one of its member functions as pure virtual function is known as abstract class. It is not used to create objects. It is used only to act as a base class to be inherited by other classes. The pure virtual function must be defined by the class which is derived from the abstract base class. The following program is an example of abstract base class.

Program 12.6: Abstract base class

```

#include <iostream>
using namespace std;
class Shape
{
protected:
    int width;
    int height;
public:
    virtual int area() = 0;    // pure virtual function
    void getdata(int w, int h)
    {
        width=w;
        height=h;
    }
};

class Rectangle: public Shape
{
public:
    int area()

```

```

    {
        return (width * height);
    }
};
class Triangle: public Shape
{
public:
    int area() {
        return (width * height)/2;
    }
};
int main(void)
{
    Rectangle Rect;
    Triangle Tri;
    Rect.getdata(5,7);
    cout << "Area of Rectangle : " << Rect.area() << "\n";
    Tri.getdata(6,7);
    cout << "Area of Triangle : " << Tri.area() << "\n";
    return 0;
}

```

The output of the program 12.6 would be:

Area of Rectangle: 35

Area of Triangle: 21

Important Points

- The existing classes are used to create new classes, this mechanism is called inheritance.
- By default, the visibility-mode is private.
- In single inheritance, there is one base class and one derived class.
- There can be any number of levels in multilevel inheritance.
- When a class inherits the features of two or more classes, is known as multiple inheritance.
- When a base class is inherited by two or more derived classes, is known as hierarchical inheritance.
- The combination of two or more forms of inheritance is known as hybrid inheritance.
- When a class is declared as a virtual base class, only one copy of the all public and protected members of that class is inherited.

- A class having at least one of its member functions as pure virtual function is known as abstract class.

Practice Questions

Objective type questions:

Q.1 In inheritance, the existing class is called

- | | |
|----------------|-----------------|
| A. Base class | B. Parent class |
| C. Super class | D. All of these |

Q.2 In inheritance, the new class is called

- | | |
|------------------|-----------------|
| A. Derived class | B. child class |
| C. Subclass | D. All of these |

Q.3 By default, the visibility-mode is

- | | |
|--------------|------------------|
| A. Public | B. Private |
| C. Protected | D. None of these |

Q.4 When there is one base class and one derived class is called

- | | |
|-------------------------|-----------------------------|
| A. Single inheritance | B. Multilevel inheritance |
| C. Multiple inheritance | D. Hierarchical inheritance |

Q.5 When a class inherits the features of two or more classes is called

- | | |
|-------------------------|-----------------------------|
| A. Single inheritance | B. Multilevel inheritance |
| C. Multiple inheritance | D. Hierarchical inheritance |

Very Short Answer Type Questions

- Q.1 What is inheritance?
 Q.2 What is single inheritance?
 Q.3 What is multilevel inheritance?
 Q.4 What is multiple inheritance?
 Q.5 What is hierarchical inheritance?
 Q.6 What is hybrid inheritance?
 Q.7 What is abstract class?

Short Answer Type Questions

- Q.1 Explain the effect of visibility-mode in inheritance.
 Q.2 What is the concept of virtual base class?

Essay Type Questions

Q.1 Write a program to create an abstract class 'shape' consist of a pure virtual function 'volume'. The 'shape' class is inherited by three classes called 'cone', 'cylinder' and 'cube', these derived classes define the pure virtual function 'volume' to compute the volume.

Answer Key

1. D 2. D 3. B 4. A 5. C

Chapter 13

DBMS concepts

Introduction to file system

An abstraction to store, retrieve and update a set of files is called, file system. A file system also includes the data structures specified by some of those abstractions. These data structures are designed to organize multiple files as a single stream of bytes. In a file system other abstractions also specified some network protocols, these are designed to allow files access on a remote machine.

The file system manages access to the data and the metadata of the files. A file system ensures the reliability and it is the major responsibility of the system.

File system disadvantages (problems):

- **Data redundancy:** Same information available in many files e.g. student address in different files for different purposes.
 - **Data Access difficulty:** It requires new program when new request arrives that is, each time new program has to write to full fill coming request because program was not available for new request.
 - **Data is isolated:** in different files which are also in different formats. Multiple users can not access the same data simultaneously because there was difficulty in supervision for concurrent request.
 - **Difficulties with security enforcement.**
- Integrity issues:** Constraints must be ensured on database.

Advantages of file system:

- Easy design with single-application.
- A single application based optimized organization.
- Efficient in term of performance.

HIERARCHY OF DATA

Data stored in computer systems can be view in following manner and we can define database management system after that.

Data are logically organized into

1. Bits (characters)
2. Fields
3. Records
4. Files

5. Databases

Bit-a bit is the smallest unit of data representation (value of a bit may be a 0 or 1)

Field - a field consists of a grouping of characters. A data field represents an attribute (a characteristic or quality) of some entity (object, person, place, or event).

Record - a record represents a collection of attributes that describe a real-world entity. A record consists of fields, with each field describing an attribute of the entity.

File - a group of related records. A **primary key** in a file is the field (or fields) whose value identifies a record among others in a data file.

Now we can define database management system. As the name suggests, the database management system is made of Database and Management System.

DATABASE: To find out what database is, we have to start from data, which is the basic building block of any DBMS.

Data: Facts, figures, statistics etc. having no particular meaning (e.g. 2, XYZ, 19 etc).

Record: Collection of related data items.

File - a group of related records

Database: Collection of interrelated data or data files or relation(for relational database) . This collection of data is interrelated so it can be a relevant information of an organization and this collection of information can be accessed using a set of application program.

Management system: A management system is a set of rules and procedures which help us to create, organize and manipulate the database. It also helps us to add, modify and delete data items in the database.

DBMS

A **database-management system (DBMS)** is a collection of interrelated data and a set of programs to access those data.

Goals of DBMS: Design of any DBMS system has following goals

1. Main goal of any DBMS system is to manage large bodies of information.
2. Provide convenient way to store information in database.
3. Efficient retrieval of information from databases.
4. Safety and security of information stored in database.
5. Avoiding the anomalies during simultaneous access of information by many

users.

Advantages of DBMS: Over conventional file system DBMS has many advantages. These advantages make DBMS more useful in many applications. Following are the advantages of DBMS.

1. **Removing the data redundancy(duplicacy):** If same information is stored in many places it will waste storage spaces and effort . This redundancy problem is handled in DBMS.

2. **Sharing of Data:** But in computerized DBMS, many users can share the same database.

3. **Data Integrity:** We can maintain data integrity by specifying integrity constrains, which are rules and restrictions about what kind of data may be entered or manipulated within the database. This increases the reliability of the database as it can be guaranteed that no wrong data can exist within the database at any point of time.

4. **Data independence:** Application programs should be as independent as possible from details of data representation and storage. The DBMS can provide an abstract view of the data to insulate application code from such details.

5. **Efficient data access:** A DBMS utilizes a variety of sophisticated techniques to store and retrieve data efficiently. This feature is especially important if the data is stored on external storage devices.

6. **Data integrity and security:** Security is ensured by providing abstract view of data and integrity constraints. DBMS provides abstract view so that no need to see all kind of information by all kind of users that is , particular user can see only the part of database .

7. **Reduced application development time:** Clearly, the DBMS supports many important functions that are common to many applications accessing data stored in the DBMS. This, in Conjunction with the high-level interface to the data, facilitates quick development of applications.

8. **Recovery in DBMS:** During transaction failure your database will be restore in its original state

Applications of DBMS: In almost all area's DBMS has its applications. Some of these are

- Banking: all transactions of banking sector

- Airline: reservation, schedules, availability
- Universities: registration, grades
- Sales: customers, products, purchases
- Manufacturing: production, inventory, orders, supply chain
- Human resources: employee records, salaries, tax deductions

Example of DBMS: There are numbers of DBMS which is currently in use

Commercial DBMS:

Company	Product
Oracle	8i, 9i, 10g
IBM	DB2, Universal Server
Microsoft	Access, SQL server
Sybase	Adaptive Server
Informix	Dynamic server

Along with commercial DBMS, one of the widely used Open source DBMS is MySQL.

Abstraction levels in a DBMS

As we have discussed one of the main goal of any DBMS is to simplify the user's interaction with the database that is, any kind of users(naive, programmers, sophisticated etc.) can easily and efficiently retrieve information from database. Abstract view of data helps to hide certain details of how data is stored and maintained.

Physical level: The physical schema specifies how the relations are actually stored in secondary storage devices. It also specifies auxiliary data structures (indexes) used to speed up the access to the relations.

Logical level: The conceptual schema describes the data stored in the database and relationships among those data that is, conceptual schema defines the logical structure of entire database. For example, in a relational database it describes all the relations stored in the database.

View level: The View level is a refinement of the conceptual level. It allows customized and authorized access to individual users or groups of users. Every database has one conceptual and one physical schema, but it can have many schemas at view level. A view(external schema) is conceptually a relation, but its records are not stored in the database instead, they are computed from other

relations. Figure 1 shows the relationship between these levels of abstraction.

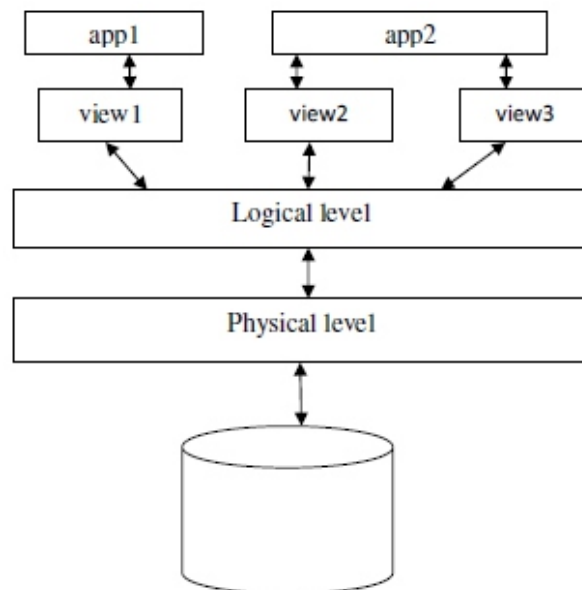


Figure 1. Data abstraction levels

Example school database: Above three schemas depicted in figure 1 can be understood by using school database example.

Example physical schema: Relations stored as unordered files and index on first column of Students.

Example conceptual schema:

Students(Roll_no: int, name: varchar, address: varchar, age: integer, class: char)

Subjects(subjectid: char, Sname: char,)

admission(Roll_no: int, ClassName:char, AdmissionDate: date)

Example external Schema (View):

Class_info(ClassName:char, Strength:integer)

Schemas and instances

A schema is a description (over all design) of the data in terms of the data model. We can say about a schema (metadata) that, a schema is a specification of how data is to be structured logically and rarely changes.

In the relational model the schema looks like:

RelationName(field1 : type1, ..., fieldn : typen)

Students(Roll_no : int, name : char, age : integer, class : char)

An instance on the other hand represents the contents of schema at any particular moment of time which changes rapidly, but always conforms to the

schema. we can compare schema and instances with type and objects of type in a programming language. An instance of the student relation, can be represented as depicted in figure 2.

Student table

Roll_no	Name	Age	Address	class
101	Harish	10	Ajmer	5th
105	kailash	20	kota	10th
109	Manish	18	Ahmadabad	9th
120	Ronak	14	Udaipur	8th
135	shanker	13	jaipur	7th

Database languages: A database system has, Data Definition Language (DDL) to specifies the data schemas and data Manipulation Language (DML) to facilitate the retrieval and update of data in the database. DML are basically of two types.

1. **Procedural DML:** It requires that the data and the procedure to obtained those data must be specified by the user.

2. **Non-procedural DML:** In non-procedural DML only required data is specified by the user without specifying the procedure to obtain those data. A DBMS provides a specialized language, called **query** language to ask questions to the DBMS. For retrieval, we query the database with the query language, which is part of the DML. Term query language and DML are synonymous.

Classification of DBMS: A DBMS system can be of several types based on following criteria.

1. **Based on users:** A first criteria is the number of users supported by the system. **Single-user systems** support only one user at a time and are mostly used with personal computers. **Multuser systems**, which include the majority of DBMS, support multiple users concurrently.

2. **Based on Architecture:** A second criteria is the number of computer system on which database system runs. A **centralized or client-server** DBMS can support multiple users, but the DBMS and the database themselves reside and runs totally at a single computer site(server machine). A **distributed**

DBMS (DDBMS) can have the actual database and DBMS software distributed over many sites, connected by a computer network. Homogeneous DDBMSs use the same DBMS software at multiple sites.

3. Based on types of data models: Third criteria is the types of data models on which the DBMS is based. These DBMS are can be of following types.

- Hierarchical databases.
- Network databases.
- Relational databases.
- Object-oriented databases

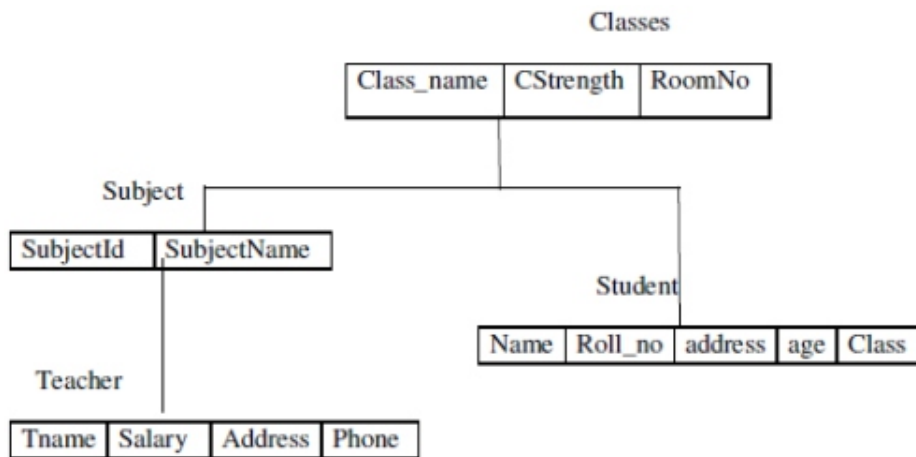
Data model:

A data model is a collection of concepts for describing data, the meaning of data, data constraints and data relationships. Some of the data models are Hierarchical, Network, Relational and Object-oriented.

Hierarchical data model :

We can find older systems that are based on a hierarchical model .The first hierarchical DBMS is “IMS” and it was released in 1968. The hierarchical DBMS is used to model one-to-many relationships, presenting data to users in a treelike structure. Within each record, data elements are organized into pieces of records called segments. To the user, each record looks like an organizational chart with one toplevel segment called the root. An upper segment is connected logically to a lower segment in a parent–child relationship. A parent segment can have more than one child, but a child can have only one parent. Figure 1 shows a hierarchical structure that might be used for school management system.

The root segment is classes, which contains basic classe's information such as name, strength, and room number. Immediately below it are two child segments: subjects (containing subject id and subject name data), student containing (name , address, rollno , age and class data). The subject segment has one child below it: teacher (containing data about teacher name , salary, address, phone and results evaluations) .



It is found that in large legacy systems where high volume transaction processing is required, hierarchical DBMS can still be used. Banks, insurance companies, and other high-volume users continue to use reliable hierarchical databases,

Network data model: A network DBMS depicts data logically as many-to-many relationships. In other words, parents can have multiple children, and a child can have more than one parent. A typical many-to-many relationship for a network DBMS is the student–subject relationship (see Figure 4 below). There are many subjects and many students in a class. A student takes many subjects, and a subject has many students.

Hierarchical and network DBMS are considered outdated and are no longer used for building new database applications.

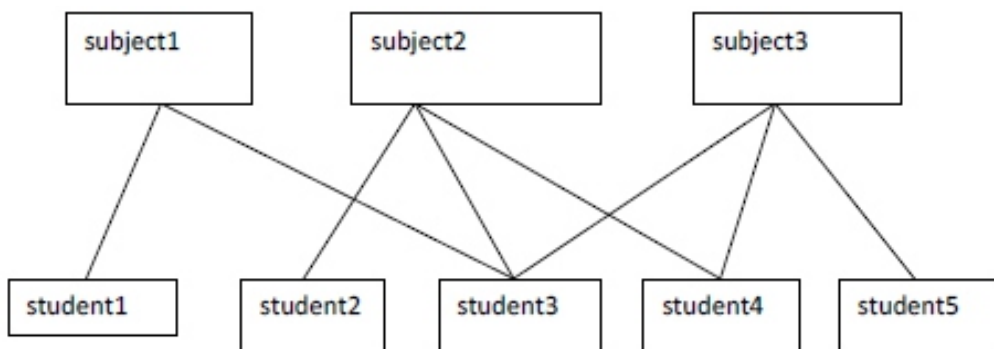


Figure 4 Network model for student subject relationship

Relational data model:

This is a record based data model. It uses a collection of relations (or tables) to represent data and relationship between data. Each relation has a list of

attributes (or columns) which has a unique name. Each attribute has a domain (or type). Each relation contains a set of tuples (or rows). Each tuple has a value for each attribute of the relation. Duplicate tuples are not allowed. It is the data model used by mostly current database systems. Given below is an example of student table showing students details.

