Computer Programming – 1

Details of Syllabus:

Sr. No	Details	Hrs
1	Structure Programming using C++	5
1.1	C++ as a superset of C programming language	
1.2	C++ fundamental : Character set, Identifiers & Keywords, data types, Constants & Variables	
1.3	Declaration : Operators & Expressions, Library function statements, Symbolic constants, Preprocessor directives	
2	Data Input and Output & Control Statements	6
2.1	getchar( ), putchar( ), scanf( ), printf( ), gets( ), puts( ),cin, cout, setw( ), endl,	
2.2	If-else, while, do-while, goto, for, nested control structures, switch, break, continue statements, comma operator	
3	Function and Arrays	7
3.1	Function prototypes, passing arguments to a function by value and by reference, recursion overloading functions, storage classes	
3.2	Defining processing array, passing array to functions, introduction to multidimensional array and strings	
4	Pointers, Structure and Unions	5
4.1	Declarations. Referencing & dereferencing passing pointer functions , pointer to functions , pointer to arrays	
4.2	Structure & Unions. Defining and processing a structure	
4.3	Creation and manipulation of linked list	
5	Object Oriented Programming using C++	8
5.1	Classes, Objects, Data encapsulation, access specifiers, private, public and protected inheritance in details, operator overloading of unary and binary arithmetic operators, virtual functions, pure virtual functions	
6		6
6.1	Late binding, friend function, object as function parameter, overriding functions, overload constructors, copy constructors, static class member	

Programs to be implemented in the lab:

Sr. No	**Topics on which the programs have to be done in lab**
1.	If – else
2.	Reverse an integer & find sum of digits of an integer using do-while
3.	Factorial using do-while
4.	Drawing a pyramid using for
5.	Calculator using switch
6.	Function
7.	Recursive function
8.	Single dimensional array
9.	Matrix Multiplication using multidimensional array
10.	String Function Implementation
11.	Passing array to a function
12.	Passing pointer to a function
13.	Pointer to an array
14.	Processing a structure employee
15.	Class
16.	Array of objects
17.	Constructor, Destructor, Copy Constructor
18.	Single and Multiple Inheritance
19.	Hybrid Inheritance
20.	Function Overloading
21.	Unary Operator Overloading
22.	Binary Operator Overloading
23.	Friend function
24.	Static Class Member
25.	Virtual function

C++ Fundamentals:

Communicating with the computer involves speaking the language the computer understands which immediately rules out English as the language of communication with computer. So there is a close analogy between learning English and learning C/C++ language. The steps in learning English language are:

$$\begin{tikzcd}
\boxed{\text{Alphabets}} \arrow[r] & \boxed{\text{Words}} \arrow[r] & \boxed{\text{Sentences}} \arrow[r] & \boxed{\text{Paragraph}}
\end{tikzcd}$$

The steps in learning C/C++ language:

$$\begin{tikzcd}
\boxed{\begin{array}{l}\text{Alphabets}\\ \text{Digits \&}\\ \text{Special}\\ \text{Symbols}\end{array}} \arrow[r] & \boxed{\begin{array}{l}\text{Constants}\\ \text{Variables}\\ \text{Keywords}\end{array}} \arrow[r] & \boxed{\text{Instructions}} \arrow[r] & \boxed{\text{Program}}
\end{tikzcd}$$

C/C++ Character Set:

A character denotes any alphabets, digits or special symbols used to represent information.

Alphabets: A,B,C…..Y,Z

a,b,c……y,z

Digits: 0,1,2,………9

Special Symbols: ~,!,@,#,$,%,^,&,*,(,),_,+,=,-,{,},[,],<,>,?,\, /

Constants, Variables and Keywords:

The alphabets, numbers and special symbols when properly combined form constants, variables and keywords. A 'constant' is an entity that does not change where as variable is an entity that may change. So when we say x = 3 then x is the name of the variable and 3 is a constant.

$$\begin{tikzcd}
 & \boxed{\text{Constants}} \arrow[dl] \arrow[dr] & \\
\boxed{\text{Primary}} \arrow[d] & & \boxed{\text{Secondary}} \arrow[d] \\
\boxed{\begin{array}{l}\text{Integer}\\ \text{Real}\\ \text{Character}\end{array}} & & \boxed{\begin{array}{l}\text{Array}\\ \text{Pointer}\\ \text{Structure}\\ \text{Union}\\ \text{Enum}\end{array}}
\end{tikzcd}$$

Rules of constructing an Integer Constant:

1. At least one digit.
2. Must not have a decimal point.
3. Positive or negative but by default is taken as positive.
4. No commas and blanks are allowed with integer constant.
5. It ranges from -32767 to 32767 and unsigned int from 0 to 65535.

Example:

Rules for constructing a Real Constant:

1. At least one digit.
2. Must have a decimal point.
3. Positive or negative but by default is taken as positive.
4. No commas and blanks are allowed with integer constant.
5. The mantissa and the exponent should be separated by e.
6. The mantissa should have a positive or negative sign but default is positive.
7. It ranges from -3.4e38 to 3.4e38

Example:

Rules for constructing a Character Constant:

1. A character constant is a single alphabet, a single digit or a single specified symbol enclosed with single inverted commas. Both the inverted commas should point to the left.
2. The maximum length of a character constant can be one character.

Example: 'A','6','+'.

Variables:

An entity that may vary during program execution is called a variable. Variable names are given to locations in memory. These locations can contain integer, real or character constants. A particular type of variable can hold only the same type of constant.

Rules for constructing variable names:

1. A variable name is a combination of 1 to 31 alphabets, digits or underscores. Do not write unnecessary long names as it adds up to your typing effort.
2. The first character in a variable name should be an alphabet.
3. No commas or blanks are allowed in the variable name.
4. No special symbols other than underscores are allowed in a variable name.
5. Try giving some relevant variable names.

Example: si_int,m_hra,pop_e_89.

Next how do you declare variables?

```
int si_int;
float bas_sal;
char code;
```

Keywords:

Keywords are the words whose meaning has already been explained to the C/C++ compiler. The keywords cannot be used as variable names because if we do so then we are trying to assign new meaning to the keyword which is not allowed by the compiler. So the keywords are also called as Reserved words. There are 32 reserved keywords in C and more in C++.

The first C/C++ program:

1. Each instruction in a C/C++ is written as a separate statement.
2. The statements must appear in the same order in which we wish them to be executed.
3. Blank spaces may be inserted between two words to improve readability of the statement. However no blank spaces are allowed within a variable, constant and keyword.
4. All statements are entered in small case letters.
5. Comments of the program should be enclosed /* this is a C program*/

Example:/* Calculation of Simple Interest */

```
#include<stdio.h>
main()
{       int p,n;
        float si,r;
        p=1000;
        n=3;
        r=8.5;
        /* Formula for Simple Interest */
        si=(p*n*r)/100;
        printf("\nSimple Interest : %f",si);
}
/*Simple Interest : 255.000000*/
```

printf( ) function :

The general form of printf( ) function is:

printf("<format specifier>,<list of variables>");

<format string> can contain:

%f for printing real values

%d for printing integer values

%c for printing character values

Receiving Input:

scanf( ) is used as a function for receiving input.

Example: /* Calculation of Simple Interest */

```
#include<stdio.h>
main()
{
        int p,n;
        float si,r;
        printf("\n Enter the value of p,n,r :");
        scanf("%d %d %f",&p,&n,&r);
        /* Formula for Simple Interest */
        si=(p*n*r)/100;
        printf("\nSimple Interest : %f",si);
}
/* Enter the value of p,n,r :2000 5 10.5
Simple Interest : 1050.000000*/
```

C Instructions:

There are basically three types of instructions in C/C++:

1. Type Declaration Statements:

This instruction is used to declare the type of variables being used in the program. Any variable used in the program must be declared before using it in any statement. The type declaration statement is written at the beginning of the main( ) function.

```
Ex:     int bas;
        float rs, grossal;
        char name, code;
```

a) While declaring the type of variable we can also initialize it as shown below:

   int i = 10, j=25;

   float a = 1.5, b=1.99+2.4 * 1.44;

b) The order in which we define the variables is sometimes important sometimes not. For example :

   int i = 10, j=25; is same as

   int j=25, i=10; However,

   float a = 1.5, b= a+ 3.1; is alright but

   float b = a + 3.1, a =1.5 is not. This is because here we are trying to use ‘a’ even before defining it.

c) The following statements would work:

   int a, b, c, d;

   a = b = c = d =10;

2. Arithmetic Instructions:

A C/C++ instruction consists of a variable name on the left hand side of = and names and constants on the right hand side of = are connected by arithmetic operators like +, - , * and /.

Example:
```
int ad;
        float kot, delta, alpha,beta,gamma;
          ad=3200;
        kot = 0.0056;
        delta = alpha  * beta / gamma + 3.2 * 2 / 5;
```

Here,

- *, /, -,+ are the arithmetic operators.
- = is the assignment operator
- 2, 5, 3200 are integer constants.
- 3.2 and 0.0056 are real constants.
- ad is an integer variable. kot, delta, alpha, beta, gamma are real variables

Hierarchy of Operators:

Priority	Operators	Description
1st	*, /, %	Multiplication, Division, Modular Division
2nd	+, -	Addition, Subtraction
3rd	=	Assignment

Example:

I = 2 * 3 / 4 + 4 / 4 + 8 – 2 + 5 / 8

Stepwise evaluation of this expression is shown below:

I = 2 * 3 / 4 + 4 / 4 + 8 – 2 + 5 / 8

I = 6 / 4 + 4 / 4 + 8 – 2 + 5 / 8

I = 1 + 4 / 4 + 8 – 2 + 5 / 8

I = 1 + 1 + 8 – 2 + 5 / 8

I = 1 + 1 + 8 – 2 + 0

I = 2 + 8 – 2 + 0

I = 10 – 2 + 0

I = 8 + 0

I = 8

Associativity of Operators:

If both the operators have got same priority then the left to right associativity should be taken.

3. Control Instructions in C/C++:

They are:

a) Sequence Control Instruction

b) Selection or Decision Control Instruction

c) Case Control Instruction

d) Repetition or Loop Instruction

Exercise:

1. **Which one of the following are invalid variable names:**

Basicsalary	_basic	Basic-hra	#mean
Group.	422	Population in 2006	Over time
Meter	Float	Hello	Team’svictory

2. **What will be the output of the following programs?**

```
#include<stdio.h>
main()
{
 int i=2,j=3,k,l;
 float a,b;
 k=i/j*j;
 l= j/i*i;
 a=i/j*j;
 b=j/i*i;
 printf("%d %d %f %f",k,l,a,b);
}
```

3. **A C program contains the following declarations and initial assignments:**

   int i = 8, j = 5;

   float x = 0.005, u = 0.01;

   char c= ‘c’, d= ‘d’;

   Determine the value of the following expressions. Use the values initially assigned to the variables for each expression.

   i) (3 * I – 2 * j) % (2 * d – c)

   ii) 2* x + (y = = 0)

   iii) –(i + j)

   iv) C > d

   v) (x >y) && ((i >0||(j<5))

4. **What is the output of the following program?**

```
i) main()
{         int x=10,y,z;
 z= y =x;
 y-=x--;
 z-=--x;
 x-=--x-x--;
 printf(" y=%d z = %d x = %d",y,z,x);
}
ii) main()
{         float a =0.5, b= 0.9;
 if(a && b > 0.9)
          printf("if part");
 else
          printf("else part");
 }
iii)main()
{         int a,b,c,d,e;
 a=b=c=d=e=40;
 printf("%d %d %d %d %d",a,b,c,d,e);
}
```

5. What is the output of the following expressions if a = 1, b= 2, c=3. Also give the contents of a, b and c at the end of expression:
   a. a = a + + + - - b
   b. b = b + c - - * - - a
6. If x, y and z are integer variable then evaluate the following expressions and give the final values of x, y and z. x= 10, y= 6, z= 8.
   a. x = = y + z
   b. x - - + y - - + - -z / (x * x)
   c. - - x + y - - * - - z
7. Write the following programs in C:

a) If marks obtained by a student in 5 subjects are input through a keyboard find the aggregate marks and percentage obtained by the student.

b) Temperature of a city in Fahrenheit is input through the keyboard. Write a program to convert this temperature into centigrade degrees.

c) The length and breadth of a rectangle and radius of circle are input through the keyboard. Write a C program to calculate area and perimeter of the rectangle and the area and circumference of the circle.

d) Two numbers are input through the keyboard into two locations C and D. Write a program to interchange the contents of C and D with and without using the third variable.