Example

Student table

Roll_no	Name	Age	Address	Class
101	Harish	10	Ajmer	5th
105	Kailash	20	Kota	10th
109	Manish	18	Ahmadabad	9th
120	Ronak	14	Udaipur	8th
135	Shanker	13	Jaipur	7th

Figure 5: Relational data model for student database

Database design steps or phase

Designing a DBMS application for any enterprise should follow some phases.



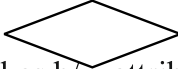
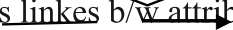
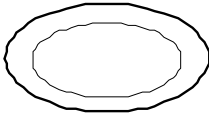

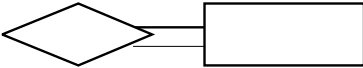
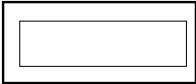
- **Requirements analysis:** In this phase data needs of an organization is identified.
- **Conceptual Database design):** Mostly done using the ER model. Concept of chosen data model is applied on identified data to construct the conceptual schema.
- **Logical Database design:** In this phase high level conceptual schema maps onto the implementation data model of the database system that will be used e.g. for RDBMS it is relational model.
- **Schema refinement:** Normalization is applied to refine schema into smaller schema
- **Physical Database Design:** This phase specified the physical features of the database e.g. internal storage structure, form of file organization etc.

E-R model and E-R diagram

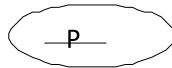
E-R model:

Entity-Relationship (ER) model is a popular conceptual data model. The model describes data to be stored and the constraints over the data. E-R model views the real world as a collection of **entities** and **relationships** among entities. An entity is an “object” in real world that can be differentiable from other objects.

E-R diagram: It is basically graphical representation of overall logical structure of a database. The main components in this diagram are as follows.

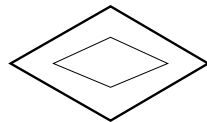
Symbol name	Symbol	Purpose
Rectangles		Represent entity set
Ellipses		Represents attributes
Diamonds		Represents relationships
Lines		Represents links b/w attributes to entity set and entity sets to relationship set
Double ellipses		Multivalued attributes
Dashed ellipses		Represents derived attributes
Double lines		total participation
Double rectangles		Represents weak entity sets

Ellipses with line



Primary key

Double diamonds



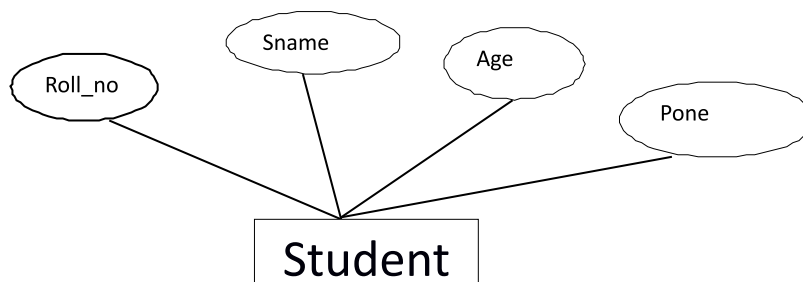
Identifying relationship for weak
entity set

Entity: is an object in the real world that is distinguishable from all other objects. E.g., A class, A teacher, The address of the teacher, a student, a subject. An entity is described using a set of attributes. Each attribute has a domain of possible values.

Entity Set: An entity set is a collection of entities of similar objects(same type). In an entity set values of attributes are used to distinguish one entity from another of same type. For each entity set, we should identify a key. A key is a minimal set of attributes that uniquely identify an entity in a set. All entries in a given entity set have the same attributes (the values may be different).

Representation of entity set and attributes in E-R diagram: An entity set can be represented as a rectangle in E-R diagram.

Graphical representation in E-R diagram: Example of student entity

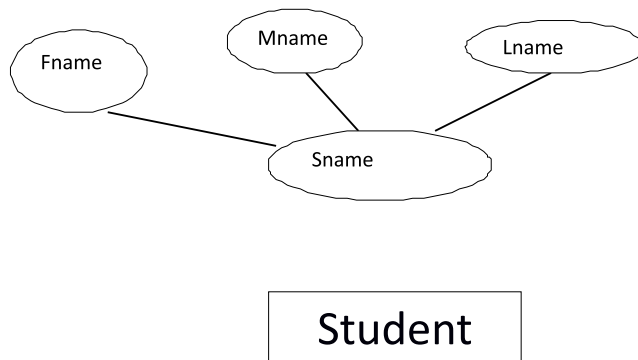


Different attribute types:

Attributes are properties of entities and relationships like, attributes of tuples or objects. Attributes can be of following types and represented as ovals in E-R diagram.

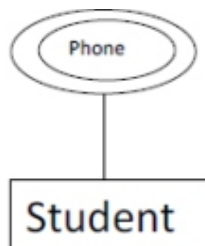
- **Simple attribute:** This kind of attributes contains a single value e.g. in above student entity Roll_no and age are simple attributes.
- **Composite attribute:** This kind of attributes contains several components e.g. Sname attributes contains first name, middle name, last name components.

Graphical representation in E-R diagram: Example of student entity and its attribute Sname.



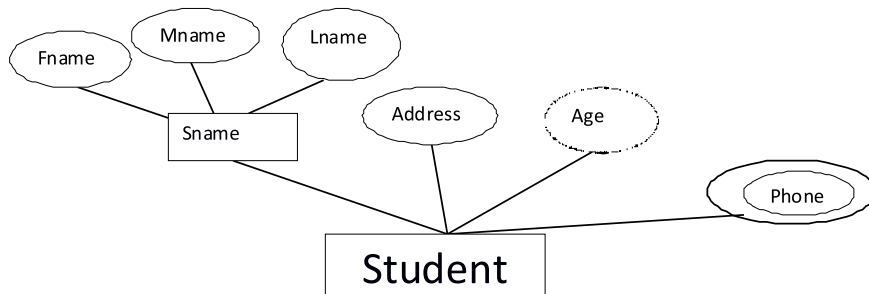
- **Multi-valued attribute:** This kind of attributes contains more than one value e.g. phone attributes contains many mobiles numbers, land line number, office number.

Graphical representation in E-R diagram: Example of student entity and its phone attribute.



- **Derived attribute:** Value of this kind of attributes can be computed from other attributes e.g., age can be computed from data of birth and the current date.

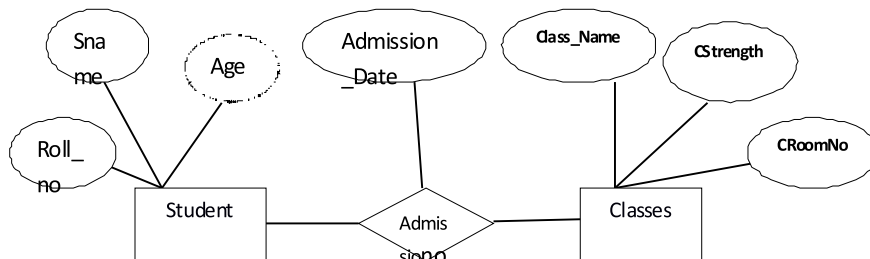
Graphical representation in E-R diagram: Example of student entity and its age attribute along with multi valued and composite attribute.



Relationship: A relationship is an association among two or more entities. Relationship that involve two entity sets are called binary (or degree two) relationship.

Relationship set: A set of relationships of the same type (among same entity sets) e.g. student admission in classes. Here admission is a relationship between student and classes entity sets. It can be represented as a diamond in E-R diagram.

Graphical representation in E-R diagram:



Attributes of relationships: A relationship can also include its own attributes (called descriptive attributes). For example assume that students take admission in particular class on a particular date.

So where do the admission date go? Following are two possibilities here

With Student ?

1. But a student can have different admission dates for different classes.

With classes ?

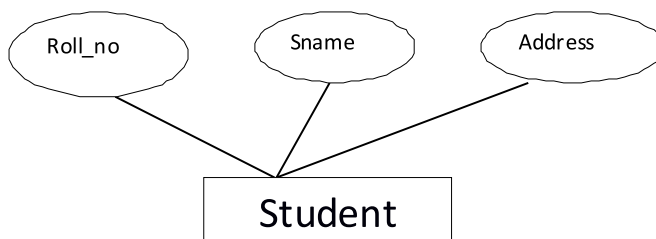
2. But a class can assign different admission date for multiple students. So it will go with admission as shown in above relationship example of E-R diagram..

Constraints:

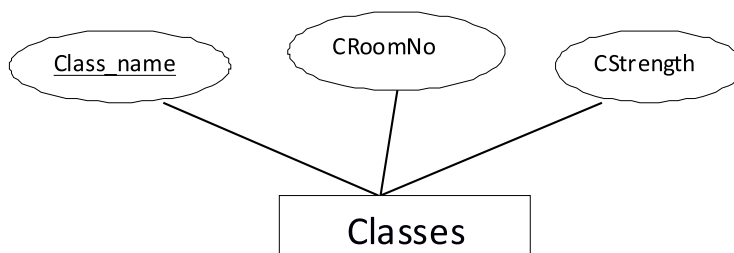
E-R model describes data to be stored and also the constraints over the data. These constraints are mapping cardinality, key constraints and participation constraints.

Key constraints: A key is a set of attributes whose values can belong to at most one entity in an entity set that is, a set of attributes that can uniquely identify an entity e.g. Roll_no is a key of student entity set. A key of an entity set is represented by underlining all attributes in the key.

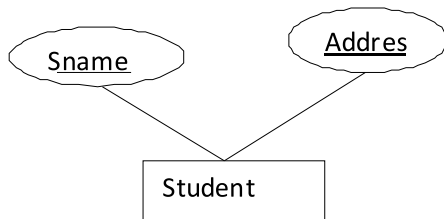
Graphical representation in E-R diagram: Example of Roll_no attribute in student entity.



E.g2. In classes entity set Class_Name is the primary key which is unique because we are assuming that 12 different classes from 1st to 12th exist in a school without any sections.



- Composite Key: Two or more attributes are used to serve as a key e.g. Name or Address alone cannot uniquely identify a student, but together they can identify a student.



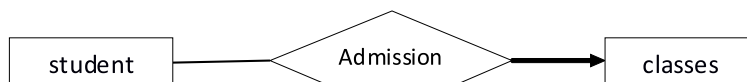
- **Candidate key:** A minimal set of attributes that uniquely identifies an entity is called a candidate key. e.g. {Roll_no} and {Sname, Address} both are two candidate keys but {Roll_no, Sname} is not a candidate key. If there are many candidate keys, we should choose one candidate key as the primary key.

Mapping cardinality constraints: Mapping cardinality between two entity sets e1 and e2 for binary relationship R can be of following types.

- **Many to one:** Each entity in e1 is related to 0 or 1 entity in e2, but each entity in e2 is related to 0 or more in e1.

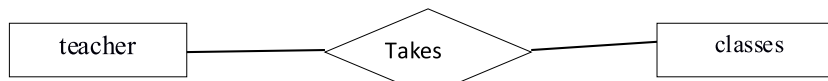
Example: One student can be in one class at a time but one class can have many students.

Graphical representation in E-R diagram: Using lines.



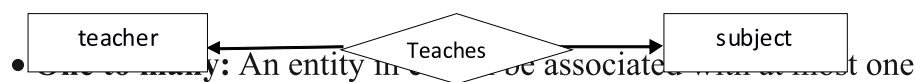
- **Many to many:** Each entity in e1 is related to 0 or more entities in e2 and vice versa.

Example: Many teachers can teach one class or many classes can be taught by a single teacher.



- **One to one:** Each entity in e1 is related to 0 or 1 entity in e2 and vice versa.

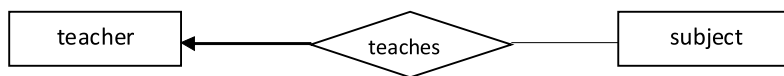
Example: A teacher can only teach one subject e.g. Hindi , English , Maths . It is assuming that in school particular subject teacher is fixed.



- **One to one:** An entity in e1 is associated with at most one entity in e2 and vice versa.

entity in e1.

Example: A teacher can also teach many subject in special case if teacher of particular subject is not available in school.



Participation constraints: Diagram below shows the participation of an entity set's all entities in a relationship. There are two types of participation constraints for an entity in a relationship.

- **Total:** Every instance of the entity is present in the relationship (represent it by a thick line).

Example: In this example participation of student entity is total because every student must take admission in one class.

Graphical representation in E-R diagram: Using double lines.

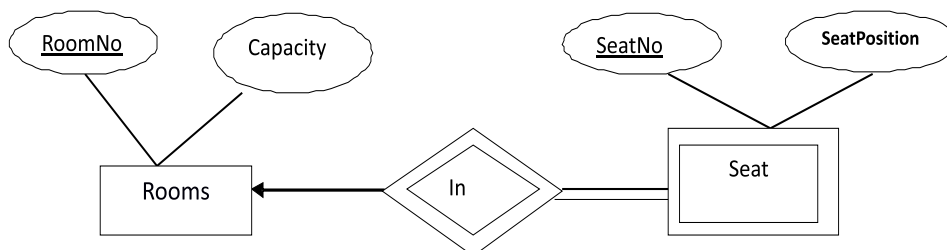


- **Partial:** Not every instance of the entity is present in the relationship.

Example: Participation of classes is **partial** if some classes do not have student admission.

Weak Entities: Sometimes, the key of an entity set E can not be completely formed by its own attributes, but from the keys of other (one or more) entity sets to which E is linked by many to one (or one to one) relationship sets E can be completely formed that is, An entity cannot be uniquely identified by all attributes related to this entity. Such entity is called weak entity.

Example: Suppose each room in school can have seat numbers for each student. Attributes of seats are SeatNo and SeatPosition. These attributes can not uniquely identified a seat entity. So seat is an weak entity.



Weak entity: can be represented graphically in E-R diagram using double diamond. A weak entity can be identified uniquely only by considering some of its attributes in conjunction with the primary key of another entity (called **identifying or owner entity**) and relating relationship is called **identifying relationship set**.

Database design for a school management system:

Firstly, we assume some of the characteristics of a school.

1. A school have many classes from 1st to 12th.
2. Each class study number of subjects.
3. Many teachers work in a school.
4. A teacher can teach only one subject.
5. Each teacher can take any number of classes with same subject.
6. A student can take dmission in any class.
7. Each class can have any number of students.
8. Each class has its own time table.

How we can start E-R modeling?

Designing of an E-R model can be done by identifying.

- Identify the Entities
- Find relationships
- Identify the key attributes for every Entity
- Identify other relevant attributes
- Draw complete E-R diagram with all attributes including Primary Key

Step 1: Identify the Entities:

- CLASSES
- STUDENT
- SUBJECT
- TEACHER

Step 2: Finding the relationships:

- Many classes contains many subjects and class time table contains

subjects , hence the cardinality between subjects and classes is many to many .

- One class has multiple teachers and one teacher belongs to many classes for same subject , hence the cardinality between classes and teacher is many to Many.
- One class admits multiple students and one student admits in one and only one class.
Hence the cardinality between class and student is one to Many.
- One subject is taught by only one teacher, hence the cardinality between subject and teacher is one to one.
- Many subjects read by multiple students and one student reads multiple subjects in a class, hence the cardinality between subjects and student is Many to Many.

Step 3: Identify the key attributes and other relevant attributes:

- SubjectId is the key attribute and SubjectName is the other attribute for “subject” Entity.
- Roll_no is the key attribute and Sname, Age, Address, phone are other attributes for “student” Entity
- Tname is the key attribute and address, phone, salary are other attributes for “Teacher” Entity.
- Class_name is the key attribute and CStrength, CRoomNo are other attributes for the Entity “classes”.
- CRoomNo is the key attribute and PeriodNo is other attributes for the Entity “timetable”.

Complete E-R diagram of the school database

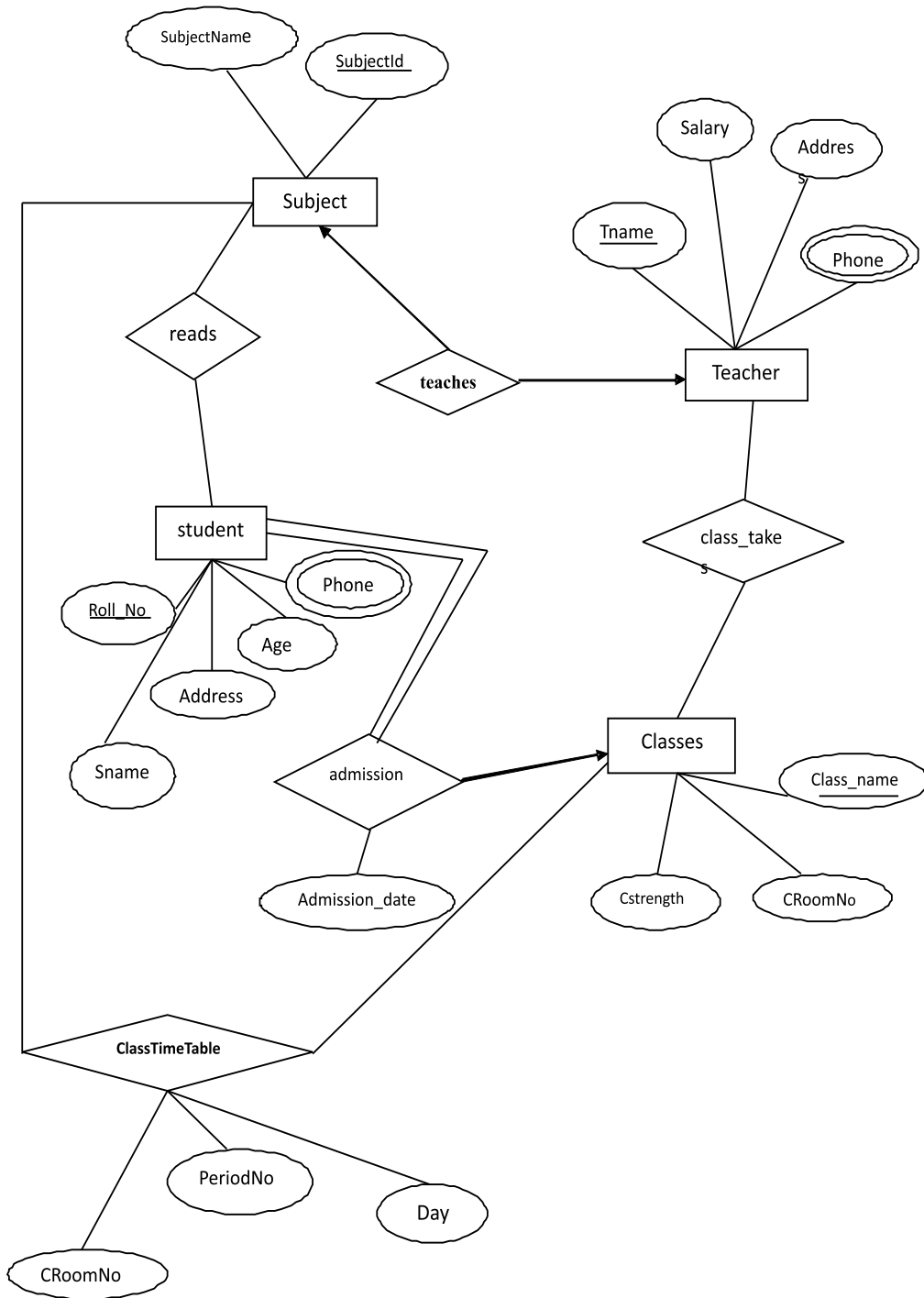


Figure 6. E-R diagram of a school from 1st to 12th

Translating E-R into Relational schema: Given E-R diagram in Fig 6

can be converted into relational design. This mapping can be done by some steps.

1. Translating entity sets: For entity set all its attributes will form columns of table and its key attribute will be key column that is,

- Attributes → columns
- Key attributes → key columns

So converted schema's of school E-R Diagram are.

Student(Roll_No, Sname, age, Address, Phone)

Classes(Class_name, Cstrength, CRoomNo)

Teacher(Tname, Salary, Address, Phone)

Subject(SubjectId, SubjectName)

2. Translating relationship sets:

A relationship set will also be translated into a table as.

- Keys of connected entity sets will be the columns of table.
- Attributes of the relationship set (if any) will be columns of table.
- Foreign key in relationship set will be primary key of participating entity sets.
- Multiplicity of the relationship set determines the key of the table and will be as follows.

For (one-to-one Relationship): Primary key of either entity set can be primary key of relationship.

For (one-to-many Relationship) or many to one: Primary key of the entity set on the “many “ side of the relationship set will be primary key of relationship set.

For (Many-to-many Relationship): Combination of primary keys of participating entity sets will form the primary key of relationship.

So converted schema's of school diagram for relationships are.

Class_takes(Class_name, Tname)

Admission(Roll_No, Class_name, Admission_date)

Teaches(SubjectId, Tname)

Reads(Roll_No, SubjectId)

ClassTimeTable(Class_name, SubjectId, PeriodNo, CRoomNo, PeriodNo)

After merging resultant schema's of school database:

Student(Roll_No , Sname, age, Address, Phone, Class_name, admission_date)
 Classes(Class_name, Cstrength, CRoomNo)
 Teacher(Tname, Salary, Address, Phone)
 Subject(SubjectId, SubjectName)
 Class_takes(Class_name, Tname)
 Teaches(SubjectId, Tname)
 Reads(Roll_No, SubjectId)
 ClassTimeTable(Class_name, SubjectId, PeriodNo, CRoomNo, PeriodNo)

Introduction to normalization

One more method for designing a relational database is to use a process commonly named as normalization. The main purpose is to generate a set of relation schemas that allows us to store information without unnecessary redundancy, along with that also allows us to retrieve information easily. So to do all design, design the schemas in appropriate normal forms using the concepts of functional dependencies.

Bad database and purpose of normalization: To understand the concept of bad database and purpose of normalization we consider following student table.

Student

Roll_no	Name	Age	Address	Phone	class	Subject
101	Harish	10	Ajmer	1234567891	5th	Hindi
101	Harish	10	Ajmer	1234567891	5th	Math's
101	Harish	10	Ajmer	1234567891	5th	Sanskrit
120	Ronak	14	Udaipur	2222222222	8th	Hindi
120	Ronak	14	Udaipur	2222222222	8th	Sanskrit

By analyzing the above schema we can easily find that this design is not a good database design.

Why this design is bad?

For student reads subjects(Roll_no, name, age, address, class, subject) This design has redundancy, because the name of a student, age, address, class is recorded multiple times, once for each subjects the student is reading. This

redundancy caused serious problems in design that is, it wastes storage space and introduce potential inconsistency in database. That is why this database design is bad.

A bad database design also caused many problems(Anomalies) during operation on database.

Update anomalies: all repeated data needs to be updated when one copy is updated. For example if we want to update the address of a particular student we have to update all tuples of that student.

Insertion anomalies: It might not be possible to store certain data unless some other, unrelated information is also stored. We need to know the phone in order to insert a tuple .This could be fixed with a NULL value but nulls also caused problems or hard to handle.

Deletion anomalies: It may not be possible to delete certain information without losing some other, unrelated information that is if we delete all the tuples with a given (class, Roll_no) we might lose that association. So if we want to detect and remove redundancy in designs we need a systematic approach. In another word, if we want to design a good database then we have to normalized it by using Dependencies, decompositions and normal forms.

Functional dependencies:

A functional dependency (FD) is a kind of IC that generalizes the concept of a key. Let R be a relation schema, with X and Y be nonempty sets of attributes in R. For an instance r of R, we say that the FD

$X \rightarrow Y$ (X functionally determines Y)

is satisfied if: $\forall t_1, t_2 \in r, t_1.X = t_2.X \Rightarrow t_1.Y = t_2.Y$

$X \rightarrow Y$ means that whenever two tuples in R agree on all the attributes in X, they must also agree on all attributes in Y

Example of a Functional Dependency:

The functional dependency $AB \rightarrow C$: satisfies for following instance

A	B	C	D
a1	b1	c1	d1
a1	b1	c1	d2
a1	b2	c2	d1
a2	b1	c3	d1

- A primary key is a special case of a FD: If $X \rightarrow Y$ (holds, where Y is the set of all attributes, then X is a superkey
- FDs should hold for any instance of a relation.
- Given a set of FDs we can usually find additional FDs that also hold.

Example: Given a key we can always find a superkey.

Redefining keys using FDs:

A set of attributes K is a key for a relation R if $K \rightarrow$ all (other) attributes of R that is, K is a “super key”. No proper subset of K satisfies the above condition that is, K is minimal.

Normalization

Normalization is a process of organizing the data in database to avoid data redundancy, insertion anomaly, update anomaly & deletion anomaly. In this process we check a given relation schema against certain normal form to check whether or not it satisfies certain normal form. If a relation schema does not satisfy certain normal form then we decompose it into smaller schemas.

Normalization is used for mainly two purposes,

- Eliminating redundant (useless) data.
- Ensuring data dependencies make sense i.e data is logically stored.

Relational database design: To design a relational database ,we need to know for a given schema that it is a good design. If design is not good then we decompose it into smaller schemas but the decomposition should be good. After that we check each decomposed schema for certain normal forms. If a given relation schema is in a normal form then we know some problems cannot arise.

Normal forms: Various normal forms are

1. First Normal Form(1NF)
2. Second Normal Form(2NF)
3. Third Normal Form(3NF)
4. BCNF

First Normal Form(1NF): A relation is in first normal form if every field contains only atomic values (no lists nor sets).

Example:

Student table given below in figure 7 is not in 1NF but in figure 8 is in 1NF

Student

Roll_no	Name	Age	Subject
101	Harish	10	hindi,maths
120	Ronak	14	maths

Figure 7. Student table showing subjects of students not in 1NF

Student

Roll_no	Name	Age	Subject
101	Harish	10	Hindi
101	Harish	10	Maths
101	Harish	10	Sanskrit
120	Ronak	14	Hindi
120	Ronak	14	Maths

Figure 8. Student table showing subject of student in 1NF

Second normal form

As per the Second Normal Form there must not be any partial dependency of any column on primary key. It means that for a table that has primary key, every non prime attribute in the table should be fully functionally dependent on primary key attribute . If any column depends only on one part of primary key, then the table fails in Second normal form.

Student Reads Subject (Roll_no, subject Id, Sname, address, Subject Name)

In this student subject relation primary key attribute are Roll_no, and subjectId. According to the rule, non-key attributes, i.e. Sname and SubjectName must be dependent upon both and not on any of the prime key attribute individually. But we find that Sname can be identified by Roll_no and SubjectName can be identified by SubjectId independently. This is called partial dependency, which is not allowed in Second Normal Form.

We broke the relation in two as depicted in the below picture. So there exists no partial dependency.

Student(Roll_no, Sname, address)

Subject(SubjectId, SubjectName)

Third Normal Form

If R is a relation schema, F is a set of functional dependencies on R and A is a single attribute in R then to check whether this schema is in 3NF or not, for every FD we check the following conditions. If for any FD, these conditions fails then given schema R will not be in 3NF.

Conditions:

- If any FD is trivial that is, in $\beta A (A \beta)$, *or*
- IF left side attribute in FD is a key for schema *or*
- If right side attribute is *part of* some key(s) for R

BCNF: To check for a relation schema that, whether this schema is in BCNF or not, for every FD we check the following conditions. If for any FD, these conditions fails then given schema R will not be in BCNF.

Conditions:

- If any FD is trivial that is, in $\beta A (A \beta)$, *or*
- IF left side attribute in FD is a key for schema *or*

If any relation is in BCNF then it will be also in 3NF. That is BCNF implies 3NF but 3NF can not implies BCNF.

Design goals

So for a good relational database design, a relational schema should be in BCNF with Lossless- join and dependency preservation. If we cannot achieve this, we accept 3NF with Lossless-join and dependency preservation.

Important Points:

- DBMS is used to maintain and query large datasets.
- Benefits include recovery from system crashes, concurrent access, quick application development, data integrity, security and sharing of Data.
- Levels of abstraction give data independence.
- A DBMS typically has a layered architecture.
- Functional components of DBMS are file manager, Buffer manager, Query processor, Data files, Data dictionary and Indices.
- A DBMS system can be of several types based on criteria, based on users, architecture and types of data models
- Entity: is an object in the real world that is distinguishable from all other objects.

- A minimal set of attributes that uniquely identifies an entity is called a candidate key.
- A bad database design caused many anomalies during operation on database such as update anomalies, insertion anomalies and deletion anomalies.
- A functional dependency (FD) is a kind of IC that generalizes the concept of a key
- If any relation is in BCNF then it will be also in 3NF. That is BCNF implies 3NF but 3NF can not implies BCNF.

Practice Questions

Objective type questions:

- Q1. Which one of the following is not a goal of DBMS.
- a) Managing large information b) Efficient retrieval
c) Preventing concurrent access d) Safety of data
- Q2. Which one of the following is an example of a commercial DBMS.
- a) Oracle b) IBM
c) Sybase d) all
- Q3. Which one of the following is the simplest level of data abstraction.
- a) Physical b) Logical
c) View d) none of these
- Q4. Normalization means
- a) Joining relations b) Decomposition of relation
c) both d) none of these
- Q5. Which normal form is more restricted?
- a) 1NF b) 2NF
c) BCNF d) 3NF

Very short answer type questions:

- Q1 What is DBMS?.
- Q2. Define a record.
- Q3. Give names of different data redundancy.
- Q4. Define database schema.
- Q5. What is the role of indexes in DBMS?
- Q6. What is a query language?
- Q7. Differentiate between procedural and non -procedural DML.

- Q8. Differentiate between schema and instances.
Q9. What are the steps of designing a database?
Q10. What is an entity?

Short answer type questions:

- Q1. What do you understand by atomicity?
Q2. Differentiate between logical and physical data independence.
Q3. What is a weak entity? Draw it in E-R diagram?
Q4. Differentiate between primary and composite key.
Q5. What is a bad database?.

Essay type questions:

- Q1. What are the various components of DBMS? Explain with suitable diagram.
Q2. What is a data model.? Explain hierarchical data model? How it is differ from network data model.
Q3. Explain various types of attributes and relationships in E-R diagram and also give graphical representation for them.
Q4. Design an E-R diagram for a school consisting of different classes from 1st to 10th.
Q5. What is normalization? Give various forms of normalization.

Answers key for objective questions

- Q1: c Q2: d Q3: c Q4: b Q5: c

Chapter 14

SQL

Relational database concepts:

Relational database is a collection of tables. Each table has a unique name and consists of many columns. Each column of a table also has a unique name. The name of relational database is derived from mathematical relation because there is a close correspondence between them.

Table(Relation): In RDBMS mainly data store in a special kind of object called table. In another world, a table is a collection of related entries and consists of rows and columns.

Following is an example of a student table.

Student table

Roll_no	Name	Age	Address	class
101	Harish	10	Ajmer	5th
105	Kailash	20	Kota	10th
109	Manish	18	Ahmadabad	9th
120	Ronak	14	Udaipur	8th
135	Shanker	13	Jaipur	7th

Figure1 The student relation

Field: Field of a table represents its column which is used to store the specific information about a record. For example in above given table student Roll_no, name, address and class are fields of table.

Records: It is also called the row of a table. Basically it is an individual entry of a table available in that table. For example following is the one individual entry of a student table or row of a table or tuple of a table.

Student table

105	kailash	20	kota	10th
-----	---------	----	------	------

Column: A column of a table is that vertical entry of a table which keeps all information related with a specific field. For example Roll_no is a column of a student table which keeps the following information.

Roll_no
101
105
109
120
135

Domain: Domain of a field is the set of values permitted for that field. For example domain of a field name is the set of all names.

Database schema: A schema of a database is the logical design of a database, which is rarely changed. For example following is the schema of student, classes and teacher table.

Student(Roll_no, name, age, address, class)

classes(Class_name, CStrength, CRoomNo)

Teacher(Tname, Address, salary, phone)

Relational Database instance: A collection of data or information stored in a table at any particular moment of time is called the database instance. Given below is an instance of student relation, which can be changed at any moment of time by making a new entry in the table or deleting an entry from table.