8. Write a short note on operators?

Chapter 2.1 : Console Input / Output functions:

Formatted Functions				Unformatted Functions		
Type	Input	Output		Type	Input	Output

char	scanf( )	printf( )		char	getch( )
					getche( )
					getchar( )
						putch( )
						putchar( )

int	scanf( )	printf( )		int	-	-
float	scanf( )	printf( )		float	-	-
string	scanf( )	printf( )		string	gets( )	puts( )

Format Specifications:

Data type		Format Specifier
Integer	Short signed	%d or %i
	Short unsigned	%u
	Long signed	%ld
	Long unsigned	%lu
	Unsigned Hexadecimal	%x
	Unsigned Octal	%o

Implemet getchar( ), getch( ), getche( ):

```
#include<stdio.h>
#include<conio.h>
main()
{
        char ch;
        printf("\nPress any key to continue :");
        getch();
        printf("\nType any character :");
        ch=getche();
        printf("\nYou have typed %c",ch);
        printf("\nType any character :");
        getchar();
}
/*
Press any key to continue :
Type any character :y
You have typed y
Type any character :h*/
```

Chapter 2.2: Decision Control Structure:

In the previous cases we have executed sequential instructions i.e. they are executed in the order in which they appear. But if we want the instructions to follow a sequence pattern depending on the condition then we use decision control structure. There are relational operators used to check the condition.

This expression	Is true if
x = = y	X is equal to y
x ! = y	X is not equal to y
x < y	X is less than y
x > y	X is greater than y
x < = y	X is less than or equal to y
x > = y	X is greater than or equal to y

if statement:

The general form of if statement looks like this:

if (this condition is true)

Execute this statement

Example:

```
#include<stdio.h>
main()
{
        int num;
        printf("\nEnter the number :");
        scanf("%d",&num);
        if(num<10)
                printf("Good data is correct");
}
/*Enter the number :12
Enter the number :4
Good data is correct*/
```

Example:

While purchasing certain items a discount of 10% is offered if the quantity purchased is more than 100. The quantity and price per item is input through the keyboard, write a program to calculate the total expenses.

```
#include<stdio.h>
main()
{
        int qty,rate,dis=0;
        float total;
        printf("\nEnter the qty and rate :");
        scanf("%d %d",&qty,&rate);
        if(qty>100)
                dis=10;
        total=(qty*rate)-(qty*rate*dis/100);
        printf("\nTotal expenses : %f",total);
}
/*Enter the qty and rate :90 15
Total expenses : 1350.000000
Enter the qty and rate :105 15
Total expenses : 1418.000000*/
```

If-Else Statement:

The expression following if evaluates to true and performs else part if it evaluates to false.

Syntax:

```
if (condition)
        true statements;
else
        false statements;
```

Example:

If an employee's basic salary is less than Rs.1500 then his HRA = 10% of basic salary and DA = 90% of the basic salary. If the salary is either equal to or above Rs. 1500 then HRA = Rs. 500 and DA = 98% of the basic salary. If the employee's salary is input through keyboard then write a program to find gross salary.

```
#include<stdio.h>
main()
{
        float bs,gs,da,hra;
        float total;
        printf("\nEnter basic salary :");
        scanf("%f",&bs);
        if(bs<1500)
        {
                hra=bs*10/100;
                da=bs*90/100;
        }
        else
        {
                hra=500;
                da=bs*98/100;
        }
        gs=bs+hra+da;
        printf("\nBasic salary : %f",bs);
        printf("\nHRA          : %f",hra);
        printf("\nDA           : %f",da);
        printf("\nGross salary : %f",gs);
}
/*
Enter basic salary :2000
Basic salary : 2000.000000
HRA          : 500.000000
DA           : 1960.000000
Gross salary : 4460.000000
Enter basic salary :1400
Basic salary : 1400.000000
HRA          : 140.000000
DA           : 1260.000000
Gross salary : 2800.000000*/
```

Write a program to calculate the salary as per the following table:

Gender	Years of service	Qualification	Salary
Male	> = 10	Post graduate	15000
	> = 10	Graduate	10000
	< 10	Post graduate	10000
	< 10	Graduate	7000
Female	> = 10	Post graduate	12000
	> = 10	Graduate	9000
	< 10	Post graduate	10000
	< 10	Graduate	6000

```
#include<stdio.h>
main()
{
        char g;
        int yos,qual,sal=0;
        printf("\nEnter the gender, years of service :");
        scanf("%c %d",&g,&yos);
        printf("\nEnter qualification(0 for grad & 1 for postgrad) :");
        scanf("%d",&qual);
        if(g=='m' &&yos>=10 && qual==1)
                sal=15000;
        else if(g=='m' &&yos>=10 && qual==0)
                sal=10000;
        else if(g=='m' &&yos<10 && qual==1)
                sal=10000;
        else if(g=='m' &&yos<10 && qual==0)
                sal=7000;
        else if(g=='f' &&yos>=10 && qual==1)
                sal=12000;
        else if(g=='f' &&yos>=10 && qual==0)
```

```
                sal=9000;
        else if(g=='f' &&yos<10 && qual==1)
                sal=10000;
        else if(g=='f' &&yos<10 && qual==0)
                sal=6000;
        printf("\nSalary of employee :%d",sal);
}
/*Enter the gender, years of service :m 10
Enter qualification(0 for grad & 1 for postgrad) :1
Salary of employee :15000*/
```

Write a program to find greatest of three numbers:

```
#include<stdio.h>
main()
{
        int a,b,c;
        printf("\nEnter the three numbers :");
        scanf("%d %d %d",&a,&b,&c);
        if(a>b && a>c)
                printf("%d is greatest",a);
        else if(b>a && b>c)
                printf("%d is greatest",b);
        else if(c>b && c>a)
                printf("%d is greatest",c);
}
/*
Enter the three numbers :56 78 34
78 is greatest*/
```

Write a program to print whether a number is even or odd.

```
#include<stdio.h>
main()
{
        int a,b,c;
        printf("\nEnter a number :");
        scanf("%d",&a);
        if(a%2==0)
                printf("%d is even number",a);
        else
                printf("%d is odd number",a);
}
/*
Enter a number :67
67 is odd number
Enter a number :44
44 is even number*/
```

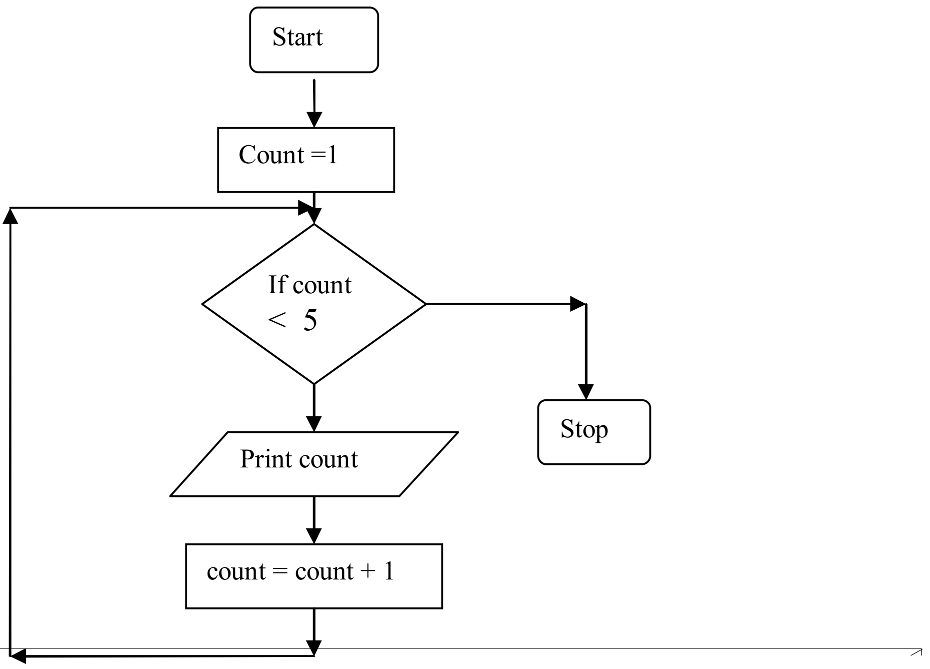
Loop Control Structure:

The versatility of the computer lies in its ability to perform a set of instructions repeatedly. This involves repeating some portion of the program either specified number of times or until a particular condition is specified. This repetitive operation is done through a loop control instruction.

There are three methods by way of which we can repeat a part of the program. They are:

i. **Using a while statement**
ii. **Using a do-while statement**
iii. **Using a for statement**

WhileLoop:

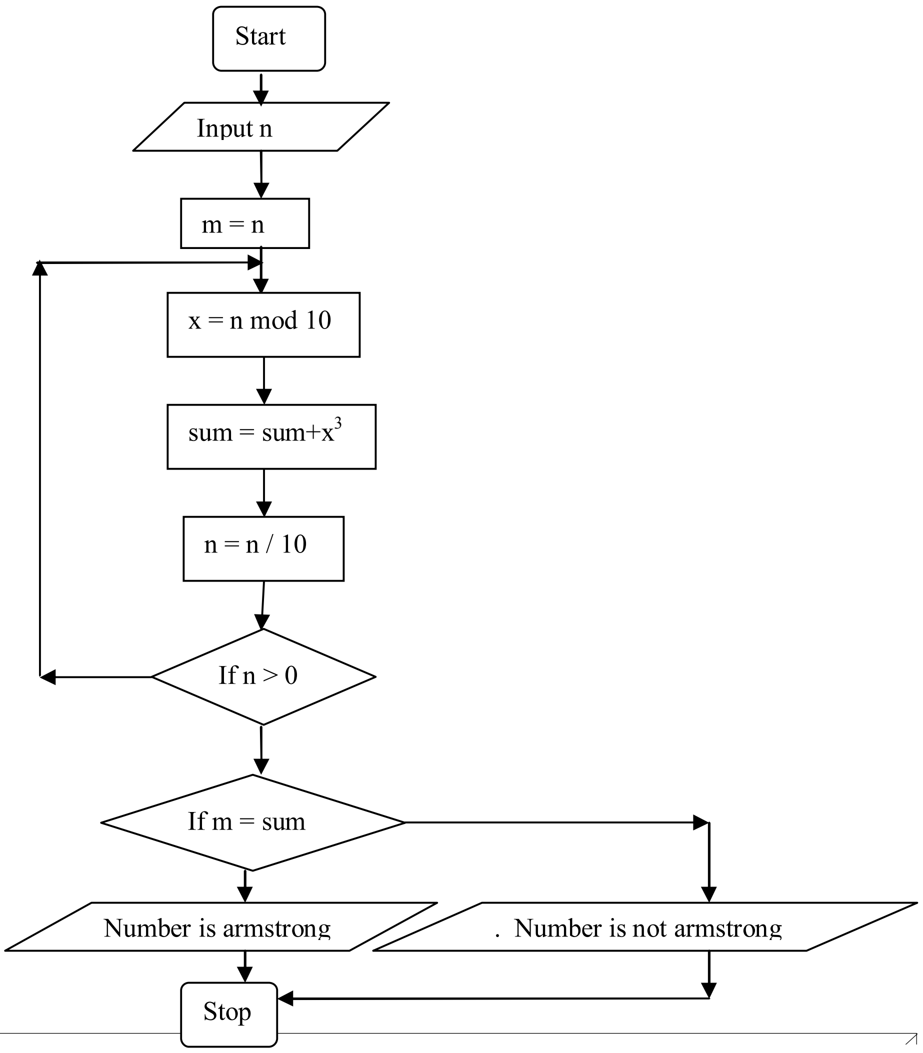


The above flowchart will print numbers from 1 to 5. Let us see how we implement in C.

```
#include<stdio.h>
main()
{
        int i=1,count;
        printf("\nEnter n :");
        scanf("%d",&count);
        while(i<=count)
        {       printf("%d ",i);
                i++;
        }
}
/*Enter n :7
1 2 3 4 5 6 7*/
```

The Do-While Loop:

With the help of a flowchart using do-while loop we will find out whether a number is an Armstrong number

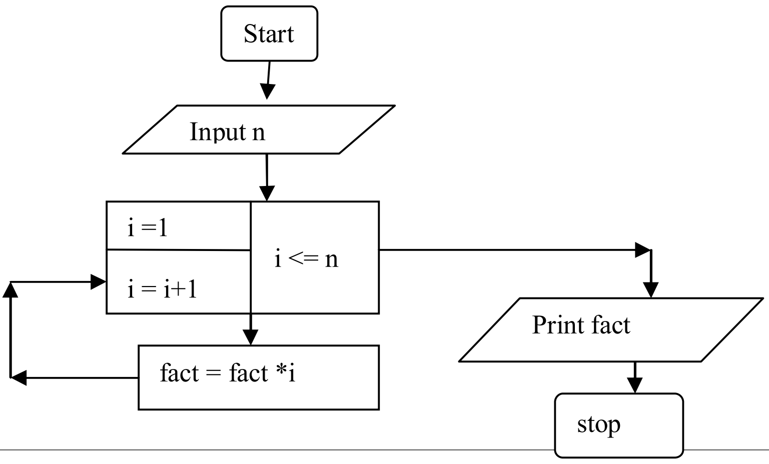


Implementation In C:

```
#include<stdio.h>
main()
{
        int n,m,x,sum=0;
        printf("\nEnter the n:");
        scanf("%d",&n);
        m=n;
        do
        {       x=n%10;
                sum=sum+x*x*x;
                n=n/10;
        }
        while(n>0);
        if(m==sum)
                printf("\nThe %d is an armstrong number",m);
        else
                printf("\nThe %d is not an armstrong number",m);
}
/* Enter the n:153
The 153 is an armstrong number
Enter the n:133
The 133 is not an armstrong number*/
```

For Loop:

With the help of a flowchart implement factorial of a number.



```
#include<stdio.h>
main()
{
        int i,n,fact=1;
        printf("\nEnter n :");
        scanf("%d",&n);
        for(i=1;i<=n;i++)
        {
                fact=fact*i;
        }
        printf("\n%d factorial is %d",n,fact);
}
/* Enter n :5
5 factorial is 120*/
```

The goto statement:

Write a program to read an integer from keyboard and if it is an even number divide by 2 and if odd multiply by 3 add 1 to it. Display all intermediate values until the number converges to 1. Also count and display the number of iterations required for converge.

```
#include<stdio.h>
main()
{
        int x;
        printf("\n Enter the value of x :");
        scanf("%d",&x);
        l: if(x%2==0)
        {
                x=x/2;
                printf("%d  ",x);
        }
        else
        {
                x=x*3+1;
        }
         if(x>1)
                goto l;
}
/*Enter the value of x :7
11  17  26  13  20  10  5  8  4  2  1
 Enter the value of x :18
9  14  7  11  17  26  13  20  10  5  8  4  2  1*/
```

The break statement:

We often come across situations where we have to jump out of loop instantly without waiting to get back to the conditional test. The keyword break allows us to do this. When the break is encountered inside any loop, control automatically passes to the first statement after the loop. A break is usually associated with an 'if'.