Student

RollNo	Name	Age	Address	Class
101	Harish	10	Ajmer	5 th
105	Kailash	20	Kota	10 th
109	Manish	18	Ahmadabad	9 th
120	Ronak	14	Udaipur	8 th
135	Shanker	13	Jaipur	7 th

Primary key: It is a set of one or more attributes (field) of a table which can be used to identify uniquely any row or tuples of that table. Then such set of attribute taken collectively forms a primary key of table. Primary key is also a kind of constraints. For example in above given table in figure 1 student primary key is Roll_no field because in student table using this field all the students can be identified uniquely and also with the help of Roll_no the record of any student can be easily retrieved from student table.

For example suppose the value of Roll_no is 105. If we take this value then the records retrieved from table will be of student “kailash”.

Data constraints: Data constraints are rules on columns of a table to define the limit of data entry, so that the data entries in the table should fixed the

consistency, accuracy and reliability of database and there should not be any kind of data consistency loss in database when changed is made by authorized users of database. Data constraints can be of column level and table level. The main difference between column level and table level constraints is that, column level constraints are for a single column whereas table level constraints are applied on whole table.

Following are the examples of data constraints

- 1) Class of a student can not be null.
- 2) Roll_no of two students can not be same.
- 3) There will be a matching class in classes relation for every class of student relation.

Constraints on a single relation: Following are the constraints on a single relation

- 1) Not null
- 2) Unique
- 3) Check (<predicate>)

Not null constraints: This constraints restricts the entry of null value for a field or attribute in any table that is, if this constraints is ensured for a field and any operation in this table to try to insert the value null by changing then an error will be generated.

For example, you do not want the value null for class attribute of student table. Similarly value of roll_no attribute should not be null because it is a primary key of student table.

Unique constraints: This constraint ensure that any two tuples or rows of any relation can not be same for all primary key attributes that is, for both the tuples values of all the attributes will not be same. Unique constraints form the candidate key of a relation which can be more than one in a relation.

Check constraints: Check constraints can be applied on both, domain and relation declaration. If it is applied on relation declaration then all tuples of relation must fulfill the condition specified by the check clause that is, check clause ensures that all the values of a column must fulfill the condition applied on column.

For example the values of class field attribute in student table will be 1st, 2nd, 3rd, -
---, 9th, 10th, 11th, 12th.

check (class in ('1st', '2nd', '3rd', ----- '9th', '10th', '11th', '12th'))

Entity integrity constraints: Entity integrity ensures that, two rows or records of a table can not be duplicated and also the field which identifies the every records of the table is a unique field. The value of this field will not be null. Entity integrity constraints can be imposed by primary key. If we define primary key for every entity then it fulfills the entity integrity itself.

For example in given student table if the primary key of this table is Roll_no

field then Roll_no of each student in this table will be different and also the value of this field for any student will not be null that is, all will have its own Roll_no. Due to different Roll_no this table cannot have two same rows.

Student table

RollNo	Name	Address	Age	Class
110	Komal	jaipur	17	12th
120	Ronak	Udaipur	14	8th
105	kailash	kota	20	10th
107	hari	chittorgarh	10	5 th

Referential integrity: If we want to ensure that a value that appears in any relation for a given set of attributes also appears for certain set of attributes in another relation that is, both the tables have same values for some of the attributes then this condition is called referential integrity. Referential integrity can be ensured by foreign key constraints.

Foreign key integrity constraint: To understand this constraint we consider the following table as a example. There are two tables student and classes (name of tables) is given and any moment inserted values in the table is also mentioned. The primary key of given student table is Roll_no field whereas primary key of classes table is class_name field.

student

RollNo	Name	Age	Address	Class
101	Harish	10	Ajmer	5 th
105	Kailash	20	Kota	10 th
109	Manish	18	Ahmadabad	9 th
120	Ronak	14	Udaipur	8 th
135	Shanker	13	jaipur	7 th

Classes

Class_name	Class_room	Strength
12 th	F-1	95
10 th	F-2	80
9 th	F-3	70
4 th	F-4	110

Figure 2 Foreign key constraints on student relation

We are assuming that one and only one class of all standard exist such as, one class of 12th, one class of 10th, one class of 9th and so on that is sections of a class does not exists.

The student table among its attributes (Roll_no, address, age, name, class) keeps one attribute(class) which is the primary key of another table (classes).

For example class field of student table is the primary key of classes table. So this attribute class in student table is called the foreign key of student table which will refer to table classes.

The relation student is called the referencing relation whereas classes is called the referenced relation of foreign key. To become foreign key domain type of a foreign key attribute and another relation attribute should be same and number of attributes in foreign key field and another relation must be same that is, should be compatible.

We can ensured following by foreign key

- 1) We can not delete any record from classes table until a matching record is available in related table student.
- 2) We can not change the value of primary key in classes table until a related record of such record is available in another table.
- 3) We can not insert a new value in student table's class field until this value is not available in the primary key field (class_name) of classes table.
- 4) Although you can insert the null value in foreign key field but if we want that this should not reject then we use cascade in SQL.

Introduction of SQL:- SQL is a combination of relational algebra and relational calculus. It's a standard language of relational database management systems which is used to organize, manage the data available and retrieval of data from relational database. It is an important feature of SQL.

Companies like Oracle, IBM, DB2, Sybase and Ingress, use SQL as a standard programming language for their database. Basic version of this is called sequel which is developed by IBM.

Versions of SQL are SQL-86, SQL-89(extended standard), SQL-92 and SQL-1999. Latest version of SQL is SQL-2003.

SQL is not only a query language but also a standard itself which has following types:-

1. Data Definition Language (DDL)
2. Data Manipulation Language (DML)
3. Data Control Language (DCL)

Most of the commercial relational Database like IBM, Oracle, Microsoft, and Sybase etc use SQL. All Databases do not support all features available in different versions. So to use all the features of particular version always follow

manual according to the Database system.

Advantages of SQL:- SQL has many advantages for all the commercial (Oracle, IBM, DB2, Sybase) as well as for open source (MySQL, Postgres) database systems.

1. **High speed:-** SQL is a kind of language which retrieve the data efficiently and quickly from a database of a big organization. So SQL is a fast language.
2. **Easy to learn:-** It is easy to learn because of short size of programming code that is, there is no need to write many lines of code.
3. **Well defined standard:-** SQL is a standard language which is standardized by ANSI & ISO.

DDL(Data Definition Language) :- it's a part of SQL by which the schema of the database is specified. It also has specification for all the relations of database which are as follows:-

1. Specification for relational schema.
2. Specification for domain of each attribute value.
3. Specification for constraint.
4. Specification for creating index.
5. To provide authorization and security.
6. To provide physical storage for each relation.

Basic domain types of SQL:- there are few built in domain types available in all versions of SQL

1. **Numerical Data Types:-** which includes following:-

Int or Integer :- used for big size int values , occupies 4 byte of storage .

Small Int :- used for small sized integers, occupies 2 bytes of storage.

Tiny Int :- used for very small sized integers and unsigned integers.

Float(M,D):- used for floating point numbers and valid for signed numbers only. Here M is the total number of digits before decimal and D is the total number of digits after decimal

Double(M,D) only valid for floating point numbers having double precision. It is equivalent to REAL.

2. **String Type:-** which includes following:-

(i). **CHAR(C):-** this data type is valid for fixed length string. Here C is a length of the string which is given by user. If any string shorter than this size is stored then remaining space will be padded with the spaces.

(ii). **VARCHAR(C):-** used for variable sized string, here C is the maximum length of string which is specified by the user.

3. **Date or Time Date Types:-** Includes following:-

(i). **Date:-** this data type is defined in the formats YYYY-MM-DD i.e. date is

stored in this format and it can be in between 1000-01-01 to 9999-12-31. For example if age of a student in student table is 1st july 1980 then it will stored in the 1980-07-01 format.

(ii) **DATETIME** :-Date and time together can be stored in YYYY-MM-DD HH:MM:SS Formate. Date and time can be stored from 1000-01-01 00:00:00 to 9999-12-31 23:59:59. For example 2:35 PM and 1st july 1980 can be stored as 1980-07-01 14:35:00.

(iii) **Time**:- It stores time in HH:MM:SS format.

(iv) **Year**:- It stores year in 2 and 4 digit format.

Data Manipulation language (DML):- It is that part of SQL which is also known as query language so DML and query language are synonyms. DML is used to manipulate (Insert, Delete, Update and Retrieve) the data stored in any relation. The main commands of DML are:

- (i) SELECT
- (ii) UPDATE
- (iii) INSERT
- (iv) DELETE

Data Control Language :- DCL is the collection of SQL commands which provide security and maintain the rights of Data Manipulation. DCL includes the following Commands :-

- (i) COMMIT
- (ii) ROLLBACK
- (iii) GRANT
- (iv) REVOKE

DDL Commands and Syntax:-

CREATE :- CREATE Table command is used to create a table and a relation . The syntax of command is defined as following:-

Syntax: MYSQL generic syntax for creating a table is.

CREATE TABLE table_Name (F1 D1, F2 D2,..... ,Fn Dn<Integrity Constraints1,.....<ICk>);

Here in above syntax each Fi is a name of a table field or attribute and Di is the domain type of values under each Fi.

Apart from that many types of constraints can also be applied on the table like PRIMARY KEY, FOREIGN KEY etc.

For example if table named as student is to be created whose schema is defined as Student(Roll_No, Name, Age, Address, Class) and schema of Relation classes is defined as Classes(Class_Name , CRoomNo, Cstrength) then creation of student table can be defined as:

```
CREATE TABLE Student
(Roll_No int NOT NULL AUTO_INCREMENT, Name CHAR(20), Age int
NOT NULL, Address VARCHAR(30), Class CHAR(10) NOT NULL,
primary key(Roll_no), foreign key(class) references classes(class_name));
```

Here, in above example Roll_No can be defined as primary key and foreign key constraint is used to prevent the operation that try to violate the link between student and classes table. class can be taken as foreign key. Here NOT NULL and Auto increment are constraint used to define limitations i.e. NOT NULL is used to define that there must not be any NULL value in the attribute Roll_No. There would be an error if NULL value is inserted in Roll_No whereas Auto_increment is used to increase the value of Roll_no field by 1. It has value 1 by default. If we want to start the sequence by another value then we use the syntax.

```
ALTER table student AUTO_INCREMENT=100
```

Syntax for Classes table

```
CREATE Table Classes (Class_Name CHAR(10) NOT-NULL, CRoomNo
CHAR(10),
PRIMARYKEY(Class_Name));
```

Above table can be created by MySQL Prompt like

```
root@host# MYSQL-u root-p
```

```
enter passsword : *****
```

```
MySQL> use School_Management
```

Use command is used to use database named as School_Managemnt.

```
MySQL> CREATE DATABASE School_Management;
```

```
MySQL> use School_Management
```

```
MySQL> CREATE TABLE Student (Roll_No Int NOT NULL
AUTO_INCREMENT, Name CHAR(20), Age int NOT NULL, Address
VARCHAR(30), Class CHAR(10) NOT NULL, PRIMARY KEY (Roll_No),
FOREIGN KEY (Class) REFERENCES Classes(Class_Name));
```

```
-> Query ok, 0 row affected
```

In MYSQL termination is done by placing semicolon(;) at the end of any command or statement Any table can be removed from the database by the following command.

Generic syntax:

```
DROP TABLE table_name;
```

To remove the table named as Student the command is

```
DROP TABLE Student;
```

ALTER table command :-This command is used to add, delete, and modify the column in the database.

Following are the appropriate syntax:-

1. To add a new column

```
ALTER TABLE table_name ADD column_name datatype;
```

For example a new column Cstrength can be added to the table Classes

```
ALTER TABLE Classes ADD Cstrength int;
```

2. To remove existing column

```
ALTER TABLE table_name DROP COLUMN column_name;
```

3. to change the datatype of any existing column

```
ALTER TABLE table_name MODIFY column_name datatype ;
```

For example:-

```
ALTER TABLE Student MODIFY Age Date;
```

Other example:-

```
CREATE TABLE Teacher(Tname VARCHAR(20); DOB Date, Salary  
FLOAT(5,2),Address VARCHAR(30),phone int PRIMARY KEY (Tname));
```

```
CREATE TABLE Teaches(Tname VARCHAR(20), Class_name CHAR(10),  
PRIMARY KEY(Tname, Class_name), FOREIGN KEY(Tname)  
REFERENCES Teacher (Tname) , FOREIGN  
KEY(Class_Name)REFECENCES Classes (Classs_Name));
```

Check Constraints Syntax:-

Example of Check Constraints Syntax is following

```
CREATE TABLE Student (Roll_No int NOT NULL AUTO_INCREMENT,  
Name CHAR(20), Age int NOT NULL, Address VARCHAR(30), Class  
CHAR(10) NOT NULL, PRIMARY KEY(Roll_No), Check(Class in ('1st',  
'2nd', '3rd', '4th', '5th', '6th', '7th', '8th', '9th', '10th', '11th', '12th')));
```

Create Index Command:-This command is used to create index of any table. Index is not visible to the user but help them to search the data rapidly in the table by the following syntax.

```
CREATE INDEX Index_name ON table_name (column_name)
```

For example

```
CREATE INDEX SIndex on Student(Address)
```

Data Manipulation commands and their syntax:-

SQL DML commands are as follows:

- (i) INSERT
- (ii) DELETE
- (iii) UPDATE
- (iv) SELECT

(i) INSERT Command:- A blank table is created by CREATE TABLE COMMAND i.e. there is no value, records and tuples in that table. To fill up the values or data in that table insert command is used.

MySQL Syntax :-

INSERT INTO
Table_Name(Column_Name1,Column_Name2,.....,Column_Namen)VALUES(Value1,value2,.....valuen);
To insert string values, values are enclosed within single (' ') or Double(“”)quotes.

Insertion Through MySQL Command Prompt:-

For example if new values are to be inserted in Student table then
MySQL>USE School_Management;
MySQL>INSERT INTO Student (Name, Age, Address, Class)'
VALUES(“HARI”,15,”Chittorgarh”,”10th”);
In the above example we have not taken the column Roll_No because it is in auto increment mode which means MySQL automatically gives the value of Roll_No in sorted manner.

Other syntax is :

INSERT INTO Student VALUES (”prakash”,18,”jaipur”,”12th”);
In other syntax of INSERT INTO we can directly insert so many tuples by using SELECT instead of specifying tuples individually.

(ii) **DELETE command :-** An entire tuple is deleted by this command but specific value of any attribute can't be deleted by this.

Syntax:-

DELETE FROM T, WHERE P;
Here T is a relation from which tuple is to be deleted and P is a predicate (condition), according to which tuples would be deleted .
DELETE FROM Student, WHERE Roll_No=105;
By above command all the tuple having Roll_NO=105 would be deleted from the Student table. To delete all the tuples of any table we write.
DELETE FROM Student ;
DELETE FROM Classes;
To delete all the records of Class 10th students from the table Classes
DELETE FROM Student, WHERE Class=”10th”;

(iii) **UPDATE command**:- UPDATE command is used to change the value of any attribute from the given table, i.e. if we do not wish to change the entire tuple but only a specific value of any attribute then update command is used.

Syntax:

UPDATE table_name SET first_field=value1, second_field=value2
[WHERE clause];

Two or more values of fields can be updated in following manner

MySQL> UPDATE Student,
SET Class="11th", WHERE Roll_No=12;

In above example, student whose Roll_No is 12th then we are try to change the class from 10th to 11th.

To change the address of a student:-

UPDATE Student ,SET Address="Ajmer", WHERE Roll_No=102;
student

RollNo	Name	Age	Address	Class
11	Hari	15	Jaipur	10th
101	Prakash	16	Kota	11 th
102	Basu	11	Udaipur	6 th
120	Viveka	9	Jaipur	4 th

After update the following changes will occur in the table i.e. table will be in following form:-

Roll_No	Name	Age	Address	Class
11	Hari	15	Jaipur	10th
101	Prakash	16	Kota	11 th
102	Basu	11	Ajmer	6 th
120	Viveka	9	Jaipur	4 th

(iv) **SELECT statement**:- There are following three clauses in any SQL query expression:-

- (i) SELECT
- (ii) FROM
- (iii) WHERE

i.e. any SQL query expression consists of three clauses in its basic structure.

- SELECT clause is used to display the attributes in the output relation

- FROM clause is used to address the relation which is used in query expression. Relation written in the FROM clause join the form of Cartesian product.
- WHERE clause is used to write the predicate (condition) which is basically applied to the relation written in FROM clause and Boolean values of which (true or false)

SQL query is in this form:-

```
SELECT AT1, AT2, AT3 , ..., ATn,
FROM r1, r2, r3, ... rn,
WHERE P;
```

Here ATi denotes attribute and r_i denote relation and P is a predicate

SELECT clause :- It keeps all the attributes which has to be retrieved from the relation.

Example :- To make it crystal clear we take the example of student table from School_Management database as followed:-

Roll_No	Name	Age		Address		Class
101	Bhagat singh	20		Ajmer		12th
105	Chandra	17		Kota		11 th
12	Shekhar	16		Jaipur		10th
17	DinDayal	15		Udaipur		9 th
90	Rohit	9		Ajmer		5th

Syntax of SELECT command is:

```
SELECT field_names, FROM reation_names;
```

Several fields can be fetched by writing field_names. We can also use astrick(*) to display all the relation in the output relation.

For example:-

```
SELECT Roll_No, Age, FROM Student;
```

Output->

RollNo		Age
101		20
105		17
12		16
17		15
90		9

```
SELECT * FROM Student ;
```

Output->

RollNo	Name	Age	Address	Class
101	Bhagat	20	Ajmer	12 th
105	Chandra	17	Kota	11 th
12	Shekhar	16	Jaipur	10 th
17	DeenDayal	15	Udaipur	9 th
90	Rohit	9	Ajmer	5 th

Duplicate fields are removed by DISTINCT keyword when used with SELECT.

SELECT DISTINCT Address FROM Student ;

The output of above query produces a relation with no duplicate Address.

Output->

Address
Ajmer
Kota
Jaipur
Udaipur

Arithmetic operations can be used in SELECT command along with field names .Operations which are used are as follows:-

Description	Operator
Addition	+
Subtraction	-
Division	/
Multiplication	*

To understand this we take a table records of all the teachers. The name of table is Teacher which contain attributes Teacher (Tname, Address, Salary, PhoneNo);

Tname	Address	Salary	Phoneno
Radha krishan	Kerla	5000	1234567899
Lalaji	Udaipur	750	1111162123
Sarvopalli	Rampur	2000	1312171080
Headgevar	Maharastra	9000	1111110510

SELECT Tname, Salary *10, FROM Teacher;

Output of above query is a relation which contain Tname , Salary, but all the

values of Salary is multiplied by 10

Output ->

Tname	Salary*10
Radha krishnan	50,000
Lalaji	7500
Sarvopalli	20,000
Leadgevan	90,000

Use of SELECT statement with WHERE clause:-

One or more condition are written in WHERE clause and retrieval can be done only if condition is satisfied.

For example:- query for table teacher is

SELECT Tname, Salary, FROM Teacher WHERE Salary>2000;

Output ->

Tname	Salary
Radha krishnan	5000
Harivansh	9000

Following logical connections are used in WHERE clause :-

- (i) AND
- (ii) OR
- (iii) NOT

Comparative operators are used in WHERE clause

Description	Operators
Less than	<
Less than or equal to	<=
Greater than or equal to	>=
Greater than	>
Equal to	=
Not equal to	!= or <>

Example:-

SELECT Tname, FROM Teacher, WHERE Salary>2000 AND Address="Kerala";

Output ->

Tname
Radha krishnan

3. Data Control Language commands (DCL):- These commands are related to database security. It gives privileges to users, so that users can access the some database objects.

GRANT command:- This command is used to grant permission by one user to another user to use its created objects, i.e. until or unless say user A can't grant access of its created table to user B, user B can't access it. This permission is given by using GRANT command.

GRANT statement provides many kinds of privileges on many objects (Table, View)

Syntax:-

```
GRANT [type of permission] ON [database name] .  
[table_name] TO'[user name] '@'localhost';
```

Here types of permissions are.

- CREATE- Give permission to create new table or database.
- DROP - Give permission to drop the table or database.
- DELETE - Give permission to delete lines or tuples.
- INSERT- Give permission to insert a new lines or tuples.
- SELECT - Give permission to read the table or database.
- UPDATE- Give permission to update the tuples.

If we use asterisk (*) in place of database name or table name in above syntax then it gives the permission to access any database or any table.

Syntax:-

```
GRANT ALL Privileges ON *.* TO 'new_user @'localhost';
```

This command gives permission to read, write, execute and to perform all the operations on all the databases and tables.

Example- To understand GRANT command in MySQL

Create New Users:- Default user in MySQL is Root, which has full access on all the databases. To create new user the syntax is

```
MySQL> CREATE USER 'new_user' '@'localhost' IDENTIFIED BY  
'Password';
```

Example:-

```
MySQL> CREATE USER '14EEACS350' '@'localhost' IDENTIFIED BY  
'123456';
```

Using above syntax a new user named as '14EEACS350' is created and password is '123456', although this user doesn't have any kind of access to database even though this user is not allowed to login therefore privileges are to be given to the user.

Example:- GRANT ALL PRIVILEGES ON
School_Management.* TO '14EEACS350'@'localhost'
IDENTIFIED BY '123456';

By above example user 14EEACS350 is given access on all the tables of School_Management database.

Execute following command once access is given to the user.

FLUSH PRIVILEGES;

To make all the changes effective

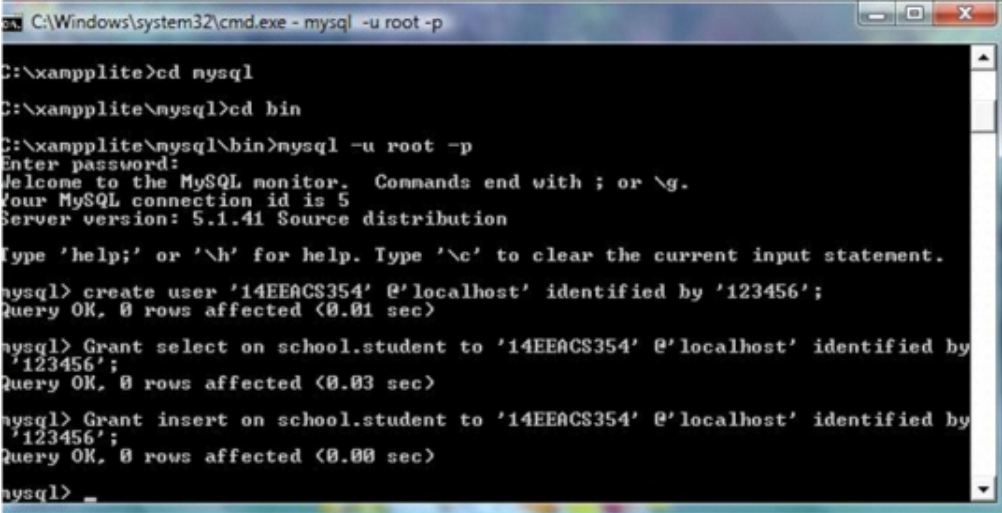
If administration doesn't want to give all the privileges to user '14EEACS350', then to give access to read only,

GRANT SELECT ON School_Management.Student TO '14EEACS350' @
'localhost' IDENTIFIED BY '123456';

To give privileges of insertion in Student table

GRANT INSERT ON School_Management.Student TO '14EEACS350' @
'localhost' IDENTIFIED BY '123456';

Like above example, other privileges can be given to the users. Below is the screen shot for assigning privileges insert and select to new user.



```
C:\Windows\system32\cmd.exe - mysql -u root -p
C:\xampplite>cd mysql
C:\xampplite\mysql>cd bin
C:\xampplite\mysql\bin>mysql -u root -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 5
Server version: 5.1.41 Source distribution

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> create user '14EEACS354' @'localhost' identified by '123456';
Query OK, 0 rows affected (0.01 sec)

mysql> Grant select on school.student to '14EEACS354' @'localhost' identified by
'123456';
Query OK, 0 rows affected (0.03 sec)

mysql> Grant insert on school.student to '14EEACS354' @'localhost' identified by
'123456';
Query OK, 0 rows affected (0.00 sec)

mysql> _
```

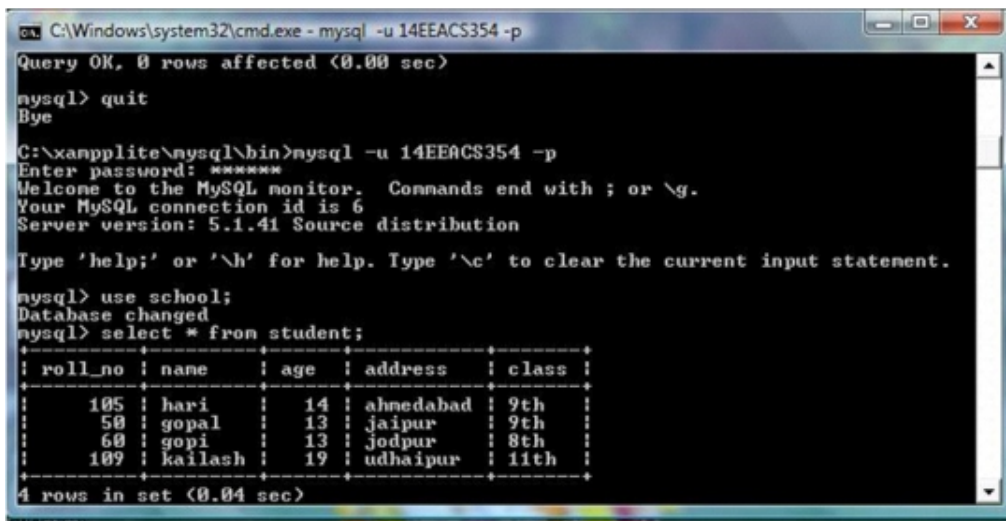
Figure 3 Screen shot of privileges assigning to new user

User '14EEACS350' is only given privileges to insert and select. So this user is allowed to perform select and insert on student table and not any other operation.

To see this we first logout by entering QUIT command and then again go for new login

MySQL-u[new username]-P;

MySQL> MySQL-u 14EEACS350 -P;
Enter password :123456;
And then perform different command on Student table following are some screen shots.



```
C:\Windows\system32\cmd.exe - mysql -u 14EEACS354 -p
Query OK, 0 rows affected (0.00 sec)
mysql> quit
Bye
C:\xampp\mysql\bin>mysql -u 14EEACS354 -p
Enter password: *****
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 6
Server version: 5.1.41 Source distribution

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> use school;
Database changed
mysql> select * from student;
+----+-----+-----+-----+-----+
| roll_no | name  | age  | address | class |
+----+-----+-----+-----+-----+
| 105    | hari  | 14   | ahmedabad | 9th   |
| 50     | gopal | 13   | jaipur   | 9th   |
| 60     | gopi  | 13   | jodpur   | 8th   |
| 109    | kailash | 19  | udhaipur | 11th  |
+----+-----+-----+-----+-----+
4 rows in set (0.04 sec)
```

```

C:\Windows\system32\cmd.exe - mysql -u 14EEACS354 -p
'123456';
Query OK, 0 rows affected (0.00 sec)

mysql> quit
Bye

C:\xampp\mysql\bin>mysql -u 14EEACS354 -p
Enter password: *****
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 6
Server version: 5.1.41 Source distribution

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> use school;
Database changed
mysql> select * from student;
+----+-----+-----+-----+-----+
| roll_no | name  | age  | address | class |
+----+-----+-----+-----+-----+
| 105    | hari  | 14   | ahmedabad | 9th   |
| 50     | gopal | 13   | jaipur   | 9th   |
| 60     | gopi  | 13   | jodpur   | 8th   |
| 109    | kailash | 19  | udhaipur | 11th  |
+----+-----+-----+-----+-----+
4 rows in set (0.04 sec)

mysql> insert into student values(54, 'board', 10, 'ajmer', '5th');
Query OK, 1 row affected (0.03 sec)

mysql> select * from student;
+----+-----+-----+-----+-----+
| roll_no | name  | age  | address | class |
+----+-----+-----+-----+-----+
| 105    | hari  | 14   | ahmedabad | 9th   |
| 50     | gopal | 13   | jaipur   | 9th   |
| 60     | gopi  | 13   | jodpur   | 8th   |
| 109    | kailash | 19  | udhaipur | 11th  |
| 54     | board | 10   | ajmer    | 5th   |
+----+-----+-----+-----+-----+
5 rows in set (0.00 sec)

mysql> delete from student where roll_no=60;
ERROR 1142 (42000): DELETE command denied to user '14EEACS354'@'localhost' for table 'student'
mysql> _

```

REVOKE command :- This command is used to take back the privileges from any object(table) and its syntaxes are same as to GRANT command.

Syntax:- REVOKE [type of permission] ON [database_name].
[table_name] FROM '[user name] '@'localhost';

For example :- DROP can be used to eliminate any user.

DROP USER '14EEACS350' '@'localhost';

Example:-

REVOKE DELETE ON Student FROM 14EEACS350;

COMMIT command :- It is used to make the changes permanent in the database i.e. if any change apply to the database is committed by commit

command. By default auto commit mode is enabled in MySQL i.e. if any changes or updating of a table occurs, then that updating is stored on MySQL disk to make it permanent. We can disable auto commit by START TRANSACTION.

ROLLBACK:- the rollback undo all the transaction to a certain checkpoint before failure has occurred. In case of cascaded rollback there are multiple failure due to single failure which causes multiple times of rollback in different transaction.

Introduction to SQL operators :- These are reserved words which are used in WHERE clause of any SQL query. Main operators are :-

- (i) Comparison
- (ii) Arithmetic
- (iii) Logical
- (iv) Operators used to negate the condition.

SQL comparison operators:-

Operators	Description
(=)	this operator is used to check the equality or non equality of two operators. Condition is true in case of equality.
(<> or !=)	this operator is used to check the equality or non equality of two operators. Condition is true in case of non equality.
(>)	if the value of left hand side operator is greater than right hand side operator then condition is true.
(<)	if the value of left hand side operator is less than right hand side operator then condition is true.
(>=)	if the value of left hand side operator is greater than or equals to right hand side operator then condition is true.
(<=)	if the value of left hand side operator is less than or equals to right hand side operator then condition is true.

SQL Arithmetic operators:-

Operators	Descriptions
(+)	used to add left hand side operand and right hand side operand.
(-)	used to subtract the value of right hand side operand from left hand side operand.
(*)	used to multiply left hand side operand and right hand side operand.
(% or modulus)	Right hand side operand divides left hand side operand

and return remainder as a result.

To understand arithmetic and comparison operators consider the following student, teacher and classes table as example:-

Student

Roll No	Name	Age	Address	Class
109	Omprakash	9	Jodhpur	4th
111	Prakash	15	Jaipur	10th
91	Suman	11	Jaipur	6th
75	Shanker	13	Ajmer	8th

Teacher

Tname	Address	Salary	PhoneNo
Radha krishnan	Kerla	3000	1234567899
Rajesh	Jodhpur	5000	9413962123
Lalaji	Ajmer	9000	9312171080
Hariom	Kerla	40000	5189310510

Classes

Class_name	CroomNo	CStrength
12th	F-1	90
6th	F-17	65
9th	S-21	110

Figure 4 Student , Teacher and classes table

Example 1.

SELECT * FROM Student,WHERE Age=15;

Output ->

Roll_No	Name	Age	Address	Class
111	Prakash	15	Jaipur	10 th

Because there is only one row with Age=15 in student table

Example 2.

SELECT * FROM Student, WHERE Age>11;

Output ->

Roll_No	Name	Age	Address	Class
111	Prakash	15	Jaipur	10 th
75	Shanker	13	Ajmer	8th

Example 3.

SELECT * FROM Student, WHERE Age <= 13;

Output ->

Roll_No	Name	Age	Address	Class
109	Omprakash	9	Jodhpur	4th
91	Suman	11	Jaipur	6th
75	Shanker	13	Ajmer	8th

SQL Logical operators:-

Operators

Description

AND a AND b is true if both(a and b) of them is true.

OR a OR b is true if either a or b is true.

For example:-

Consider a table Teacher

(i) SELECT Tname , Salary FROM Teacher WHERE Salary <= 5000 and Salary >=4000;

Output ->

Tname	Salary
Rajesh	5000

Here in this example only one record is available where both the conditions, Salary <= 5000 and Salary >=4000 is satisfied.