Example:

Write a program to determine whether a number is prime or not.

```
#include<stdio.h>
main()
{
        int i,n;
        printf("\n Enter the value :");
        scanf("%d",&n);
        for(i=2;i<n;i++)
        {
                if(n%i==0)
                        break;
        }
        if(i==n)
                printf("\n %d is a prime no",n);
        else
                printf("\n %d is not a prime no",n);
}
/*
Enter the value :15
 15 is not a prime no
 Enter the value :17
 17 is a prime no*/
```

Implementation of nested loops:

```
#include<iostream.h>
main()
{
	int r,c,sum,i;
	for(r=1;r<=3;r++)
	{	for(c=1;c<=3;c++)
		{	cout<<"\nr="<<r<<" c="<<c<<" sum="<<r+c;
		}
	}
}
/*r=1 c=1 sum=2
r=1 c=2 sum=3
r=1 c=3 sum=4
r=2 c=1 sum=3
r=2 c=2 sum=4
r=2 c=3 sum=5
r=3 c=1 sum=4
r=3 c=2 sum=5
r=3 c=3 sum=6*/
```

1. Write a program to generate the following pattern:

a)

*	*	*	*	*
*	*	*	*	*
*	*	*	*	*
*	*	*	*	*
*	*	*	*	*

```
#include<iostream.h>
main()
{	int r,c,sum,i;
	for(r=1;r<=5;r++)
	{	for(c=1;c<=5;c++)
		{	cout<<"* ";
		}	cout<<endl;
	}
}
/*
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
*/
```

b)

*				
*	*			
*	*	*		
*	*	*	*	
*	*	*	*	*

```
#include<iostream.h>
main()
{         int r,c,sum,i;
          for(r=1;r<=5;r++)
          {
                    for(c=1;c<=r;c++)
                    {
                              cout<<"* ";
                    }
                    cout<<endl;
          }
}/*
*
* *
* * *
* * * *
* * * * *
*/
```

c)

*				*
	*		*	
		*		
	*		*	
*				*

```
#include<iostream.h>
main()
{         int r,c,sum,i;
          for(r=1;r<=6;r++)
          {         for(c=1;c<=6;c++)
                    {         if(r==c||(r+c)==7)
                                        cout<<"*";
                              else
                                        cout<<" ";
                    }
                    cout<<endl;
          }
}/*
*    *
 *  *
  **
  **
 *  *
*    *
*/
```

d)

1				
1	**2**			
1	**2**	**3**		
1	**2**	**3**	**4**	
1	**2**	**3**	**4**	**5**

```
#include<iostream.h>
main()
{
        int r,c,sum,i;
        for(r=1;r<=5;r++)
        {
                for(c=1;c<=r;c++)
                {
                                cout<<c<<" ";
                }
                cout<<endl;
        }
}
/*
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5*/
```

e)

1				
2	**2**			
3	**3**	**3**		
4	**4**	**4**	**4**	
5	**5**	**5**	**5**	**5**

```
#include<iostream.h>
main()
{
        int r,c,sum,i;
        for(r=1;r<=5;r++)
        {
                for(c=1;c<=r;c++)
                {
                                cout<<r<<" ";
                }
                cout<<endl;
        }
}
/*1
  2 2
  3 3 3
  4 4 4 4
  5 5 5 5 5*/
```

f)

1					
1	**1**				
1	**2**	**1**			
1	**3**	**3**	**1**		
1	**4**	**6**	**4**	**1**	
1	**5**	**10**	**10**	**5**	**1**

```
#include<iostream.h>
main()
{
        int x,m,b;
        for(m=0;m<=5;m++)
        {
                for(x=0,b=1;x<=m;x++)
                {
                        if(m==0||x==0)
                                cout<<b<<" ";
                        else
                        {
                                b=b*(m-x+1)/x;
                                cout<<b<<" ";
                        }
                }
                cout<<endl;
        }
}
/*
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1*/
```

Write a program to print different series taking minimum value of terms as 5:

a) $\sin(x) = x - x^3/3! + x^5/5! - x^7/7! + x^9/9!$

```
#include<iostream.h>
#include<math.h>
main()
{
        int i,n,k,j=1,s=1;
        float x,sum=0,num,den;
        cout<<"\nEnter no.of terms :";
        cin>>n;
        cout<<"\nEnter x :";
        cin>>x;
        x=x*3.14/180;
        for(k=1,i=1;k<=n;k++)
        {
                num=pow(x,i);
                den=1;
                j=1;
                while(j<=i)
                {
                        den=den*j;
                        j++;
                }
                sum=sum+(num/den)*s;
                s=s*(-1);
                i=i+2;
        }
        cout<<"\nSum of sine series :"<<sum;
}
/*
Enter no.of terms :5
Enter x :30
Sum of sine series :0.49977
Enter no.of terms :5
Enter x :60
Sum of sine series :0.86576
Enter no.of terms :5
Enter x :90
Sum of sine series :1.000003*/
```

Exercise:

1. When interest compounds ‘q’ times per year at an annual rate of r% for n years, the principal compounds to an amount ‘a’ as per the following formula: $A = p *(1+r/q)^{nq}$
2. Write a program to read 10 sets of ‘p’,’n’,’r’ and ‘q’ and calculate the corresponding ‘a’ ‘s.
3. The natural algorithm can be approximated by the following series:

   $((x-1)/x) +(1/2) ((x-1)/x)^2 + (1/2) ((x-1)/x)^3 +(1/2) ((x-1)/x)^4$

   If x is input through the keyboard, write a program to calculate the sum of first seven terms of the series.

8. Write a program to find out whether a number is a palindrome or not.
9. Write a program to generate
   a) $^nP_r = n!/(n-r)!$
   b) $^nC_r = n!/(r!)(n-r)!$
10. Write a program to read a number, reverse the number and display the sum of digits of number.
11. Two numbers are entered from the keyboard. Find out the output when one is raised on to the other.
12. Write a program to print all ASCII values and their equivalent from 0 to 255.
13. Write a program to enter numbers & count no. of positive and negative numbers entered.
14. Write a program to calculate the octal equivalent of the received number.
15. Write a program to print all prime numbers from 1 to 300.
16. Write a program to find the sum of seven terms using a for loop for the following series:

    1/1!+2/2!+3/3!+………..+7/7!

17. Write a program to print the multiplication table of the number entered by the user. The table should be displayed in the following form:

    29 * 1 = 29

    …………….

    29 * 10 = 290
18. Write a program to print different series taking minimum value of terms as 5:
    b) cos (x) = 1- x^2/2!+ x^4/4!- x^6/6!+ x^8/8!
    c) e^x = 1 + x + x2/2! +x^3/3! +x^4/4!
19. Write a program to print Fibonacci series.
20. Write a program which reads 3 integers and shall output 3 integers in decreasing order.
21. Write a program which reads three sides of a triangle and shall print out whether a triangle is a right angle triangle or not.
22. Write a program to solve a quadratic equation ax^2+bx+c
23. Write a program to find G.C.D and L.C.M for two numbers entered by the user.
24. Write the menu driven program to perform the trigonometric functions

    S ---sin(x)   C ---cos(x)   T ---tan(x)   X ---exit
25. Write a program which reads and multiplies together 20 real numbers.
26. Write a program that asks the user for an integer that prints its entire factors say for example when the use enters 150 the program should print 2, 3, 5….
27. Write to compute the distance S fallen by an object in free fall a program. The formula is $S = S_0+V_0t+1/2at^2$. Make a table of S for t = 1, 5, 10, 15, …100.
28. Write a program to read two natural numbers r1 and r2 where r2 is greater than r1and generate all prime numbers between r1 and r2.
29. Write a program which finds four digit perfect squares where the number represented by first two digits and last two digits are perfect squares. For ex :1681 = 49^2, 16 =4^2 ,81 = 9^2
30. A famous conjecture holds that all positive integers converge to 1 when treated in the following fashion.

    Step 1: If number is odd it is multiplied by three and one is added

    Step 2 : If the number is even it is divide by 2.
31. Write a program to perform the following operations:
    a) Display any number of stars on the screen by using a loop.
    b) Display the menu containing the following:

       i) Whole screen   ii) Half screen   iii) Top 3 lines iv) Bottom three lines
32. Write a program to display the number of days in the calendar format for the year 2005.

Chapter 3.1 Functions

A function is self contained block of statements that perform a coherent task of some kind. Every C program could be thought of as a collection of these functions. The function main ( ) is executed first and it calls the other function directly or indirectly. It is necessary that every program must have the main ( ). We may have user defined functions which can be invoked from other pats of the program. Functions are building blocks of C programs where all the program activity occurs. As we have noted earlier, using a function is something like hiring a person to do a specific job for you. Let us look at two things a function that calls or activates the function and the function itself.

Let us consider the following program:

```
#include<iostream.h>
void message();
main()
{
        message();
        cout<<"\nCry and you stop the monotony";
}
void message()
{
        cout<<"\nSmile the world smile's back at you";
}
/*
Smile the world smile's back at you
Cry and you stop the monotony*/
```

There are three categories of functions:

1. Functions with no arguments and no return values:

```
#include<iostream.h>
#include<math.h>
void printstar();
void invest();
main()
{
        cout<<endl<<endl;
        printstar();
        invest();
        printstar();
}
void printstar()
{
        int i;
        for(i=1;i<40;i++)
                cout<<"*";
}
```

```
void invest()
{
        float p,v,r;
        int n;
        cout<<"\nEnter p,n,r :";
        cin>>p>>n>>r;
        v=p*(pow((1+r/100),n));
        cout<<"\nValue of inverstent is :"<<v<<endl;
}
/*
*************************************
Enter p,n,r :1000 5 5

Value of inverstent is :1276.281616
*************************************/
```

2. Function having arguments and no return value:

```
#include<iostream.h>
void power(int,int);
main()
{
        int x,n;
        cout<<"\nEnter the value of x and n :";
        cin>>x>>n;
        power(x,n);
}
void power(int x,int n)
{
        int i,res=1;
        for(i=0;i<n;i++)
                res=res*x;
        cout<<"The result is : "<<res;
}
/*
Enter the value of x and n :3 7
The result is : 2187*/
```

3. Function having arguments and return value:

```
#include<iostream.h>
int result(int,int,char);
main()
{
        int a,b,c;
        char op;
        cout<<"\nEnter a and b :";
        cin>>a>>b;
        cout<<"\nEnter the operator :";
        cin>>op;
        c=result(a,b,op);
        cout<<"The result is : "<<c;
}
```

```
int result(int a,int b,char op)
{
        int res;
        switch(op)
        {
                case '+':res=a+b;
                                        break;
                case '-':res=a-b;
                                        break;
                case '*':res=a*b;
                                        break;
                case '/':res=a/b;
                                        break;
                default:cout<<"\nWrong operator";
        }
        return(res);
}
/*
Enter a and b :78 5
Enter the operator :*
The result is : 390*/
```

Recursion:

In C a function can call itself this is called as recursion. A function is said to be recursive if there exists a statement in its body for the function call itself. For example the Fibonacci terms are 0,1,1,2….In this sequence each term except the first two are obtained by adding its two immediate predecessor terms. The recursive definition of the sequence is:

fib(n) = 0 if n = 1

= 1 if n = 2

fib(n-1) + fib(n -2) if n > 2

While coding recursive function, we must take care that there exists a reachable termination condition inside the function so that function may not be invoked endlessly. The following program illustrates the use of recursion.

```
#include<stdio.h>
int fib(int n)
{
        if(n==0 || n==1)
                return(1);
        else
                return(fib(n-1)+fib(n-2));
}
main()
{
        int x,i;
        printf("\nEnter the number of terms :");
        scanf("%d",&x);
        for(i=0;i<x;i++)
                printf("%d ",fib(i));
}
/*Enter the number of terms :10
1 1 2 3 5 8 13 21 34 55*/
```

Write a program to find sum of digits of an integer.

```
#include<iostream.h>
int sum(int,int);
main()
{
        int a,b;
        cout<<"\nEnter a : ";
        cin>>a;
        b=sum(0,a);
        cout<<"\nThe sum of digits is :"<<b;
}
int sum(int s,int n)
{
        if(n==0)
                return(s);
        else
        {
                int d=n%10;
                s=s+d;
                return(sum(s,n/10));
        }
}
/*Enter a : 4587
The sum of digits is :24*/
```

Exercise:

1. Write a program which has a function that checks whether a number is prime or not.
2. Write a program to include a function which finds G.C.D and L.C.M of two variables using recursive and non-recursive functions.
3. Write a program to convert a number into binary, octal and hexadecimal using functions.
4. Write a function swap to interchange value of two variables with and without using a third variable. Call this function in main ( ).
5. Write a function to find reverse a number using recursive functions.
6. Write a recursive function to compute x ^{n} where x and n are taken as integers.
7. Write a program to find the value of BIO = n! /r! * (n-r)!
8. Write a recursive function to implement Fibonacci series for n=10.

Chapter 3.2: Array

The C language provides a capability that enables the user to design a set of similar data types called array. Let us consider the following program:

The above program has printed the value of as x =10. Because when the value 10 has been assigned to x, the earlier value of x i.e. 5 is lost. Thus ordinary variables are capable of holding only one value at a time. However there are situations in which we want to store more than one value at a time in a single variable. For example we want to arrange the percentage of marks in ascending order. So in order to store this data we have to construct 100 variables and the other is to construct an array which can store 100 values.

So we define Array as a collective name given to a group of 'similar quantities'. These similar quantities could be percentage of marks salaries of 300 employees and age of 50 employees etc. Each member in the group is referred to by the position in the group. For example assume the following group numbers which represent the percentage of marks obtained by 5 students. per = {48, 88, 34, 23, 96}.

If we want to refer to the second number then we will say per[1] because an array by default starts from the position 0. So the general notion would be per[i] where I can take value from 0, 1,2,3,4 depending upon the position being referred. Here per is the subscripted variable and 'i' is its subscript. An array is a collection of all ints, floats and chars; usually the array of characters is called a string whereas an array of ints and floats is called as a simple array.

Let us try to write a program to find the average of marks obtained by student:

```
#include<iostream.h>
main()
{
        int sum=0,n;
        int i,marks[10];
        cout<<"\nEnter the number of students :";
        cin>>n;
        for(i=0;i<n;i++)
        {
                cout<<"\nEnter marks :";
                cin>>marks[i];
                sum=sum+marks[i];
        }
        cout<<"\nSum of marks is :"<<sum;
        cout<<"\nAverage of marks is :"<<sum/n;
}
/*
Enter the number of students :5
Enter marks :56
Enter marks :78
Enter marks :89
Enter marks :45
Enter marks :56
Sum of marks is :324
Average of marks is :64*/
```

Array Initialization:

In the previous cases we have used an array that did not have any values in them to begin with. We managed to store values in them during program execution. Let us now see how we initialize an array while declaring it. Following are the few examples that demonstrate this:

```
int num[6] = {2,4,12,6,45,5};
int n[]={2,4,12,6,45,5};
float press[]={2.3,34.2,-23.4,-11.3};
```

Passing array elements to a function:

Array elements can be passed to a function by calling the function by value or by reference. In the call by value we pass the value of array elements and in call by reference we pass addresses of array elements to the function. Let us consider a program:

```
#include<iostream.h>
int fun(int[],int);
main()
{
        int sum=0,n;
        int i,marks[10];
        cout<<"\nEnter the number of students :";
        cin>>n;
        for(i=0;i<n;i++)
        {
                cout<<"\nEnter marks :";
                cin>>marks[i];

        }
        sum=fun(marks,n);
        cout<<"\nSum of marks is :"<<sum;
        cout<<"\nAverage of marks is :"<<sum/n;
}
int fun(int x[],int n)
{
        int s=0;
        for(int i=0;i<n;i++)
        {
                s=s+x[i];
        }
```

```
	return(s);
}
/*Enter the number of students :5
Enter marks :56
Enter marks :67
Enter marks :78
Enter marks :23
Enter marks :45
Sum of marks is :269
Average of marks is :53*/
```

Write a program to sort an array in ascending order:

```
#include<iostream.h>
main()
{
	int sum=0,n;
	int i,x[10];
	cout<<"\nEnter the number of elements :";
	cin>>n;
	cout<<"\nEnter data :";
	for(i=0;i<n;i++)
	{
		cin>>x[i];
	}
	for(i=0;i<n-1;i++)
	{
		for(int j=i+1;j<n;j++)
		{
			if(x[i]>x[j])
			{
				int temp=x[i];
				x[i]=x[j];
				x[j]=temp;
			}
		}
	}
	cout<<"\nThe sorted array is :"<<endl;
	for(i=0;i<n;i++)
		cout<<x[i]<<" ";
}
/*
Enter the number of elements :7
Enter data :78 89 76 54 12 34 23
The sorted array is :
12 23 34 54 76 78 89*/
```

Exercise:

1. Write a program to enter 10 numbers in an array and then search for the element in the array. Display appropriate messages if found or not found.

2. Write a program to search an element in an array.
3. Enter ten numbers in an array. Write a program to find out the number of positive, negative, even and odd numbers.
4. Enter ten numbers in the array and find the sum of all alternate numbers.
5. Write a program to display duplicate numbers in an array.
6. Write a program to delete duplicate numbers in an array.
7. Write a program to find the largest & the second largest in the list entered by the user.
8. Write a program to input data in an array carrying 25 elements. Compute the sum and average. Then count the number of elements entered greater than the average and number of elements less than the average.

Strings:

A string is a collection of characters enclosed within quotes. In C/C++ a string is defined as a character array terminated by NULL character '\0'. Each element of a string is used as one element in the array housing it. So the character arrays must be declared one character longer than the size of the string we wish to store. The last bytes store the string terminator '\0'.

There is no data type available as string so we have to declare it as an array of characters.

Implementing of string functions:

```
#include<iostream.h>
#include<string.h>
main()
{
    char x[20],y[10];
    cout<<"\nEnter the string x :";
    cin>>x;
    cout<<"\nEnter the string y :";
    cin>>y;
    int a =strlen(x);
    cout<<"\nThe string is :"<<x;
    cout<<"\nThe length of an array :"<<a;
    strcat(x,y);
    cout<<"\nThe concatenated string is :"<<x;
    strcpy(x,y);
    cout<<"\nThe y string copied to x gives :"<<x;
    a=strcmp(x,y);
    if(a==0)
        cout<<"\nStrings are x and y equal";
    else
        cout<<"\nStrings x and y are not equal";
}
/*Enter the string x :shalini
```

```
Enter the string y :puri
The string is :shalini
The length of an array :7
The concatenated string is :shalinipuri
The y string copied to x gives :puri
Strings are x and y equal*/
```

Finding the length of string without using predefined functions:

```
#include<iostream.h>
main()
{
    char x[10];
    int i;
    cout<<"\nEnter the string :";
    cin>>x;
    cout<<"\nThe string is :";
    for(i=0;x[i]!='\0';i++)
        cout<<x[i];
    cout<<"\nLength of string is :"<<i;
}
/*
Enter the string :shalini
The string is :shalini
Length of string is :7*/
```

Concatenating two strings without using string functions:

```
#include<iostream.h>
main()
{
    char x[20],y[10];
    int i,j,k;
    cout<<"\nEnter the string :";
    cin>>x;
    cout<<"\nEnter the string :";
    cin>>y;
    for(i=0;x[i]!='\0';i++);
    for(j=i,k=0;y[k]!='\0';k++,j++)
        x[j]=y[k];
    cout<<"\nThe concatenated string is :"<<x;
}
/*
Enter the string :shalini
Enter the string :puri
The concatenated string is :shalinipuri*/
```

Exercise:

1. Write a program to display and count no. of vowels in a given string.
2. Accept the string and calculate length without using string functions.

3. Write a program to reverse a string without using predefined string functions.
4. Write a program to sort the array in alphabetical order.
5. Write a program to eliminate blank spaces from a line.
6. Write a program to calculate the frequency of a character in a given string.
7. Write a program with functions that deletes a character from the string and adds a character in the string at a required position.
8. Write a program to find string s2 in string s1 where s2 is substring & s1 is main string.

Two dimensional arrays:

We have seen one dimensional array up till now but it is also possible to have arrays with more than one dimension. The two dimensional arrays is called a matrix. So let us write a sample program that stores a roll number and marks obtained by student side by side in a matrix.

```
#include<iostream.h>
main()
{
    int stud[4][2];
    int i;
    for(i=0;i<4;i++)
    {
        cout<<"\nEnter Roll NO and marks :";
        cin>>stud[i][0]>>stud[i][1];
    }
    for(i=0;i<4;i++)
        cout<<"\nRoll No :"<<stud[i][0]<<"  Marks :"<<stud[i][1];
}
/*
Enter Roll NO and marks :1 45
Enter Roll NO and marks :2 56
Enter Roll NO and marks :3 67
Enter Roll NO and marks :4 78
Roll No :1  Marks :45
Roll No :2  Marks :56
Roll No :3  Marks :67
Roll No :4  Marks :78*/
```

1. **Write a program to accept a two dimensional array and display the :**
    a) **Reverse order of rows**
    b) **Reverse order of columns**
    c) **Reverse of order and columns**
    d) **Matrix along with row and column sum.**

```
#include<iostream.h>
main()
```

```
{
    int x[4][4];
    int i,j,sum;
    int sc[]={0,0,0,0};
    for(i=0;i<4;i++)
    {
            cout<<"\nEnter the value of "<<i<<"th row";
            for(j=0;j<4;j++)
            {
                    cin>>x[i][j];
            }
    }
    cout<<"\n The x array is :"<<endl;
    for(i=0;i<4;i++)
    {
            for(j=0;j<4;j++)
            {
                    cout<<x[i][j]<<" ";
            }
            cout<<endl;
    }
    cout<<"\nReverse order of rows :"<<endl;
     for(i=3;i>=0;i--)
    {
            for(j=0;j<4;j++)
            {
                    cout<<x[i][j]<<" ";
            }
            cout<<endl;
    }
    cout<<"\nReverse order of columns :"<<endl;
     for(i=0;i<4;i++)
    {
            for(j=3;j>=0;j--)
            {       cout<<x[i][j]<<" ";
                    }
                    cout<<endl;             }
    cout<<"\nReverse order of rows and columns :"<<endl;
     for(i=3;i>=0;i--)
    {
            for(j=3;j>=0;j--)
            {       cout<<x[i][j]<<" ";
            }
            cout<<endl;
    }
    cout<<"\nSum of rows and columns :"<<endl;
    for(i=0;i<4;i++)
    {       sum=0;
            for(j=0;j<4;j++)
            {
```

```
                        cout<<x[i][j]<<" ";
                        sum=sum+x[i][j];
                        sc[i]=sc[i]+x[j][i];
                }
                cout<<"="<<sum<<endl;
        }
        cout<<"============="<<endl;
        for(i=0;i<4;i++)
                cout<<sc[i]<<" ";
    }
    /*Enter the value of 0th row1 2 3 4
    Enter the value of 1th row5 6 7 8
    Enter the value of 2th row9 10 11 12
    Enter the value of 3th row13 14 15 16
    The x array is :
    1 2 3 4
    5 6 7 8
    9 10 11 12
    13 14 15 16
    Reverse order of rows :
    13 14 15 16
    9 10 11 12
    5 6 7 8
    1 2 3 4
    Reverse order of columns :
    4 3 2 1
    8 7 6 5
    12 11 10 9
    16 15 14 13
    Reverse order of rows and columns :
    16 15 14 13
    12 11 10 9
    8 7 6 5
    4 3 2 1
    Sum of rows and columns :
    1 2 3 4 =10
    5 6 7 8 =26
    9 10 11 12 =42
    13 14 15 16 =58
    =============
    28 32 36 40*/
```