(ii) SELECT Tname , Salary, FROM Teacher, WHERE Salary <= 5000 OR Address = "Kerla";

Output ->

Tname	Salary
Radha krishnan	3000
Rajesh	5000
Hariom	40000

Here in this example there are three entries are found as a result because both conditions Salary <=5000 and second condition address=" Kerla" are true for

Output ->

Roll_No	Name	Age	Address	Class
91	Suman	11	Jaipur	6th
75	Shanker	13	Ajmer	8th

Example (2): Names ending with “sh”;

MySQL> SELECT * FROM Student WHERE Name LIKE '%sh';

Output ->

Roll_No	Name	Age	Address	Class
109	Omprakash	9	Jodhpur	4th
111	Prakash	15	Jaipur	10 th

Example 3: Find all the names consist of 5 characters

MySQL> SELECT * FROM Student WHERE Name LIKE '-----';

Output->

Roll_No	Name	Age	Address	Class
91	Suman	11	Jaipur	6th

If a special character (%,_) is to be added in some pattern then escape character (\) Backslash is to be used by placing it before the special character.

Example:-

- LIKE 'BJ\%BHARAT%' will match all the string starts with BJ%BHARAT
- LIKE 'BJ\\BHARAT%' will match all the string starts with BJ\BHARAT

SQL NOT LIKE is used to find mismatches. Extended regular expression are used for pattern matching in SQL which are of two types . REGEXP(RLIKE) and NOT REGEXP(NOT RLIKE).

“[...]” is a character class which matches any character inside the bracket.

Example “[pqr]” matches “p”, ”q” or “r”

“[a-z]” matches any letter

Operators Descriptions

IS NULL this operator is used to compare any value with a NULL value .

If no value exist in any field of an attribute than NULL value is used to represent it.

Example:- if no PhoneNo exists for a teacher in teacher table then, there would be a NULL value and for testing of that key word NULL is used in Mysql

Example:- SELECT Tname FROM Teacher WHERE PhoneNo IS NULL;

Operators and NULL values :-

Operator	Value1	Value2	Output
+, -, *, or /	value	NULL	NULL

+, -, * or /	NULL	value	NULL
>, <, >=, <=, <>	NULL	value	unknown
<,>,,>=,<=,<>,,=	value	NULL	unknown
AND	true	unknown	unknown
AND	false	unknown	false
AND	unknown	unknown	unknown
OR	true	unknown	true
OR	false	unknown	unknown
OR	unknown	unknown	unknown
NOT	unknown	unknown	unknown

That is, if result of predicate in WHERE clause is false or unknown than no tuple displays in result .

NOTE:- all aggregate function except (count(*)) will ignore the NULL value as their input calculation .

SET Operators:- set operations are applied to the relations are as follows:-

- (i) UNION
- (ii) UNION ALL
- (iii) INTERSECT
- (iv) EXCEPT

UNION set operator :- This operator is used to combine the result set of two or more than two SELECT statement. It removes duplicate tuples.

NOTE :-UNION can be applied to only those relations which have same number of fields and their data type must also be same.

Syntax:- MySQL UNION operator syntax is

SELECT ex1, ex2, ... ,ex_n FROM tables [WHERE condition]

UNION [DISTINCT]

SELECT ex1, ex2, ... ex_n FROM tables WHERE condition;

Here, DISTINCT keyword is not required because UNION itself removes the duplicates.

Example:-

SELECT Class FROM Student

UNION

SELECT Class_name FROM Classes;

Output ->

Class
4th
10th
6th
8th
12th
9th

Here in this example there is only one field i.e. SELECT statement return only one field. Data types of both the fields are same. Column name of return relation will be the name of first SELECT statement column name because we know that UNION will remove all duplicates that's why value 6th occur only once in the relation. But if want to keep all duplicates tuples then we will use UNION ALL.

Syntax :- Syntax of UNION ALL is same as the syntax of UNION what matters is that UNION ALL IS written instead of UNION.

SQL INTERSECT operator:- It is used to return a relation which contains the tuples common to given relations i.e. if there exist any tuple in two or more than two relation than that common tuple will be the result of intersection of the two relations.

Example:-

```
SELECT Class FROM Student
INTERSECT
SELECT Class_name, FROM Classes;
```

Output ->

Class
6th

Note:- Intersection is not available in MySQL but can be done through IN OPERATOR.

Syntax:-

MySQL IN OPERATOR syntax

Expression IN(value1 , value2 , ... value_n);

Here, expression is value which we want to test and value1,value2,...valuen are values in which we have to test the value.If any value matches the test value then IN OPERATOR returns true.

SQL EXCEPT operator :- it combines two SELECT statement and return a relation which contain those tuples of first SELECT statement which are not in

the second SELECT statement.

Syntax:-

SELECT column names FROM tables [WHERE clause]

EXCEPT

SELECT column names FROM tables WHERE clause;

EXCEPT operator is not available in MySQL but to fulfill the needs NOT IN operator is used.

SQL functions :- Many built in functions are available in SQL which are as follows:-

(i) Date and Time function:- date and time functions and there description which are used in MySQL are as follows:-

Function name	description
ADDDATE()	to add dates
ADDTIME()	to add times
CURDATE()	return a present date
CURTIME()	return current time
DATE_SUB()	subtracts two dates
NOW()	it return present date & time
STR_TO_DATE()	it changes string into date

Example 1. Syntax:- SELECT ADDDATE(expr, days)

MySQL> SELECT ADDDATE('1980-07-01',32);

Output -> 1980-08-02

Example 2. Syntax:- SELECT ADDTIME(expr2, expr1)

Here expr1 is added to expr2

MySQL> SELECT ADDTIME('1999-12-31 23:59:59.999999', '11:1:1.000002');

Output -> 2000-01-02 01:01:01.000001

Example 3.

CURDATE syntax :- SELECT CURDATE();

Return present date in YYYY-MM-DD format.

String function :- useful string functions of MySQL are as follows:-

Name	description
ASCII()	return a numeric value of left most character
BIN(N)	return a string representation of binary value of N
BIT_LENGTH(str)	return the length of string in bits

CHAR(N)	this function consider each argument N as an integer and gives its string represent. This string is the combination of all those characters which are passed as an argument.
CHAR_LENGTH(Str)	it measure the string length in characters.
CONCAT(Str1,Str2,...)	augmented strings are concatenated and return as result.
FIELD(Str,Str1,Str2,...)	it return an index of string str from the given list, if string is not obtained than result will be 0.
LOAD_FILE(file_name)	read the file and return its contain in the form of string. To use this path to that file must be specified.
REPLACE(Str, from_Str, to_Str)	it returns a string str after removing all the occurrences of from_Str and replace it by to_Str.

There are many functions in MySQL except the listed ones.

Example1:- MySQL> SELECT ASCII('3')

Output ->

ASCII('3')
51

Example2:- MySQL> SELECT BIN(2)

Output -> 1 0

Example3:- MySQL> SELECT BIT_LENGTH('BHARAT')

Output ->

BIT_LENGTH('BHARAT')
48

Example4:- MySQL> SELECT CONCAT('BH','A','GAT',' ','SI','NGH')

Output -> BHAGAT SINGH

Example5:- MySQL> UPDATE Student SET Address=
LOAD_FILE('pathname') WHERE Roll_No=105 ;

Example6:- MySQL> SELECT CONCAT(Roll_No, Name, Class) FROM Student

Output ->

CONCAT(Roll_No, Name, Class)
105hari9th
50gopal9th
60gopi8th
109kailash11th

Example7:- MySQL> SELECT CHAR(66,72,65,82,65,84)

Output->

CHAR(66,72,65,82,65,84)
BHARAT

RAND function:- Is used to generate any number randomly between 0 to 1 in MySQL.

Example :- MySQL> SELECT RAND(), RAND();

Output-> RAND()

RAND()
0 . 0 3 0 1 4 5 6 7 8 4 5 3 5 7

0.93969467893221

SQRT function :- It is used to calculate the square root of any Number.

Example :- MySQL> SELECT SQRT(64)

Output ->

SQRT
8

If square root of Salary field is calculated from Teacher then this is done as follows-

MySQL>SELECT Tname, SORT(Salary)

FROM Teacher

Output->

Tname	Salary
Radha Krishnan	54.7722557505166
Rajesh	70.7106781188548
Lalaji	94.8683298050514
Hariom	200

Numeric functions:- these functions are used in mathematical operations.

Few important functions are listed.

Function	Description
(i)ABS(V)	it returns a full value of function V
(ii)GREATEST(n1,n2,...)	it return the greatest value among the parameter list values.
(iii)INTERVAL(N,n1,n2,n3,----)	it compairs the value of N to the values(n1,n2.....) one by one if N<n1it return 0 else return 1 for N<n2 and return 2 For N<n3.
(iv) LEAST(N1,N2....)	It is a inverse of GREATEST

Example(1) MySQL> SELECT ABS(-6);

Output ->

ABS(-6)
6

Example(2) MySQL> SELECT GREATEST(4,3,7,9,8,0,10,50,70,11)

Output ->

GREATEST(4,3,7,9,8,0,10,50,70,11)
70

Example(3) MySQL> SELECT INTERVAL(4,3,5,8,11,12,17,18)

Output ->

INTERVAL(4,3,5,8,11,12,17,18)
1

Aggregate functions:- This function accepts collection of values as an input in MySQL and return a single value as an output .There are five types of built in aggregate function in MySQL.

AVERAGE: Avg()
MAXIMUM: Max()
MINIMUM: Min()
TOTAL: Sum()
COUNT: Count()

Here input to the function Sum and Average must be numbers while other operators can work on string also.

Avg() function: This function is used to calculate the average value of given field values.

Example:- MySQL> SELECT Avg(Salary) FROM Teacher;

Output->

Avg(Salary)
14250.0000

In above example Avg() function return the average value of Salary field values.

Sum() Function: This function return the Sum of all the values of any fields.

Example:- MySQL> SELECT Sum(Salary) FROM Teacher;

OUTPUT->

Sum(Salary)
57000

Max() function:- return the maximum value among the value of any record set

Example :- MySQL> SELECT Max(Salary) FROM Teacher

Output->

Max(Salary)
40000

Min() function:- It returns record with minimum value.

Example :- MySQL> SELECT Min(Salary) FROM Teacher

Output->

Min(Salary)
3000

Count() function :-it is used to count the number of records in the table i.e. total number of records can be calculated.

Exampler1:- SELECT Count (*) FROM Student

Output ->

Count(*)
4

Example2 :- SELECT Count (*) FROM Student WHERE Class='9th'

Output ->

Count(*)
2

SQL ORDER BY clause:- to sort the rows in proper order "ORDER BY" clause is used or by this clause we can sort any columns value either in ascending or descending order.

Syntax:-

SELECT field1,field2....filedn FROM T1,T2...Tn ORDER BY field1,

field2...filedn [Asc[Desc]];

Clause can be applied to many number of fields for which keyword Asc and Desc is used . Asc stand for ascending and Desc stands for descending.

Example 1:- MySQL> SELECT Roll_NO, Age FROM Student ORDER BY Age Desc;

Output:-

Roll_No	Age
109	19
105	14
50	13
60	13

Example 2:- MySQL> SELECT Roll_NO, Age FROM Student ORDER BY Age Desc, Roll_No Desc;

Output:-

RollNo	Age
109	19
105	14
60	13
50	13

Example:- Query :- find all the students who sits in room number F-17

MySQL> SELECT Name FROM Student, Classes WHERE Class=Class_name AND CRoomNo='F-17' ORDER BY Name Asc;

In this example many tables are used because we required information from different tables. Like student name can be retrieve from Student table and CRoomNo field can be retrieved from Classes table. Both the tables are joined by their primary and foreign key. Because both the keys are same for the table i.e. tables are linked through their keys. Two tables are joined by a field which is common to the table. Here ORDER BY clause is used because desired result must be in increasing order.

Output->Students who want to sit in F-17

Name
--

SQL GROUP BY clause:- It is useful clause of MySQL. Many important

queries can be written by this clause. Collection of values of one or many columns can be grouped by this clause that is attributes mentioned in group by clause are used to form groups. Attributes or attributes values which are same for all tuples will represent a single group.

This clause can be understood by following example of student table:

Roll_No	Name	Age	Address	Class
1	Ajay	9	Jaipur	4th
2	Vijay	17	Kota	12th
10	Hari	11	Udaipur	7th
17	Shanker	13	Jaipur	8th
21	Om	21	Ajmer	12th
51	Mayank	15	Ajmer	9th
90	Anju	18	Ajmer	11th
53	Suman	12	Ajmer	10th
64	Kamal	10	Kota	4th
500	Komal	16	Udaipur	9th
700	Aryabhata	11	Jaipur	7th
900	Bodhayan	13	Jodhpur	8th

Figure 5 Student table

Example:- Query : find the number of student in each class

If we write the syntax like that then result obtained will be wrong.

MySQL> SELECT Count(*) FROM Student

Output->

Count (*)
12

By above query result will show the total number of student i.e. total number of students in the table is same as the total number of student in the school. Therefore it returns the total number of tuples in the relation. For appropriate result GROUP BY clause is used with Count aggregate function and syntax is MySQL> SELECT Class, Count(Roll_No) FROM Student GROUP BY Class;

After grouping Student table can be depicted as given below because group is making by Class attribute in GROUP BY clause so tuples with same Class will be display in a group.

Output ->

Class	Roll_No	Age	Name	Address
11th	90	18	Anju	Ajmer
12th	2	17	Vijay	Kota
12th	21	21	Om	Ajmer
10th	53	12	Suman	Ajmer
9th	51	15	Mayank	Ajmer
9th	500	16	Komal	Udaipur
8th	17	13	Shanker	Jaipur
8th	900	13	Bhodhayan	Jodhpur
7th	10	11	Hari	Udaipur
7th	700	11	AryaBhatta	Jaipur
4th	1	9	Ajay	Jaipur
4th	64	10	Komal	Kota

Class	Count(Roll_No)
10th	1
11th	1
12th	2
9th	2
8th	2
7th	2
4th	2

The column by which group is made , any calculation like Count, Avg, Max, Min etc can be applied by aggregate function. Therefore in this query Count aggregate function can be applied for each groups tuples having same class value because there are only two fields in SELECT statement so following result are obtained.

Class	Count(Roll_No)
10th	1
11th	1
12th	2
9th	2
8th	2
7th	2
4th	2

Note :- any attribute which occur outside the aggregate function in SELECT

statement are written inside the GROUP BY clause.

For example-> Class attribute which appears in SELECT statement will also be in GROUP BY clause.

Example 2:- Name all the classes in which number of student are more than 1.

Syntax:-

```
MySQL> SELECT Class, Count(Roll_No) FROM Student GROUP BY Class  
HAVING Count(Roll_No);
```

Output->

Class	Count(Roll_No)
12th	2
9th	2
8th	2
7th	2
4th	2

Above output contains only those tuples whose count is more than 1. To check condition for a single tuple in SQL can be done by using WHERE clause But to see the condition for tuples in the groups made by GROUP BY clause, HAVING clause is used. This is the main difference between WHERE and HAVING clause.

Predicate of HAVING clause are applied after making the group by GROUP clause. Therefore aggregate function can also be used with it.

Note:- If WHERE, HAVING, GROUP BY occur simultaneously than predicate is applied to WHERE clause first after that all the tuples for which condition is satisfied will be kept in the same group by group clause. At the end having clause is applied for each group. Groups which don't fulfill the condition of having will be extracted.

SQL RENAME operation:- Name of any relation and attributes can be changed by RENAME operation using 'AS' clause. It is used for renaming and syntax is.

Old_relation/ attribute_name as new_name 'AS' clause can be used in both SELECT and FROM statement.

Example 1:- MySQL> SELECT Avg(Salary) AS AvSalary FROM Teacher;

AvSalary
14250.0000

Example 2:- MySQL> SELECT Class, Count(Roll_No) AS total, FROM Student GROUP BY Class;

Output->

Class	total
10th	1
11th	1
12th	2
9th	2
8th	2
7th	2
4th	2

SQL JOIN:- JOIN keyword is used to combine two or more than two tuples from the given relation. In joins two relations are taken as input and return a single relation as an output.

There are many mechanisms to perform joins:-

- (1) Cartesian product mechanism
- (2) Inner join
- (3) Outer join (left, right, full)

Every join type stated earlier also has one Join condition with it. So each Join expression is consists of a Join type and a Join condition which is used in FROM clause.