2. **Write a program to perform the matrix multiplication of two matrices.**

```
#include<iostream.h>
main()
{
    int x[4][4],y[4][4],z[4][4];
    int i,j,k;
    for(i=0;i<4;i++)
    {
        cout<<"\nEnter the value of "<<i<<"th row";
```

```
            for(j=0;j<4;j++)
            {
                    cin>>x[i][j];
            }
        }
        cout<<"\n The x array is :";
        for(i=0;i<4;i++)
        {
            for(j=0;j<4;j++)
            {
                    y[i][j]=x[i][j];
                    cout<<x[i][j]<<" ";
            }
            cout<<endl;
        }
        cout<<"\nThe y array is :";
         for(i=0;i<4;i++)
        {
            for(j=0;j<4;j++)
            {       cout<<y[i][j]<<" ";
            }
            cout<<endl;
        }
        cout<<"\nThe Z array is :";
         for(i=0;i<4;i++)
        {
            for(j=0;j<4;j++)
            {
                    z[i][j]=0;
                    for(k=0;k<4;k++)
                    {
                            z[i][j]=z[i][j]+(x[i][k]*y[k][j]);
                    }
            }
        }
         for(i=0;i<4;i++)
        {
            for(j=0;j<4;j++)
            {       cout<<z[i][j]<<" ";
            }
            cout<<endl;
        }
    }
    /*Enter the value of 0th row1 2 3 4
    Enter the value of 1th row5 6 7 8
    Enter the value of 2th row9 10 11 12
    Enter the value of 3th row13 14 15 16
    The x array is :
    1 2 3 4
    5 6 7 8
```

```
9 10 11 12
13 14 15 16
The y array is :
1 2 3 4
5 6 7 8
9 10 11 12
13 14 15 16
The Z array is :
90 100 110 120
202 228 254 280
314 356 398 440
426 484 542 600*/
```

3. Write a program to print the square of two matrices

```
#include<iostream.h>
main()
{
    int x[4][4],y[4][4];
    int i,j;
    for(i=0;i<4;i++)
    {
        cout<<"\nEnter the value of "<<i<<"th row";
        for(j=0;j<4;j++)
        {       cin>>x[i][j];
        }
    }
    cout<<"\n The x array is :"<<endl;
    for(i=0;i<4;i++)
    {
        for(j=0;j<4;j++)
        {       cout<<x[i][j]<<" ";
        }
        cout<<endl;
    }
    cout<<"\nSquare of the matrix :"<<endl;
    for(i=0;i<4;i++)
    {
        for(j=0;j<4;j++)
        {
                y[i][j]=x[i][j]*x[i][j];
                cout<<y[i][j]<<" ";
        }
        cout<<endl;
    }
}
/*Enter the value of 0th row1 2 3 4
Enter the value of 1th row5 6 7 8
Enter the value of 2th row9 10 11 12
Enter the value of 3th row13 14 15 16
The x array is :
```

```
1 2 3 4
5 6 7 8
9 10 11 12
13 14 15 16
Square of the matrix :
1 4 9 16
25 36 49 64
81 100 121 144
169 196 225 256*/
```

Exercise:

1. Four experiments are performed where each experiment consisting of six test results. The result of each experiment is as follows. Write a program to compute and display the average of the test results for each experiment.

1st Experiment results	23.2	31.5	16.9	28.0	26.3	28.2
2nd Experiment results	34.8	45.2	20.8	39.4	33.4	36.8
3rd Experiment results	19.4	50.6	45.1	20.8	50.6	13.4
4th Experiment results	36.9	42.7	20.8	10.2	16.8	42.3

2. Write a program that declares three dimensional arrays named price, quantity and amount. Each array should be declared in main ( ) and should be capable of holding 10 double precision numbers. The numbers that should be stored in price 10.20,11,30,13.14,16.9,18.1,2.71,7.55,15.12,9.45,17.0. The numbers that should be stored in quantity are 3,9.7,6.40,4.5,5.6,6.2,7,2.8,15.0,18.0. Your program should pass these three arrays to a function name extend ( ) which should calculate the elements in the amount array as the product of the corresponding elements in the price and quantity arrays after extend ( ) has put values into the amount array the value should be displayed in the function.
3. Write a program which contains the function to do the following:
   i) To check whether the matrix is symmetric or not. A matrix is symmetric if its transpose equals the matrix itself.
   ii) To find the sum of elements lying above the main diagonal.
4. Write a menu driven program to perform the following operations on matrices.
   i) Addition ii) Subtraction iii) Transpose iv) Multiplication

Chapter 4.1: Pointers

Let us consider the notation:

int i =4;

This declaration tells the compiler to:

1. Reserve space in memory to hold the integer values
2. Associate the name 'i' with the memory location
3. Store the value 4 at this location

In the above figure we can see the variable 'i' is stored in memory location 3001.

Variable	Memory	Address
i	4	3001

References:

A reference is an alias or synonym for another value. It is declared by the syntax:

Type &rc_name =var_name;

where type is the variable's type, ref_name is the name of the reference and var_name is the name of the variable. For example in the declaration:

int &rn = n //rn is a synonym for n.

rn is declared to be a reference to the variable n which must be already have been declared.

```
#include<iostream.h>
main()
{
        int n=44;
        int &rn=n;
        cout<<"\nn="<<n<<" ,rn ="<<rn<<endl;
        --n;
        cout<<"\nn="<<n<<" ,rn ="<<rn<<endl;
        rn*=2;
        cout<<"\nn="<<n<<" ,rn ="<<rn<<endl;
}
/*n=44 ,rn =44
n=43 ,rn =43
n=86 ,rn =86*/
```

So we say n and m are different names for the same variables. Decrementing n changes rn also and multiplying rn value by 2 creates a change in n also.

Pointers:

As the name suggests the pointers are used to point to something. We have studied so far variables of type int, char, array etc. they hold the value of their type. Pointer doesn't hold any is a named memory location so pointers actually point to memory locations.

We declare using (*value at address) notation.

Consider the following notion:

int *p;

This statement indicates that 'p' is a pointer of type integer. It means the value at address contained in p as int.

This declaration tells the C compiler to reserve memory for pointer variable 'p' when we execute the following statement

p = &i;

address of I is put as value for p. Pictorially it can be expressed as shown in figure below:

Variable	Memory	Address
i	4	3001
p	3001	5001

The new operator:

When a pointer is declared like this:

float *p;

It allocates memory for the pointer itself. The value of the pointer will be some memory address but the memory is yet not allocated. This means that storage could already be in use by some other variable. In this case p is not initialized which means it is not pointing to any allocated memory. Any attempt to access the memory to which it points will be an error.

```
p = 3.14 //error
```

A good way to avoid this problem is to initialize pointers when they are declared.

```
float x =3.14;
float *p =&x;
cout<< *p;
```

In this case accessing *p is no problem because the memory needed to store the float 3.14 was automatically allocated when 'x' was declares 'p' points to the same allocated memory.

Another way to avoid the problem of declaring pointer is to allocate memory explicitly for pointer itself for pointer itself. This is done with new operator.

```
float *q;
q = new float;
*q=3.14;
```

The new operator returns the address of a block of ‘s’ unallocated bytes in memory where ‘s’ is the size of float. Assigning the address of ‘q’ guarantees that *q is not currently in use by any other variables.

We can also combine these lines and write:

```
float *q = new float(3.14);
```

The new operator will return zero if there is no enough memory available to allocate a block of required size.

The delete operator:

The delete operator reverses action of the new operator returning allocated memory to the free store. It should be applied to pointers that have been explicitly allocated by new operator.

```
float *q = new float(3.14);
delete q;
*q = 2.17 //error
```

Pointer to array:

An array is an ordered collection of consecutive memory locations or data objects. Here the consecutive memory location means locations with consecutive memory addresses. Suppose x is an array of 10 elements then the addresses of the element x[1] can be obtained from x[0] by adding 1 to it. The address of the elements of an array is ordered in incremental increasing order

```
#include<iostream.h>
main()
{
        int arr[]={11,-20,30,41,15};
        int *ptr;
        ptr=&arr[0];
        cout<<"\nThe array elements are :"<<endl;
        while(ptr<ptr+5)
        {
                cout<<*ptr<<" ";
                ptr++;
        }
}
/*The array elements are :
11 -20 30 41 15*/
```

So the above program the pointer points to the first element of an array initially. When it is incremented it points to the next element in the array. This is called as the pointer to an array.

Call by value, Call by pointer and Call by reference:

```
#include<iostream.h>
void swap(int,int);
void swap1(int&,int&);
void swap(int*,int*);
main()
{
	int a,b,c,d;
	cout<<"\nEnter the value :";
	cin>>a>>b;
	c=a;d=b;
	cout<<"\n Values before calling call by value :"<<endl;
	cout<<a<<" "<<b;
	swap(a,b);
	cout<<"\n Values after calling call by value :"<<endl;
	cout<<a<<" "<<b;
	cout<<"\n Values before calling call by pointer :"<<endl;
	cout<<a<<" "<<b;
	swap(&a,&b);
	cout<<"\n Values after calling call by pointer :"<<endl;
	cout<<a<<" "<<b;
	cout<<"\n Values before calling call by reference :"<<endl;
	cout<<c<<" "<<d;
	swap1(c,d);
	cout<<"\n Values after calling call by reference :"<<endl;
	cout<<c<<" "<<d;
}
void swap(int x,int y)
{
	int temp;
	temp=x;
	x=y;
	y=temp;
}
void swap(int *x,int *y)
{
	int temp;
	temp=*x;
	*x=*y;
	*y=temp;
}
void swap1(int &x,int &y)
{
	int temp;
	temp=x;
	x=y;
	y=temp;
}
/*
Enter the value :45 67

 Values before calling call by value :
45 67
```

```
 Values after calling call by value :
45 67
 Values before calling call by pointer :
45 67
 Values after calling call by pointer :
67 45
 Values before calling call by reference :
45 67
 Values after calling call by reference :
67 45*/
```

Questions:

1. Write a program to illustrate parameter passing technique to add two numbers using call by vale and call by address.
2. Explain the following functions:

   i) malloc ( )   ii) calloc( )   iii) free( )
3. The below program contains the following statements:]

   ```
   float  a =0.001, b= 0.003;
   float c, *pa, *pb;
   pa = &a;
   *pa = 2*a;
   Pb=&b;
   c=3*(*pb - *pa);
   ```
   The address of a as 1130, b= 1134 and c=1138 then what values are assigned to-
   i) &a   ii) *pa   iii) c   iv) pb   v) *pa   vi) &c

4. A program contains the following declarations

   int a[5] = {10,40,60,75,85}

   Assume the starting memory address is 65535. What is the meaning of the following?

   i) sizeof(A) ii) A iii)*A iv) *(A+2) v) *A+2 vi) A+2

Chapter 4.2: Structures and Unions:

We can create and use the data types other than the fundamental data types. These are known as user defined data types. Data types using the keyword **struct** are known as structures. As seen earlier arrays have similar data type elements. A structure is a collection of mixed data types referenced by a single name. it is a group of related data items i.e. structural elements of arbitrary types. The general structure of declaring the syntax is:

```
struct <name>
{
        <type > <member1>;
        <type> <member2>;
        ………………..
        <type> <memberN>;
} <structural variables>;
```

Where <name > is the name of the structure i.e. name of the new data type. The keyword **struct** and <name> are used to declare structured variables.

For example:

```
struct employee
{
        int empno;
        char name[20];
        char designation[20];
        char dept[20];
} emp;
```

The structure variable can be declared as:

```
struct employee
{
        int empno;
        char name[20];
        char designation[20];
        char dept[20];
};
struct employee emp;
```

Accessing structure elements:

Structure use a dot(.) operator the below program illustrate the use of structure

```
#include<iostream.h>
struct employee
{       int empno;
        char name[20];
        float salary;
};
main()
{
        struct employee emp;
        cout<<"\nEnter the data :";
        cout<<"\nEnter the employee no :";
        cin>>emp.empno;
        cout<<"\nEnter the name :";
        cin>>emp.name;
        cout<<"\nEnter the salary :";
        cin>>emp.salary;
        cout<<"\nDisplay the details :";
        cout<<"\nEmployee Number : "<<emp.empno;
        cout<<"\nEmployee Name   : "<<emp.name;
        cout<<"\nEmployee Salary : "<<emp.salary;
}
/*Enter the data :
Enter the employee no :1
Enter the name :shalini
Enter the salary :14000
Display the details :
Employee Number : 1
Employee Name   : shalini
Employee Salary : 14000 */
```

Structure within a structure:

The following program illustrates the use of structure within a structure with the help of above example:

```
#include<iostream.h>
struct date
{
        int dd,mm,yy;
};
struct employee
{
        int empno;
        char name[20];
        float salary;
        struct date d;
};
main()
{
        struct employee emp;
        clrscr();
        cout<<"\nEnter the data for employee :"<<endl;
        cout<<"\nEnter the employee no :";
        cin>>emp.empno;
        cout<<"\nEnter the name :";
        cin>>emp.name;
        cout<<"\nEnter the salary :";
        cin>>emp.salary;
        cout<<"\nEnter Date of Birth (dd/mm/yy) :";
        cin>>emp.d.dd>>emp.d.mm>>emp.d.yy;
        cout<<"\nDisplay the details :";
        cout<<"\nEmployee Number        : "<<emp.empno;
        cout<<"\nEmployee Name          : "<<emp.name;
        cout<<"\nEmployee Salary        : "<<emp.salary;
        cout<<"\nEmployee Date of birth : "<<emp.d.dd<<"/"<<emp.d.mm<<"/"<<emp.d.yy;
        getch();
}
/*Enter the data for employee :
Enter the employee no :101
Enter the name :abc
Enter the salary :2000
Enter Date of Birth (dd/mm/yy) :01 03 1977
Display the details :
Employee Number        : 101
Employee Name          : abc
Employee Salary        : 2000
Employee Date of birth : 1/3/1977*/
```

Array of structures:

```
#include<iostream.h>
struct employee
{
        int empno;
        char name[20];
        float salary;
};
main()
{
        struct employee emp[5];
        cout<<"\nEnter the data for 3 employees :"<<endl;
        for(int i=0;i<3;i++)
        {
                cout<<"\nEnter the employee no :";
                cin>>emp[i].empno;
                cout<<"\nEnter the name :";
                cin>>emp[i].name;
                cout<<"\nEnter the salary :";
                cin>>emp[i].salary;
        }
        cout<<"\nDisplay the details :";
        for(i=0;i<3;i++)
        {
                cout<<"\nEmployee Number : "<<emp[i].empno;
                cout<<"\nEmployee Name   : "<<emp[i].name;
                cout<<"\nEmployee Salary : "<<emp[i].salary;
        }
}
```

```
/*Enter the data for 3 employees :
Enter the employee no :1
Enter the name :shalini
Enter the salary :3000
Enter the employee no :2
Enter the name :ajay
Enter the salary :5000
Enter the employee no :3
Enter the name :abhi
Enter the salary :6000
Display the details :
Employee Number : 1
Employee Name   : shalini
Employee Salary : 3000
Employee Number : 2
Employee Name   : ajay
Employee Salary : 5000
Employee Number : 3
Employee Name   : abhi
Employee Salary : 6000*/
```

Pointer to a structure:

```
#include<iostream.h>
struct employee
{
        int empno;
        char name[20];
        float salary;
};
main()
{
        struct employee *emp;
        cout<<"\nEnter the data for 3 employees :"<<endl;
        for(int i=0;i<3;i++)
        {
                cout<<"\nEnter the employee no :";
                cin>>emp->empno;
                cout<<"\nEnter the name :";
                cin>>emp->name;
                cout<<"\nEnter the salary :";
                cin>>emp->salary;
                emp++;
        }
        emp=emp-3;
        cout<<"\nDisplay the details :";
        for(i=0;i<3;i++)
        {
                cout<<"\nEmployee Number : "<<emp->empno;
                cout<<"\nEmployee Name   : "<<emp->name;
                cout<<"\nEmployee Salary : "<<emp->salary;
                emp++;
        }
}
/*Enter the data for 3 employees :
Enter the employee no :1
Enter the name :shalini
Enter the salary :2000
Enter the employee no :2
Enter the name :ajay
Enter the salary :5000
Enter the employee no :3
Enter the name :amar
Enter the salary :7000
Display the details :
Employee Number : 1
Employee Name   : shalini
Employee Salary : 2000
Employee Number : 2
Employee Name   : ajay
Employee Salary : 5000
Employee Number : 3
Employee Name   : amar
Employee Salary : 7000*/
```

Chapter 4.3: Create a singly linked list:

```
#include<iostream.h>
#include<malloc.h>
#include<stdlib.h>
void Delete();
void create();
void display();
void ADDB(int);
void ADDE(int,int);
struct List
{
        int data;
        struct List *next;
}*head;
main()
{
        head = NULL;
        int choice,element,pos;
        do
        {
                cout<<"\n\t\tMAIN MENU";
                cout<<"\n\t\t1. CREATE";
                cout<<"\n\t\t2. ADD AT BEGINING";
                cout<<"\n\t\t3. ADD AFTER A POSITION";
                cout<<"\n\t\t4. DELETE";
                cout<<"\n\t\t5. EXIT";
                cout<<"\n\tEnter the choice : (1,2,3,4,5) ";
                cin>>choice;
                switch(choice)
                {
                        case 1 : create();
                                        break;
                        case 2 : cout<<"\nEnter the element :";
                                        cin>>element;
                                        ADDB(element);
                                        break;
                        case 3 : cout<<"\nEnter element & position :";
                                        cin>>element>>pos;
                                        ADDE(element,pos);
                                        break;
                        case 4:  Delete();
                                        break;
                        case 5: display();
                                    break;
                        case 6:exit(0);
                }
        }
        while(choice!=6);
}
void create()
{
        struct List *start, *q;
        start=(struct List *)malloc(sizeof(struct List));
        cout<<"\nEnter the data : ";
        cin>>start->data;
        start->next=NULL;
        if(head==NULL)
                head=start;
        else
        {
                q=head;
                while(q->next!=NULL)
                {
                        q=q->next;
                }
                q->next=start;
        }
}
void ADDB(int x)
{
        struct List *q;
        q=(struct List *)malloc(sizeof(struct List));
        q->data=x;
        q->next=NULL;
```

```
		q=head;
	}
	void ADDE(int element,int pos)
	{
		struct List *q,*temp;
		q=head;
		for(int i=0;i<pos-1;i++)
		{
			q=q->next;
			if(q==NULL)
			{
				cout<<"\nThere are less elements then the"<<pos<<"element";
				return;
			}
		}
		temp=(struct List *)malloc(sizeof(struct List));
		temp->next=q->next;
		temp->data=element;
		q->next=temp;
	}
	void Delete()
	{
		struct List *q,*start;
		int found=0,num;
		q=(struct List *)malloc(sizeof(struct List));
		start=head;
		if(start==NULL)
			cout<<"\nThe list is empty";
		else
		{
			cout<<"\nEnter the number to be deleted :";
			cin>>num;
			while((found==0)&&(start!=NULL))
			{
				if(start->data==num)
					found=1;
				else
					start=start->next;
			}
		}
		if(found==0)
		{
			cout<<"\nNo such numebr in the list";
			getch();
		}
		else
		{
			if(start==head)
			{
				head=head->next;
```

```
                        start->next=NULL;
                        free(start);
                }
                else
                {
                        q=head;
                        while(q->next!=start)
                                q=q->next;
                        q->next=start->next;
                        start->next=NULL;
                        free(start);
                }
                cout<<"\nElement deleted";
        }
}
void display()
{
        struct List *p;
        p=head;
        while(p!=NULL)
        {
                cout<<p->data;
                p=p->next;
        }
}
/*              MAIN MENU
            1. CREATE
            2. ADD AT BEGINING
            3. ADD AFTER A POSITION
            4. DELETE
            5. DISPLAY
            6. EXIT
      Enter the choice : (1,2,3,4,5,6) 1
Enter the data : 1
            MAIN MENU
            1. CREATE
            2. ADD AT BEGINING
            3. ADD AFTER A POSITION
            4. DELETE
            5. DISPLAY
            6. EXIT
      Enter the choice : (1,2,3,4,5,6) 1
Enter the data : 2
            MAIN MENU
            1. CREATE
            2. ADD AT BEGINING
            3. ADD AFTER A POSITION
            4. DELETE
            5. DISPLAY
                  6. EXIT
      Enter the choice : (1,2,3,4,5,6) 1
Enter the data : 3
            MAIN MENU
            1. CREATE
            2. ADD AT BEGINING
            3. ADD AFTER A POSITION
            4. DELETE
            5. DISPLAY
            6. EXIT
      Enter the choice : (1,2,3,4,5,6) 1
Enter the data : 5

            MAIN MENU
            1. CREATE
            2. ADD AT BEGINING
            3. ADD AFTER A POSITION
            4. DELETE
            5. DISPLAY
            6. EXIT
      Enter the choice : (1,2,3,4,5,6) 3
Enter element & position :4 3
            MAIN MENU
            1. CREATE
            2. ADD AT BEGINING
            3. ADD AFTER A POSITION
            4. DELETE
            5. DISPLAY
            6. EXIT
      Enter the choice : (1,2,3,4,5,6) 5
12345
            MAIN MENU
            1. CREATE
            2. ADD AT BEGINING
            3. ADD AFTER A POSITION
            4. DELETE
            5. DISPLAY
            6. EXIT
      Enter the choice : (1,2,3,4,5,6) 2
Enter the element :0
            MAIN MENU
            1. CREATE
            2. ADD AT BEGINING
            3. ADD AFTER A POSITION
            4. DELETE
            5. DISPLAY
            6. EXIT
      Enter the choice : (1,2,3,4,5,6) 4
Enter the number to be deleted :3
Element deleted
            MAIN MENU
```

```
          1. CREATE
          2. ADD AT BEGINING
          3. ADD AFTER A POSITION
          4. DELETE
          5. DISPLAY
          6. EXIT
     Enter the choice : (1,2,3,4,5,6) 5
1245
          MAIN MENU
          1. CREATE
          2. ADD AT BEGINING
          3. ADD AFTER A POSITION
          4. DELETE
          5. DISPLAY
          6. EXIT
     Enter the choice : (1,2,3,4,5,6)6*/
```

Questions:

1. What is union? How does it differ from a structure? Explain with suitable example how is a union member accessed? How can a union member be processed? When is a union required in programming?
2. Define a structure called cricket that will describe the following information-
   i) Player name ii) Country Name iii) No. of matches played iv) Batting average
   Develop a program that will store information of 50 cricket players around the world using structure. Also display names of players in descending order of batting average.
3. Define a structure 'st' to contain the name, date of birth and total marks obtained. Define the structure 'dob' to represent date of birth. WAP to read data for n students in a class and sort them in descending order of batting average.
4. State the difference between structure and union?

Object Oriented Programming with features and benefits:

The major motivating factor for developing an object oriented approach is to remove the flaws encountered in the procedural approach. OOP treats as the critical element in program development and does not allow it to flow freely around the system. It ties data more closely to functions that operate on it and protect it from accidental modification from outside functions. OOP allows us to decompose a problem into number of entities called objects and then builds data and functions around those entities.

1. Objects:

Objects are basic run time entities in an object oriented system. They represent a person, a place, a bank account, a table of data or any item that the program has to handle. The programs can be easily designed in terms of objects and there is a close match between the objects in real word which posses a combination of data and functions.

2. Classes:

A class is considered as a user defined data type which contains the attributes and functions of a particular object. Using this user defined data type we will be able to create several objects of that class. So a class serves as a plan or a template that contains data and functions. A class is called as a collection of similar objects.

3. Data Hiding and Encapsulation:

The wrapping up of data and functions into a single unit (class) is known as Encapsulation. Data encapsulation is the most striking feature of a class. The data is not accessible to the outside world and only those functions which are wrapped in the class can access it. These functions provide the interface between the object's data and the program. This process of insulating the data from any other part of code which is outside the object is called data hiding or information hiding.

4. Data Abstraction:

Abstraction refers to the act of representing essential features without including the background details or explanations. Classes use the concepts of abstraction and are defined as a list of abstract attributes such as name, age, cost and weight together with functions that operate on these attributes. They encapsulate all essential properties of the objects that are to be created. The attributes are sometimes called data members because they hold information. The functions that operate on these data are called methods and member functions.

5. **Inheritance:**

Classes often have similar characteristics so by taking advantage of such object relationships we can significantly reduce the amount of code. This means that we should avoid duplication of elements within each class. All the elements and function are put under a base class. After this we declare derived classes which inherit the characteristics of the base class. So we define inheritance as the process by which objects of one class acquire the properties of objects of another class.

6. Polymorphism:

Polymorphism means the ability to take more than one form. An operation may exhibit different instances. The behavior depends upon the type of the data being used in operation. For example consider the operation of addition. For two numbers the operation will generate a sum. If the operands are strings then the operation will produce a third string by concatenation. The process of making an operator to exhibit different behaviors in different instances is known as **operator overloading**.

Using a single function name to perform different types of tasks is called **function overloading**.

The figure illustrates that a single function name can be used to handle different number and different types of arguments.

7. Dynamic Binding:

Binding refers to the linking of a procedure call to the code to be executed in response to the call. Dynamic binding/late binding means that the code associated with a given procedure call is not known until the time of the call at runtime. It is associated with polymorphism.

Consider the procedure draw ( ) in the above figure. By inheritance every object will have this procedure. Its algorithm is however unique to each object and so the draw procedure will be redefined in each class that defines the object. At run time the code matching of the object is done and the current reference object will be called.