To understand the operation of Join we consider two tables, Student given in figure 5 and Classes table of figure 6

Class_name	CRoomNo	CStrength
12th	F-1	90
11th	F-2	75
10th	F-5	99
9th	S-21	110
8th	S-10	70
7th	F-10	85
6th	F-17	65
5th	F-7	60
4th	F-9	55
3th	F-8	50
2th	S-15	35
1th	S-9	60

Figure 6 Classes table instance

We write a query to join the Student and Classes table

```
MySQL> SELECT Roll_No, Class, CStrength FROM Student AS St, Classes AS S, WHERE St.Class=S.Class_name;
```

Here Student is renamed as St and Classes as S. Cartesian product is applied to both the relation where each tuple of St is joined with each tuple of S. So total number of tuples in resultant relation is.

$$N1 * N2 = 12 * 12 = 144$$

Here N1 is total number of tuples in St table and N2 is the total number of tuples in S table but result relation contains only those tuples which fulfils the condition of WHERE clause.

OUTER JOIN OPERATIONS:- These are of following types.

- LEFT OUTER JOIN
- RIGHT OUTER JOIN
- FULL OUTER JOIN

Following join condition are used with Outer Joins:-

- 1) Natural
- 2) ON (Predicate)
- 3) Using (A1,A2,.....An)

Left outer join and ON Join condition:-Two Understand this consider two tables

Classes

Class_name		CRoomNo	CStrength
12th		F-11	90
9th		S-21	110
7th		F-10	85
4th		F-9	55

Admission

Class_name		Roll_No	admission_date
12th		79	2000/7/15
9th		89	2010/8/13
6th		69	2012/7/21
5th		49	2013/7/03

Figure 7 Classes and Admission relation

Syntax:-

Select Classes, Class_Name, Roll_No, CStrength From Classes Left OuterJ
oin Admission on Classes.class_Name=Admission.Class_Name

Here Name of Relation is written with it's attribute because Name of attributes

is common in both the relation so to avoid the ambiguity we have written name of relation along with attribute name.

Output

Class name	Roll No	CStrength
12th	79	90
9th	89	110
7th	Null	85
4th	Null	55

Matching tuples of both the relation and unmatched tuples of left side relation are present in the result relation.

RIGHT OUTER JOIN AND ON JOIN CONDITION:-

Syntax:

Classes Right Outer Join admission on
 classes.Class_Name=Admission.Class_Name

Output

Class_ Name	CRoom No	CStength	Class_ Name	Roll_ No	Admission_ date
12th	F-11	90	12th	79	2000/7/15
9th	S-12	110	9th	89	2010/8/13
Null	Null	Null	6th	69	2012/7/21
Null	Null	Null	5th	49	2013/7/03

Right outer Join is as same as left outer join but the only difference is that unmatched tuples of R.H.S relation of Join operation will be present in the result relation. Null value is assigned for the attributes of left relation.

Full-outer join ON condition:-

Syntax:

Classes full outer join admission on classes. Class_Name = Admission.
 Class_Name

Class_ Name	CRoom No	CStrength	Class_ Name	Roll_ No	Admission_ date
12th	F-11	90	12th	79	2000/7/15
9th	S-12	110	9th	89	2010/8/13
7th	F-10	85	Null	Null	Null
4th	F-9	55	Null	Null	Null
Null	Null	Null	6th	69	2012/7/21
Null	Null	Null	5th	49	2013/7/03

Here in result unmatched tuples of both the relation occur and Null Values for unmatched tuples of other relation is assigned.

Outer Join and Natural condition:- When we perform natural join of two relation, the number of tuples contain in result relation depends on the common attributes in both the relation. The common attributes appears first in result relation with single copy(without duplicate).

Example

Classes Natural right outer join admission

Class_ Name	CRoom No	CStength	Roll_ No	Admission_ date
12th	F-11	90	79	2000/7/15
9th	S-12	110	89	2010/8/13
6th	Null	Null	69	2012/7/21
5th	Null	Null	49	2013/7/03

Other outer joins for natural join condition can also be obtained like above

Inner join :- Example-

Classes inner Join Admission On Classes.Class_Name= Admission.Class_Name;

Output->

Class_ Name		CRoom No	CStength	Class_ Name		Roll_ No	Admission_ date
12th		F-11	90	12th		79	2000/7/15
9th		S-12	110	9th		89	2010/8/13

Inner join and natural condition:-

Example :- Classes Natural Inner-Join Admission

Output ->

Class Name	CRoomNo	CStength	Roll No	Admission date
12th	F-11	90	79	2000/7/15
9th	S-12	110	89	2010/8/13

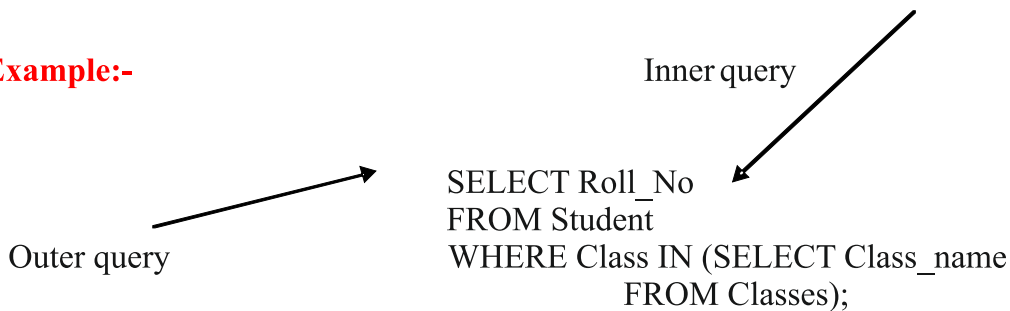
Here only one Attribute is common to both the relation, therefore Join is performed by that attribute only.

Note:- Join condition USING is also same as Natural Join only joining attributes in USING will be (A1,A2,...An) rather than all common attribute present in both relation which is the basic difference between using and natural join condition.

SQL Sub queries:- A sub query is a SQL query which is nested within another

query. A sub query can also be nested within another sub query. Sub query is called the inner query and the query in which a sub query is nested is called the outer query.

Example:-



The value return by a sub query can also be compared using comparison operators (=,>=,<= etc).

For example :- To understand this we use the Teacher table
SELECT Tname, Salary FROM Teacher WHERE Salary= (SELECT
Max(Salary) FROM Teacher);
Output ->

Tname	Salary
Hariom	40000

Example 2:- Query- Find the name of teacher having salary, less than the average salary of all teacher

SELECT Tname, Salary FROM Teacher WHERE Salary< (SELECT
Avg(Salary) FROM Teacher);
Output ->

Tname	Salary
Radha krishnan	3000
Rajesh	5000
Lalaji	9000

Important Points:

- A schema of a database is the logical design of a database , which is rarely changed.
- A collection of data or information stored in a table at any particular moment of time is called the database instance.
- **Primary key** :It is a set of one or more attributes (field) of a table which

can be used to identify uniquely any row or tuples of that table.

- Referential integrity can be ensured by foreign key constraints.
- SQL is a combination of relational algebra and relational calculus.
- Auto_increment is used to increase the value of a field by 1.
- GRANT ALL Privileges ON *.* TO 'new_user @ 'localhost'; This command gives permission to read, write ,execute and to perform all the operations on all the databases and tables.
- Attributes mentioned in group by clause are used to form groups. Attributes or attributes values which are same for all tuples will represent a single group.
- There are many mechanisms to perform joins e.g. Cartesian product mechanism, Inner join and Outer join (left, right, full).

Practice Questions

Objective type questions:

- Q1. Which one is not a SQL clause?
 a) Select b) From
 c) where d) condition
- Q2. Full form of SQL is
 a) Structure Question language
 b) syntax question language
 c) Structure query language
 d) Structure question language
- Q3. DDL stands for.
 a) Data definition language
 b) Dual data language
 c) Data data language
 d) none of these
- Q4. Count() is a
 a) String function b) numeric function
 c) both d) not exist
- Q5. Like operator is used for
 a) Concatenating strings b) count string character
 c) string matching d) all

Very Short answer type questions.

- Q1. What is SQL?
 Q2. What do you understand by SQL from clause?

- Q3. What is the importance of SQL select clause?
- Q4. Give names of types of SQL.
- Q5. What is the difference between unique and primary constraints?
- Q6. Define database instances.
- Q7. How we use order by clause in SQL.
- Q8. What is NULL in SQL?
- Q9. What do you mean by aggregate functions?
- Q10. Why we use SQL grant command.

Short answer type questions:

- Q1. What is the basic structure of SQL?
- Q2. What are the various DML commands? Give syntaxes for them.
- Q3. What is foreign key? How we create a foreign key in a table?
- Q4. Explain use of group by clause in SQL with example.
- Q5. What is the difference b/w Cartesian join and natural join.

Essay type questions:

- Q1. Explain SQL joins with taking suitable examples of tables.
- Q2. What are aggregate functions? How we use aggregate functions? Give an example of each.
- Q3. Explain the use of where, group by and having clause in a single SQL query? Give a suitable example.
- Q4. Consider following schema given.
students(Roll_no, Sname, age, phone, address, class)
Classes(Class_name, CRoomNo, CStrength) and write an SQL syntax for.
1) Find student names of those 5th class students sitting in room number F-12.
2) Find number of students of 10th class living in Ajmer.
- Q5. What do you mean by Sub queries? Why sub queries are useful. Explain use of sub queries in set comparison.

Answers key for objective questions

- Q1: d Q2: c Q3: a Q4: b Q5: c

Chapter 15

PL/SQL

Basics of PL/SQL

PL/SQL stands for Procedural Language extension of SQL. Procedural features of programming languages are incorporated with SQL that forms PL/SQL. It is developed by Oracle Corporation to enhance the capabilities of SQL.

A Simple PL/SQL Block:

A PL/SQL block consist of two things SQL and PL/SQL statements. These two are used to design a PL/SQL program

PL/SQL Block consists of three sections:

The Declaration section (optional).

The Execution section (mandatory).

The Exception Handling (or Error) section (optional).

Declaration section

This section of a PL/SQL Block which is a Optional section starts with the reserved keyword DECLARE. Purpose of the section is to declare placeholders like constants, records, variables and cursors. These placeholders are used to manipulate data in the execution section. Placeholders may be any of Variables, Constants and Records, which stores data temporarily. Cursors are also declared in this section.

Execution Section

This section of a PL/SQL Block starts and ends with the reserved keywords BEGIN and END respectively. The execution section is a mandatory section and where the logic of a program is written to perform any task. The programmatic constructs like loops, conditional statement and SQL statements forms the part of execution section.

Exception Section

This section of a PL/SQL Block which is optional section starts with the reserved keyword EXCEPTION and used for handling the errors in the program. If we use the PL/SQL Blocks then Blocks terminates with desired output.

Structure of PL/SQL Block

```
DECLARE
    Variable declaration
BEGIN
    Program Execution
Exception
```


Exception handling
END;

A SIMPLE PL/SQL CODE BLOCK THAT DISPLAYS THE WORD BHARAT

```
SQL> set serveroutput on  
1 SQL> begin  
2 dbms_output.put_line ('BHARAT');  
3 end;  
4 /
```

BHARAT

PL/SQL procedure successfully completed.

Let's discuss some important points of the PL/SQL program are:

A section of PL/SQL code block starts with the keyword Begin and is terminated with the keyword End, called the executable portion of PL/SQL block.

PL/SQL code blocks are comprised of statements. Each statement ends with a semi-colon.

PL/SQL code blocks are followed by a slash (/) in the first position of the following line as given above. This causes the code block statements to be executed.

Advantages of PL/SQL: Some advantages of PL/SQL block are:

Block Structures: In PL/SQL, blocks of code can be nested within each other. Each block is responsible for a unit of a task or a logical module. PL/SQL Blocks can be stored in the database and reused.

Better Performance: Because of processing the multiple SQL statements simultaneously by a PL/SQL engine as a single block, resultant performance is better and reducing network traffic.

Procedural Language Capability: To make it to become power full language PL/SQL consists of procedural language constructs such as conditional statements (if else statements) and loops like (FOR loops).

PL/SQL Placeholders:

Placeholders provide storage area to store data temporarily, which are used to manipulate data during the execution of a PL/ SQL block. PL/SQL Placeholders can be any of Variables, Constants and Records

Placeholders in PL/SQL :

Placeholders can be defined with a name and a datatype. This definition depends on the kind of data you want to store. Some of the datatypes used to define placeholders are, Number (n,m) , Char (n) , Varchar2 (n) , Date , Long , Long raw, Raw, Blob, Clob, Nclob and Bfile.

PL/SQL Variables:

These are placeholders that store the values that can change through the PL/SQL Block.

General Syntax to declare a variable is:

```
variable_name datatype [NOT NULL := value];
```

variable_name is the name of the variable.

datatype is a valid PL/SQL datatype.

NOT NULL is an optional specification on the variable. Value or DEFAULT value is also an optional specification, where you can initialize a variable. Each variable declaration is a separate statement and must be terminated by a semicolon.

For example, if you want to store the current salary of a teacher, you can use a variable.

```
DECLARE
```

```
salary_variable number(6);
```

“salary_variable” is a variable of datatype NUMBER and of length 6.

when a variable is specified as NOT NULL We must initialize the variable when it is declared.

For example: The below example declares two variables, one of which is a not null.

```
DECLARE
```

```
salary_variable number(4);
```

```
address varchar2(10) NOT NULL := “ajmer”;
```

A variable can changes its value in the execution or exception section of the PL/SQL Block when values assign to variables in the two ways given below.

- We can directly assign values to variables.

The Generic Syntax is:

```
variable_name:= value;
```

We can assign values to variables directly from the database columns by using a SELECT.. INTO statement

The Generic Syntax is:

```
SELECT Column_name INTO variable_name FROM table_name [where condition];
```

Example: The below program will get the salary of a teacher with name “radhakrishnan” and display it on the screen.

```
DECLARE
```

```
salary_var number(6);
```

```
tname_var char(15)=“radhakrishnan”;
```

```
BEGIN
```

```
SELECT salary INTO salary_var FROM teacher WHERE tname = tname_var;
```

```
dbms_output.put_line(salary_var);
```

```
dbms_output.put_line('The teacher '|| tname_var || 'has salary '|| salary_var);
```

```
END;  
/
```

NOTE: The backward slash '/' in the above program indicates to execute the above PL/SQL Block.

PL/SQL Constants:

If a value used in a PL/SQL Block remains unchanged throughout the program such value is basically a constant. A constant is a user-defined literal value. You can declare a constant and use it instead of actual value.

For example: If you want to write a program which will increase the salary of the teacher by 10%, you can declare a constant and use it throughout the program. Next time when you want to increase the salary again you can change the value of the constant which will be easier than changing the actual value throughout the program.

General Syntax to declare a constant is:

```
name_constant CONSTANT datatype=VALUE;
```

- name_constant is the name of the constant i.e. similar to a variable name.
- The word CONSTANT is a reserved word used to declare constant.
- VALUE - It is a value which must be assigned to a constant when it is declared. You cannot assign a value later.

For example, to declare increase_salary, you can write code as follows:

DECLARE

```
increase_salary CONSTANT number (4) := 10;
```

NOTE: You must assign a value to a constant at the time you declare it. If you do not assign a value to a constant while declaring it and try to assign a value in the execution section, you will get a error.

PL/SQL SET Serveroutput ON:

We must have to write the "SET Serveroutput ON" command when we start PL/SQL.

PL/SQL program execution start into Oracle engine so we always required to get serveroutput result and display into the screen otherwise result can't be display.

```
SQL> set serveroutput on
```

Let's discuss an example to understand the use of serveroutput result. The first line of the code turn on serveroutput then defining variables and constants. After defining placeholders dbms_output.put_line command is used to print defined variables values.

Example Code :

```
SQL> set serveroutput on
```

```
SQL> DECLARE
```

```
    Sno number(4) NOT NULL := 3
```

```
    Sname varchar2(14) := 'Hari';
```

```
    Sclass CONSTANT varchar2(10) := '9th';
```

```

BEGIN
    dbms_output.put_line('Declared Value:');
    dbms_output.put_line(' Student Number: ' || Sno || ' Student Name: ' || Sname);
    dbms_output.put_line('Constant Declared:');
    dbms_output.put_line(' student Class : ' || Sclass);
END;
/

```

Result display only if you execute "set serveroutput on" command.

Output of above code:

Declared Value:

Student Number: 3 Student Name: Hari

Constant Declared:

student Class: 9th

Conditional Statements in PL/SQL: The type of statements and their syntaxes is being discussed below.

IF THEN ELSE STATEMENT: In this IF-THEN–ELSE Statements, When the test condition is TRUE the statement 1 is executed and Statement 2 is skipped, when the test condition is FALSE, then Statement 2 is executed and statement 1 is skipped.