8. Message Passing:

An object oriented program consist a set of objects that communicate with each other. The process of programming in an object oriented language involves following basic steps:

a) Create a class that defines objects and their behaviors.
b) Creating objects from class definitions
c) Establishing communication among objects

Objects communicate with one another by sending and receiving information much the same way as people pass messages to one another. The concept of message passing makes it easier to talk about building systems that directly model or simulate the real world counterparts.

A message for an object is a request for execution of a procedure and therefore will invoke a function will invoke a function (procedure) in the receiving object that generates the desired result. Objects have a life cycle. They can be created and destroyed. Communication with an object is feasible as long as its function (message) and the information to be sent.

$$\begin{tikzcd} & \text{Staff.salary(name);} \arrow[dl] \arrow[d] \arrow[dr] & \\ \text{Object} & \text{Message} & \text{Information} \end{tikzcd}$$

Class:

A class is a way to bind the data and is associated functions together. It allows data and function to be hidden if necessary from external use. When defining a class we are creating a new abstract data type that can be treated like any other built in data type. Generally classification has two parts:

1. Class definition
2. Class function declaration

The class declaration describes the type and scope of its members. The class function definition describes how the class functions are implemented.

General form of a class declaration is:

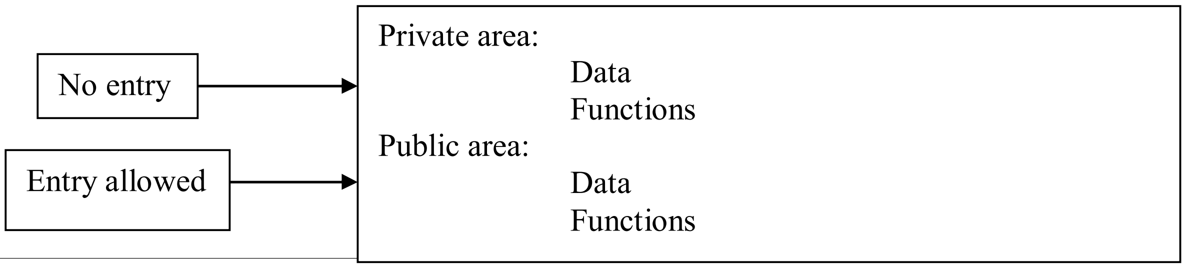
```
class class_name
{
    private:
   variable declarations;
    public:
   variable declarations;
   function declarations;
};
```

The class declaration is similar to a **struct** declaration. The keyword class specifies that what follows is an abstract data of type class_name. The class body contains the declarations of variables and functions.

Accessing specifiers / Visibility mode:

The function and variables are collectively called class members. They are usually two sections namely private and public to denote. The keywords private and public are known as visibility labels.

The class members that have been declared as private can be accessed only from the class. On the other hand public members can be accessed from outside the class also. The data hiding is the key feature of object oriented programming. The use of keyword private is optional. By default the members of a class are private. If both labels are missing then by default all members are private. Such a class completely hidden from the outside world and does not serve any purpose. The variable declared inside the class are known as data members and the functions as member functions. Only the member functions can have access to the private data members and private functions. However the public members are accessed from outside the class. The binding of data & functions together into a single class type variable is called **encapsulation.**



A simple class example:

```
#include<iostream.h>
class Employee
{   int empno;
    char name[20];
    float salary;
    public:
    void getdata()
    {
    cout<<"\nEnter the data for employee :"<<endl;
    cout<<"\nEnter the employee no :";
    cin>>empno;
    cout<<"\nEnter the name :";
    cin>>name;
    cout<<"\nEnter the salary :";
    cin>>salary;
    }
    void putdata()
    {
    cout<<"\nDisplay the details :";
    cout<<"\nEmployee Number       : "<<empno;
    cout<<"\nEmployee Name         : "<<name;
    cout<<"\nEmployee Salary       : "<<salary;
    }
};
main()
{   Employee E;
    E.getdata();
    E.putdata();
}
```

```
/*Enter the data for employee :
Enter the employee no :1
Enter the name :shalini
Enter the salary :3000
Display the details :
Employee Number        : 1
Employee Name          : shalini
Employee Salary        : 3000*/
```

Array of objects:

Array can be a data member of the objects that you define for the class. We can in the similar way have array of objects. For example say that we have an employee as a class declaration and we want to define an array of objects of type **employee** containing 3 elements. Then we declare as: employee array_name [3];

```
#include<iostream.h>
class Employee
{  private:
   int empno;
   char name[20];
   float salary;
   public:
   void getdata()
   {
        cout<<"\nEnter the data for employee :"<<endl;
        cout<<"\nEnter the employee no :";
        cin>>empno;
        cout<<"\nEnter the name :";
        cin>>name;
        cout<<"\nEnter the salary :";
        cin>>salary;
   }
   void putdata()
   {
        cout<<"\nDisplay the details :";
        cout<<"\nEmployee Number        : "<<empno;
        cout<<"\nEmployee Name          : "<<name;
        cout<<"\nEmployee Salary        : "<<salary;
   }
};
main()
{  Employee E[5];
   for(int i=0;i<3;i++)
        E[i].getdata();
   for(i=0;i<3;i++)
        E[i].putdata();
   }
/*Enter the data for employee :
Enter the employee no :1
Enter the name :shalini
Enter the salary :4000
Enter the data for employee :
Enter the employee no :2
Enter the name :abhi
Enter the salary :5000
Enter the data for employee :
Enter the employee no :3
Enter the name :amar
Enter the salary :7000
Display the details :
Employee Number       : 1
Employee Name         : shalini
Employee Salary       : 4000
Display the details :
Employee Number       : 2
Employee Name         : abhi
Employee Salary       : 5000
Display the details :
Employee Number       : 3
Employee Name         : amar
Employee Salary       : 7000*/
```

Objects as function arguments:

Class is a user defined data type and object is an instance of a class. So like any other data type an object may be as a function argument. This can be done in two ways:

1. Call by value:

A copy of the entire object is passed to the function so any changes made to the object inside the functions do not affect the object used to call the function.

2. Call by reference:

The address of the object is passed to the function so the called function works directly on the actual object used in the call. This means that any changes made to the object inside the function will reflect in the actual object.

The following program illustrates the use of objects as function parameters. It performs the addition of two rational numbers and store the result in the third rational number. Objects are passed by value. The class Rational is defined with two data members numerator and denominator as every rational number is represented as x/y where x and y are integers.

```
#include<iostream.h>
class Rational
{  int num,den;
   public:
   void set(int x,int y)
   {
   num=x;den=y;
   }
   void print()
   {
   cout<<num<<"/"<<den<<endl;
   }
   void sum(Rational A,Rational B)
   {
   Rational temp;
   temp.den=A.den*B.den;
   temp.num=A.num*B.den+B.num*A.den;
   cout<<temp.num<<"/"<<temp.den;
   }
};
main()
{   Rational X,Y;
   X.set(2,7);
   Y.set(3,8);
   X.print();
   Y.print();
   X.sum(X,Y);
   getch();
}
/*2/7
3/8
37/56*/
```

Function returning objects:

At times it is necessary to return an object from the function to main program. Following program illustrates the use of functions returning objects as value.

```
#include<iostream.h>
class Rational
{   int num,den;
    public:
    void set(int x,int y)
    {
    num=x;den=y;
    }
    void print()
    {
    cout<<num<<"/"<<den<<endl;
    }
    void sum(Rational A,Rational B)
    {
    Rational temp;
    temp.den=A.den*B.den;
    temp.num=A.num*B.den+B.num*A.den;
    cout<<temp.num<<"/"<<temp.den;
    }
};
main()
{  Rational X,Y;
   X.set(2,7);
   Y.set(3,8);
   X.print();
   Y.print();
   X.sum(X,Y);

}
/*2/7
3/8
37/56*/
```

Constructors:

A constructor is a special public member function whose task is to initialize the object data members when an object is declared. The constructor of a class should have the same name as the class. For example if the name of class is ***Employee*** then the constructor is a function with the name ***Employee( ).***

Characteristics of a constructor function are:

1. They should be declared in the public section.
2. They automatically get invoked when the objects are created
3. They do not have return types not even void and therefore they cannot return values.
4. They cannot be inherited create through a derived class constructor can call the base class constructor.
5. Like other functions they can have default arguments.

The below example shows a default constructor:

```
#include<iostream.h>
#include<string.h>
class Employee
{  int empno;
   char name[20];
   float salary;
   ispublic:
   Employee()
   {
   cout<<"\nDefaut Construtor Called :";
   empno=1;
   strcpy(name,"ABC");
   salary =4987;
   }
   ~Employee()
   {
   cout<<"\nDestructor called";
   }
```

```
    void putdata()
    {
    cout<<"\nDisplay the details :";
    cout<<"\nEmployee Number       : "<<empno;
    cout<<"\nEmployee Name         : "<<name;
    cout<<"\nEmployee Salary       : "<<salary;
    }
};
main()
{   Employee E;
    E.putdata();
}
/*Defaut Construtor Called :
Display the details :
Employee Number       : 1
Employee Name         : ABC
Employee Salary       : 4987
Destructor called*/
```

Parameterized constructors:

The constructors that can take arguments are called a parameterized constructor. When a constructor has been parameterized the object declaration statement such as

    Sample A;

may not work. We must pass initial values as arguments to the constructor function when an object is declared. This can be done in two ways:

- By calling the constructor explicitly → Sample A = Sample (“shalini”,101);
- By calling the constructor implicitly → Sample A(“shalini”,101);

The below example describes a parameterized constructor:

```
#include<iostream.h>
#include<string.h>
class Employee
{   int empno;
    char name[20];
    float salary;
    public:
    Employee(int e,char *n,float s)
    {
    cout<<"\nParamerized Construtor Called :";
    empno=e;
    strcpy(name,n);
    salary =s;
    }
    ~Employee()
    {
    cout<<"\nDestructor called";
    }
    void putdata()
    {
    cout<<"\nDisplay the details :";
    cout<<"\nEmployee Number        : "<<empno;
    cout<<"\nEmployee Name          : "<<name;
    cout<<"\nEmployee Salary        : "<<salary;
    }
};
main()
{   Employee E(201,"AJAY",3000);
    E.putdata();
}
/*Paramerized Construtor Called :
Display the details :
Employee Number        : 201
Employee Name          : AJAY
Employee Salary        : 3000
Destructor called*/
```

Copy Constructor:

Copy Constructor allows you to have the object of the same class type but with reference operator which means constructor can take parameter of the same class type but only through call by reference and not as call by value.

As the name suggests the copy constructor refers to the input object and initialize the current object by the values of the parameter object. The invocation of the copy constructor will look like:

```
        Sample ob1(ob2)  or
        Sample ob1 = ob2;
```

```
#include<iostream.h>
class Sample
{   int a;
    float b;
    char c;
    public:
    Sample()
    {
    a=10;
    b=56.78;
    c='H';
    }
    Sample(Sample &ptr)
    {
    a=ptr.a;
    b=ptr.b;
    c=ptr.c;
    }
    void display()
    {
    cout<<"\na ="<<a;
    cout<<"\nb ="<<b;
    cout<<"\nc ="<<c;
    }
};
main()
{  Sample S1;
    cout<<"\nDisplay of S1 object :";
    S1.display();
    Sample S2(S1);
    cout<<"\nDisplay of S2 object :";
    S2.display();
}
/*Display of S1 object :
a =10
b =56.779999
c =H
Display of S2 object :
a =10
b =56.779999
c =H*/
```

Dynamic Initialization of objects:

Class objects can be initialized dynamically too. Here the initial values of an object may be provided at run time. One advantage of dynamic initialization formats using overloaded constructors. This provides initialization formats using overloaded constructors. This provides the flexibility of using different format of data at runtime depending upon the situation. Consider a banking system in which data members are used to store account no, balance and account type. The program illustrates how to use class variables for holding the account details and how to construct these variables at run time using dynamic initialization.

```
#include<iostream.h>
class Account
{
    int actno;
    float amount;
    char type;
    public:
    Account(){}
    Account(int no,float bal,char t='s')
    {
    actno=no;
    amount=bal;
    type=t;
    }
    void display()
    {
    cout<<"\nAccount No   : "<<actno;
    cout<<"\nBalance Amt  : "<<amount;
    cout<<"\nAccount type : "<<type;
    }
};
main()
{
    int n;
    float b;
    char t;
    cout<<"\nEnter the no, bal & type of acct :";
    cin>>n>>b>>t;
    Account A1(n,b,t);
    cout<<"\nEnter the no, bal of acct :";
    cin>>n>>b;
    Account A2(n,b);
    A1.display();
    A2.display();
}
/*Enter the no, bal & type of acct :1 1000 c
Enter the no, bal of acct :2 5000
Account No   : 1
Balance Amt  : 1000
Account type : c
Account No   : 2
Balance Amt  : 5000
Account type : s*/
```