Syntaxes:

```

1)
IF condition
THEN
    statement 1;
ELSE
    statement 2;
END IF;

```

```

2)
IF condition 1
THEN
    statement 1;
    statement 2;
ELSIF condition2 THEN
    statement 3;
ELSE
    statement 4;
END IF

```

Iterative Statements in PL/SQL:

When we want to repeat the execution of one or more statements for specified number of times we use iterative control statements.

There are three types of loops in PL/SQL: The types of loops and their syntaxes in PL/SQL are.

Simple Loop
While Loop
For Loop

General Syntax for a Simple Loop : **LOOP**

```
Statements;  
EXIT;  
{or EXIT WHEN condition;}
```

END LOOP;

While using Simple Loop we should follow some important steps.

- We should always initialize a variable before the loop body.
- We should increment the variable in the loop.
- If we want to exit from the loop we should use a EXIT WHEN statement . If you use a EXIT statement without WHEN condition, the statements in the loop is executed only once.

While Loop

In a WHILE LOOP a set of statements executes till the condition is true. The condition is evaluated at the beginning of each iteration and remains continue until the condition becomes false.

The General Syntax for a WHILE LOOP :

```
WHILE <condition>  
LOOP statements;  
END LOOP;
```

FOR Loop

In a FOR LOOP a set of statements executes for a predetermined number of times. Iteration occurs between the start and end integer values given. The value of counter is always incremented by 1 and loop exits when the counter reaches the value of the end integer.

The General Syntax for a FOR LOOP:

```
FOR counter IN from...to  
LOOP statements;  
END LOOP;
```

- from - Start integer value.
- to - End integer value.

Important steps to follow when executing a for loop:

- The counter variable is implicitly declared in the declaration section, so it's not necessary to declare it explicitly.
- The counter variable is incremented by 1 and does not need to be incremented explicitly.
- EXIT WHEN statement and EXIT statements can be used in FOR loops but it's

not preferable.

PL/SQL Cursors: When a SQL statement executes, a temporary work area is to be created in the system memory called a cursor. The cursor contains information on a select statement and the rows of data accessed by it.

This temporary work area is used to store the data retrieved from the database, and manipulate this data. A cursor can hold more than one row, but can process only one row at a time. The set of rows the cursor holds is called the active set.

Types of cursors in PL/SQL: There are two types of cursors in PL/SQL

Implicit cursor: These are created by default when DML statements like, INSERT, UPDATE, and DELETE statements are executed. They are also created when a SELECT statement that returns just one row is executed.

Explicit cursor: They must be created when you are executing a SELECT statement that returns more than one row. Even though the cursor stores multiple records, only one record can be processed at a time, which is called as current row. When you fetch a row the current row position moves to next row. Both implicit and explicit cursors have the same functionality, but they differ in the way they are accessed.

Example of Implicit Cursors: When you execute DML statements like DELETE, INSERT, UPDATE and SELECT statements, implicit statements are created to process these statements. Some attributes called as implicit cursor attributes are provided to check the status of DML operations. These attributes are %FOUND, %NOTFOUND, %ROWCOUNT, and %ISOPEN.

For example, when you execute INSERT, UPDATE, or DELETE statements the cursor attributes tell us whether any rows are affected and how many have been affected. When a SELECT... INTO statement is executed in a PL/SQL Block, implicit cursor attributes can be used to find out whether any row has been returned by the SELECT statement. PL/SQL returns an error when no data is selected.

%FOUND Attribute: The return value is TRUE, if the DML statements like INSERT,DELETE and UPDATE affect at least one row and if SELECTINTO statement return at least one row. The return value is FALSE, if DML statements like INSERT, DELETE and UPDATE do not affect row and if SELECT....INTO statement do not return a row.

%NOTFOUND Attribute: The return value is FALSE, if DML statements like INSERT, DELETE and UPDATE affect at least one row and if SELECTINTO statement return at least one row. The return value is TRUE, if a DML statement like INSERT, DELETE and UPDATE do not affect even one row and if SELECTINTO statement does not return a row.

%ROWCOUNT Attribute: Gives the number of rows affected by the DML operations INSERT, DELETE, UPDATE and SELECT.

For Example: Consider the PL/SQL Block that uses implicit cursor attributes as

shown below:

```
DECLARE rows_var number(7);
BEGIN
  UPDATE Teacher
  SET salary = salary + 100;
IF SQL%NOTFOUND THEN
  dbms_output.put_line('salaries not updated');
ELSIF SQL%FOUND THEN
  rows_var := SQL%ROWCOUNT;
  dbms_output.put_line('Salaries for '||rows_var||' teachers are updated');
END IF;
END;
```

In the above PL/SQL Block, the salaries of all the teachers in the 'Teacher' table are updated. If none of the teacher's salary are updated we get a message 'salaries not updated'. Else we get a message like for example, 'Salaries for 100 teachers are updated' if there are 100 rows in 'Teacher' table.

Explicit Cursors: An explicit cursor is defined in the declaration section of the PL/SQL Block. It is created on a SELECT Statement which returns more than one row. We can assign a suitable name for the cursor.

General Syntax for creating a cursor :

```
CURSOR cursor_name IS select_statement;
```

- cursor_name – Name of the cursor.
- select_statement – A select query which returns multiple rows.

Steps to use an Explicit Cursor.

- DECLARE the cursor in the declaration section.
- OPEN the cursor in the Execution Section.
- FETCH the data from cursor into PL/SQL variables or records in the Execution Section.
- CLOSE the cursor in the Execution Section before you end the PL/SQL Block.

Stored Procedures: A named PL/SQL block which performs one or more specific task is called a stored procedure or in simple a proc.

A procedure has a header and a body. The header consists of the name of the procedure and the parameters or variables passed to the procedure. The body consists of a declaration section, execution section and exception section similar to a general PL/SQL Block. A procedure is similar to an anonymous PL/SQL Block but it is named for repeated usage. A procedure may or may not return any value.

Passing Parameters in Procedures: We can pass parameters to procedures in three ways.

- IN-parameters
- OUT-parameters
- IN- OUT-parameters

General Syntax to create a procedure :

```
CREATE [OR REPLACE] PROCEDURE proce_name [parameter list]
IS
    Declaration section
BEGIN
    Execution section
EXCEPTION
    Exception section
END;
```

IS - mark the beginning of the body of the procedure and is similar to DECLARE in anonymous PL/SQL Blocks. The code between IS and BEGIN forms the Declaration section.

The syntax within the brackets [] indicate they are optional. By using CREATE OR REPLACE together the procedure is created if no other procedure with the same name exists or the existing procedure is replaced with the current code.

Example

The below example creates a procedure 'student_info' which gives the information of the student.

```
1> CREATE OR REPLACE PROCEDURE student_info
2> IS
3>   CURSOR stu_cur IS
4>   SELECT roll_no, Sname, age FROM Student;
5>   stu_rec stu_cur%rowtype;
6> BEGIN
7>   FOR stu_rec in 1..7
8>   LOOP
9>     dbms_output.put_line(stu_cur.Roll_no || ' ' || stu_cur.Sname
10>    || ' ' || stu_cur.age);
11> END LOOP;
12> END;
13> /
```

There are two ways to execute a procedure.

- 1) From the SQL prompt.
EXECUTE [or EXEC] procedure_name;
- 2) Within another procedure – simply use the procedure name.
procedure_name;

PL/SQL Functions:

A function is also a named PL/SQL Block which is similar to a procedure. The major difference between a procedure and a function is that, a function must always return a

value, but a procedure may or may not return a value.

General Syntax for a function:

```
CREATE [OR REPLACE] FUNCTION func_name [parameters list]
RETURN return_datatype;
IS
Declaration_part
BEGIN
Execution_part
Return return_variable;
EXCEPTION
exception_part
Return return_variable;
END;
```

- **Return Type:** The header section defines the return type of the function. The return datatype can be any of the valid datatype such as varchar, number etc.
- The execution and exception section both should return a value which is of the datatype defined in the header section.

For example, let's create a function called "student_info_func" similar to the one created in stored proc.

```
1> CREATE OR REPLACE FUNCTION student_info_func
2> RETURN VARCHAR(10);
3> IS
4>   stu_name VARCHAR(20);
5> BEGIN
6>   SELECT Sname INTO stu_name
7>   FROM student WHERE Roll_no = '58';
8>   RETURN stu_name;
9> END;
10> /
```

In the example we are retrieving the 'Sname' of student with Roll_no 58 to variable 'stu_name'. The return type of the function is VARCHAR which is declared in line no 2. The function returns the 'stu_name' which is of type VARCHAR as the return value in line no 9.

Executing a function by.

- Since a function returns a value we can assign it to a variable.

```
student_name := student_info_func;
```

If 'student_name' is of datatype varchar we can store the name of the student by assigning the return type of the function to it.

- As a part of a SELECT statement

```
SELECT student_info_func FROM student;
```

- In a PL/SQL Statements like,
dbms_output.put_line(student_info_func);
This line displays the value returned by the function

Exception Handling: PL/SQL provides a feature to handle the Exceptions which occur in a PL/SQL Block known as exception Handling. Using Exception Handling we can test the code and avoid it from exiting abruptly. When an exception occurs, messages which explain its cause is received. PL/SQL Exception message consists of three parts.

- 1) Type of Exception
- 2) An ErrorCode
- 3) A message

By handling the exceptions we can ensure a PL/SQL block does not exit abruptly.

Exception Handling Structure

General Syntax for the exception section:

```

DECLARE
    Declaration part
BEGIN
    Exception part
EXCEPTION
WHEN excep1name THEN
    Error handling statements
WHEN excep2name THEN
    Error handling statements
WHEN Others THEN
    Error handling statements
END;
```

PL/SQL statements in the Exception Block. When an exception is raised, a search for an appropriate exception handler in the exception section starts. For example in the above example, if the error raised is 'excep1name ', then the error is handled according to the statements under it. Since, it is not possible to determine all the possible runtime errors during testing for the code, the 'WHEN Others' exception is used to manage the exceptions that are not explicitly handled. Only one exception can be raised in a Block and the control does not return to the Execution Section after the error is handled.

```

DECLCARE
    Declaration part
BEGIN
DECLARE
    Declaration part
```

```

BEGIN
    Execution part
EXCEPTION
    Exception part
END;
EXCEPTION
    Exception part
END;

```

If there are nested PL/SQL blocks as in the above case, if the exception is raised in the inner block it should be handled in the exception block of the inner PL/SQL block else the control moves to the Exception block of the next upper PL/SQL Block. If none of the blocks handle the exception the program ends abruptly with an error.

Types of Exception. There are 3 types of Exceptions.

- 1) Named System Exceptions
- 2) Unnamed System Exceptions
- 3) User-defined Exceptions

Named System Exceptions

System exceptions are automatically raised by Oracle, when a program violates a RDBMS rule. There are some system exceptions which are raised frequently, so they are pre-defined and given a name in Oracle which are known as Named System Exceptions.

For example: NO_DATA_FOUND and ZERO_DIVIDE are called Named System exceptions.

Named system exceptions are:

- 1) Not Declared explicitly,
- 2) Raised implicitly when a predefined Oracle error occurs,
- 3) Caught by referencing the standard name within an exception-handling routine.

For Example: Suppose a NO_DATA_FOUND exception is raised in a proc, we can write a code to handle the exception as given below.

```

BEGIN
    Execution part
EXCEPTION
WHEN NO_DATA_FOUND THEN
    dbms_output.put_line (' Using SELECT...INTO did not get any row. ');
END;

```

Unnamed System Exceptions

Those system exception for which oracle does not provide a name is known as unnamed system exception. These exceptions do not occur frequently. These Exceptions have a code and an associated message.

There are two ways to handle unnamed system exceptions:

1. Using the WHEN OTHERS exception handler, or
2. By associating the exception code to a name and using it as a named exception.

We can assign a name to unnamed system exceptions using a Pragma called EXCEPTION_INIT.

EXCEPTION_INIT will associate a predefined Oracle error number to a programmer defined exception name.

Steps to be followed to use unnamed system exceptions are

They are raised implicitly.

If they are not handled in WHEN others they must be handled explicitly.

To handle the exception explicitly, they must be declared using Pragma EXCEPTION_INIT as given above and handled referencing the user-defined exception name in the exception section.

The general syntax to declare unnamed system exception using EXCEPTION_INIT is:

```
DECLARE
  excep_name EXCEPTION;
  PRAGMA
  EXCEPTION_INIT (excep_name, Err_code);
BEGIN
  Execution part
EXCEPTION
  WHEN excep_name THEN
    handle the exception
END;
```

User-defined Exceptions

Apart from system exceptions we can explicitly define exceptions based on business rules. These are known as user-defined exceptions.

Steps to be followed to use user-defined exceptions:

They should be explicitly declared in the declaration section.

They should be explicitly raised in the Execution Section.

They should be handled by referencing the user-defined exception name in the exception section.

Triggers

Definition: A trigger is a PL/SQL block structure which is fired when a DML

statements like Insert, Delete, Update is executed on a database table. A trigger is triggered automatically when an associated DML statement is executed.

A database triggers has three parts-

1. **Triggering event**(That causes the trigger to be executed)
2. **Condition**(must be satisfied for trigger execution to proceed)
3. **Action**(specify the action to be taken when the trigger executes).

Trigger Syntax:

```
CREATE [OR REPLACE ] TRIGGER name_of_trigger
{BEFORE | AFTER | INSTEAD OF }
{INSERT [OR] | UPDATE [OR] | DELETE}
[OF name_of_col]
ON table_name
[REFERENCING OLD AS O NEW AS N]
[FOR EACH ROW]
WHEN (condition)
BEGIN
--- sql statements --
END;
```

- **CREATE [OR REPLACE] TRIGGER name_of_trigger** – In PL/SQL to creates a trigger with the given name or overwrites an existing trigger with the same name we use this clause.
- **{BEFORE | AFTER | INSTEAD OF }** - This clause indicates at what time should the trigger get fired. i.e for example: before or after updating a table. INSTEAD OF is used to create a trigger on a view. Before and after cannot be used to create a trigger on a view.
- **{INSERT [OR] | UPDATE [OR] | DELETE}** - This clause determines the triggering event. More than one triggering events can be used together separated by OR keyword. The trigger gets fired at all the specified triggering event.
- **[OF name_of_col]** - This clause is used with update triggers. This clause is used when you want to trigger an event only when a specific column is updated.
- **[ON table_name]** - This clause identifies the name of the table or view to which the trigger is associated.
- **[REFERENCING OLD AS O NEW AS N]** - This clause is used to reference the old and new values of the data being changed. By default, you reference the values as :old.column_name or :new.column_name. The reference names can also be changed from old (or new) to any other user-defined name. You cannot reference old values when inserting a record, or new values when deleting a record, because they do not exist.
- **[FOR EACH ROW]** - It is used to determine whether a trigger must fire

when each row gets affected (i.e. a Row Level Trigger) or just once when the entire sql statement is executed(i.e. a statement level Trigger).

- WHEN (condition) – It is valid only for row level triggers. The trigger is fired only for rows that satisfy the condition specified.

For Example: The classes of a student changes constantly. It is important to maintain the history of the classes of the students.

Types of PL/SQL Triggers

There are two types of triggers based on the level on which it is triggered

- 1) **Row level trigger** - An event is triggered for each row updated, inserted or deleted.
- 2) **Statement level trigger** - An event is triggered for each sql statement executed.

PL/SQL Trigger Execution Hierarchy

The following hierarchy is followed when a trigger is fired.

- 1) Firstly BEFORE statement trigger fires.
- 2) Next BEFORE row level trigger fires, once for each row affected.
- 3) Then AFTER row level trigger fires once for each affected row. This event will alternates between BEFORE and AFTER row level triggers.
- 4) Finally the AFTER statement level trigger fires.

Important Points:

- PL/SQL Block consists of the Declaration section, the Execution section and the Exception Handling section.
- We must have to write the "**SET Serveroutput ON**" command when we start PL/SQL.
- If you use a EXIT statement without WHEN condition, the statements in the loop is executed only once.
- A cursor can hold more than one row, but can process only one row at a time. The set of rows the cursor holds is called the active set.
- A procedure may or may not return any value.
- The major difference between a procedure and a function is that, a function must always return a value, but a procedure may or may not return a value.
- A trigger is triggered automatically when an associated DML statement is executed.

Practice Questions

Objective type questions:

- Q1. Which one is not the part of PL/SQL
- | | |
|------------|----------|
| a) Declare | b) BEGIN |
| c) Start | d) End |
- Q2. PL/SQL is developed by
- | | |
|--------|-----------|
| a) IBM | b) ORACLE |
|--------|-----------|