Destructors:

Destructors as the name suggests are used to destroy or clean up the memory used by the objects. They are used to destroy objects which are created by the constructors. That is the reason they are named as destructors which is opposite in sense from the constructors. Like a constructor the destructor is a member function whose name is the same as the class name preceded by tilde ('~'). For example the destructor name for the class sample is written as:

~Sample( )

Destructors do not take any arguments and they don't return any value. The destructors get automatically invoked upon the exit from the program or function or block according to the scope of the object created. Destructor like constructor is a special member function and should be declared in the public definition part of the class declaration. If the new operator is used by the constructor to allocate memory to objects data member the destructor should use the delete operator to free the memory allocated. An example to implement a Constructor and Destructor function is explained below:

```
#include<iostream.h>
int no_of_objects=0;
class Sample
{
   public:
   Sample()
   {
        no_of_objects++;
        cout<<"\nNo of objects created :"<<no_of_objects;
   }
   ~Sample()
   {
        cout<<"\nNo of objects destroyed :"<<no_of_objects;
        no_of_objects--;
   }
};
void function()
{
   cout<<"\nBlock of function entered : ";
   Sample S6;
}
main()
{
   cout<<"\nMain entered :";
   Sample S1,S2,S3;
   {
        cout<<"\nBlock 1 entered : ";
        Sample S4,S5;
   }
   function();
   {
        cout<<"\nBlock 2 entered :";
        Sample S7;
   }
   cout<<"\nMain reenterd";
}
/*Main entered :
No of objects created :1
No of objects created :2
No of objects created :3
Block 1 entered :
No of objects created :4
No of objects created :5
No of objects destroyed :5
No of objects destroyed :4
Block of function entered :
No of objects created :4
No of objects destroyed :4
Block 2 entered :
No of objects created :4
No of objects destroyed :4
Main reenterd
No of objects destroyed :3
No of objects destroyed :2
No of objects destroyed :1*/
```

Static data member:

A variable can be made as static by including the static keyword at the beginning of the variable's declaration. So the syntax looks like this:

```
class X
{
        public:
        static int n;  //declaration of n as static data member
};
int X::n;   //definition of n
```

- It is initialized to zero when the first object of the class is created. No other initialization is permitted.
- Only one copy of the data member is created for the entire class and is shared by the objects of that class no matter how many objects are created.
- It is visible only within the class but its lifetime is in the entire program.

The type and scope of each static member must be defined outside class definition. This is necessary because static members are stored separately rather than as part of the object. As they are associated with class and not object so they are called as class variables.

```
#include<iostream.h>
class Sample
{
   static int count;
   int number;
   public:
   void getdata(int a)
   {
   number =a;
   count++;
   }
   void getcount()
   {
   cout<<"\nCount  = "<<count;
   }
   void display()
   {
   cout<<"\nNumber = "<<number;
   }
};
int Sample::count;
```

```
main()
{
   Sample A,B,C;
   A.getcount();
   B.getcount();
   C.getcount();
   A.getdata(10);
   B.getdata(20);
   C.getdata(30);
   A.getcount();
   A.display();
   B.getcount();
   B.display();
   C.getcount();
   C.display();
}
/*Count  = 0
Count  = 0
Count  = 0
Count  = 3
Number = 10
Count  = 3
Number = 20
Count  = 3
Number = 30*/
```

Friend function:

Friend functions when declared for a class definition:

- Can access the member's private as well as public of the class directly using the objects name e.g. object1.a and so on?
- It can be declared in public part of the class declaration and will not affect the utility and meaning anyway.
- Can accept the objects arguments when declared to do so.
- Is not in the scope of the class and hence can be used as a normal function
- It cannot be called using the object name as it is not part of the class's objects.

The declaration of such function is to be done by writing the word ***'friend'*** just before the function header in the declaration. Below a program finds the maximum of two private members present in different classes.

```
#include<iostream.h>
class XYZ;
class ABC
{
   int a;
```

```
    public:
    ABC(int x)
    {
    a=x;
    }
    void display()
    {
    cout<<"\nA ="<<a;
    }
    friend void max(ABC,XYZ);
};
class XYZ
{
    int b;
    public:
    XYZ(int x)
    {
    b=x;
    }
    void display()
    {
    cout<<"\nB ="<<b;
    }
    friend void max(ABC,XYZ);
};
void max(ABC A,XYZ X)
{
    if(A.a > X.b)
    cout<<"\nMaximum No is :"<<A.a;
    else
    cout<<"\nMaximum No is :"<<X.b;
}
main()
{
    ABC A1(10);
    XYZ X1(30);
    max(A1,X1);
}
/*Maximum No is :30*/
```

Exercise:

1. Explain the following terms: Classes and objects
2. Distinguish between the following terms:

   i) Objects and Classes   ii) Data Abstraction and Data Encapsulation
3. Define a class to represent a bank account include the following data members:

   i) Name of the Depositor   ii) Type of account   iii) Account Number   iv) Balance

   and the following member functions:

i) To assign initial values ii) To deposit an amount

iii) To withdraw an amount after checking the balance iv) To display name and balance

Write a main ( ) function to test the program.

4. Illustrate with an example the concept of array of objects
5. Distinguish between structures and classes?
6. Create a student class which has the following fields:-

Name, Roll No, Marks1, Marks2, Marks3. Enter data for three students and display it?

7. What do you mean by dynamic initialization of objects? How it is achieved?
8. State giving reasons whether the following statements are true or false:

a) Functions cannot return class objects

b) Data members can be initialized under class specifier.

9. "A base class is never used to create objects". Justify?
10. What is abstract class? What is local class?
11. Explain static class members with one example each?
12. Explain access specifiers with example?
13. Explain the concept of data hiding?
14. What are copy constructors? Explain their need?
15. What are parameterized constructors explain their need?
16. Write a program to generate Fibonacci series using constructors for the following cases:
    i) Member function has been defined in the scope of class definition
    ii) Member function has been defined outside the scope of class definition
17. Write an object oriented that use Euclid's algorithm to display the G.C.D of two integers. Incorporate member functions for data input, display and calculating G.C.D.
18. Write a program to perform different operations on complex numbers using constructors for the following cases:
    i) Using inline member function ii) Using external member function
19. Write a short note on constructor with default arguments?

INHERITANCE:

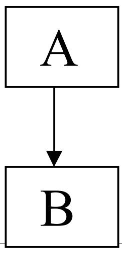
Inheritance is defined as a process of reusability. It is always nice if we could reuse something that already rather than trying to create the same allover again. It increases the reliability. For instance the reuse of a class that has already been tested, debugged and used many times can save the effort of developing and testing the same again. This is done by creating new classes reusing the properties of existing ones. The mechanism of deriving a new class from an old one is called as inheritance. The old class is referred to as the base class and the new one is called as the derived class or subclass.

Types of Inheritance:

The derived class inherits some or all of the features from the base class. A class can also inherit properties from more than one class or from more than one level.

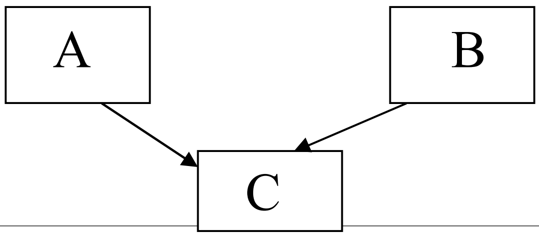
1. **Single Inheritance:**

   A derived class with only one base class is called as **Single Inheritance.**



2. **Multiple Inheritance:**

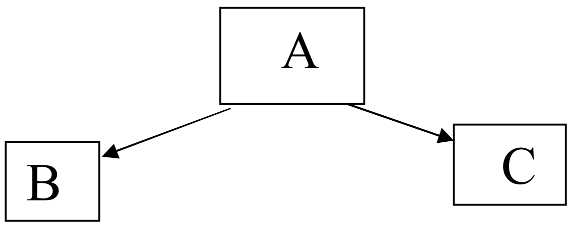
   A derived class with several base classes is called **Multiple Inheritance.**



3. **Hierarchical Inheritance**:

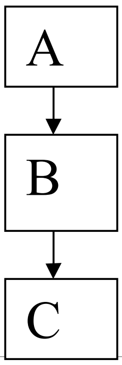
   More than one class this process is called as Hierarchical Inheritance may inherit the features of one class.

.



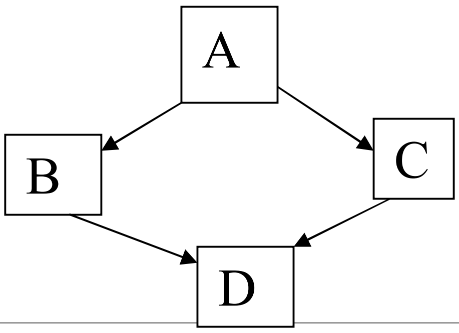
4. **Multilevel Inheritance:**

The mechanism of deriving a class from another derived class is known as **Multilevel Inheritance.**



5. Hybrid Inheritance:

There could be situations where we need to apply more two or more types of inheritances to design a program this process is called **Hybrid Inheritance.**



Public Inheritance:

When the class is publicly inherited public member of the base class becomes a public member of the derived class and therefore they are accessible ton the objects of the derived classes. The private members are not inherited and therefore the private members of a base class will never become the member of derived class. The program below shows the base classes can be inherited publicly.

```
#include<iostream.h>
class A
{
        int a;
        public:
        int b;
        void getdata(int x,int y)
        {
                a=x;
                b=y;
        }
        int get_a()
        {
                return(a);
        }
        void display()
        {
                cout<<"\na= "<<a;
                cout<<"\nb= "<<b;
        }
};
class B:public A
{
        int c;
        public:
        void mul()
        {
                c=get_a()*b;
        }
        void displayc()
        {
                cout<<"\nc ="<<c;
        }
};
main()
{
        B B1;
        B1.getdata(10,30);
        B1.display();
        B1.mul();
        B1.displayc();
}
/*a= 10
b= 30
c =300*/
```

Private Inheritance:

When a derived class privately inherits a base class, public members of the base class become private members of the derived class and therefore the public members of the base class can only be accessed by the member functions of the derived class. They are inaccessible to the objects of the derived class. Its own objects using the dot operator can access a public member of a class. The result is that no member of the base class is accessible to the objects of the derived class. The private members are not inherited and therefore the private members of a base class will never become the members of it's derive class. The below program explains how the base class can be privately inherited.

```
#include<iostream.h>
class A
{
        int a;
        public:
```

```
        int b;
        void getdata(int x,int y)
        {
                a=x;
                b=y;
        }
        int get_a()
        {
                return(a);
        }
        void display()
        {
                cout<<"\na= "<<a;
                cout<<"\nb= "<<b;
        }
};
class B:private A
{
        int c;
        public:
        void mul()
        {
                getdata(10,30);
                c=get_a()*b;
        }
        void displayc()
        {
                display();
                cout<<"\nc ="<<c;
        }
};
main()
{
        B B1;
        B1.mul();
        B1.displayc();
}
/*a= 10
b= 30
c =300*/
```

Protected Members:

C++ provides a third visibility modifier ‘protected’, which serves a limited purpose in inheritance. A member declared as protected is accessible by the member functions within its class and any class immediately derived from it. It cannot be accessed by the functions outside these two classes.

When a protected member is inherited in public mode it becomes protected in the derived class too and therefore is accessible by the member function of the derived class. A protected member inherited in the private mode derivation becomes private in the derived class. In protected derivation both public and protected members of the base class become protected members of the base class become protected of the derived class. So here using protected members we are trying to achieve multilevel inheritance.

Multilevel Inheritance with protected members:

A class ‘A’ can be derived from another derived class ‘B’. The class ‘B’ which in turn serves as the base class for the derived class C. This is called as Multilevel Inheritance and is been explained with the help of the following program:

```
#include<iostream.h>
#include<string.h>
class Student
{
        int RollNo;
        char name[20];
        public:
        void getdata(int r,char *n)
        {
                RollNo=r;
                strcpy(name,n);
        }
        void putdata()
        {
                cout<<"\nStudent Details : ";
                cout<<"\nRoll No : "<<RollNo;
                cout<<"\nName    : "<<name;
        }
};
class Test:public Student
{       protected:
        int m1,m2;
        public:
        void getmarks(int x,int y)
        {
                m1=x;
                m2=y;
        }
        void putmarks()
        {
                cout<<"\n Marks 1 : "<<m1;
                cout<<"\n Marks 2 : "<<m2;
        }
};
class Result : public Test
{       int total;
        public:
        void gettotal()
        {
                total=m1+m2;
        }
        void display()
        {
                cout<<"\nTotal Marks   : "<<total;
                cout<<"\nAverage Marks : "<<total/2;
        }
};.
main()
{       Result R;
        R.getdata(1,"Shalini");
        R.getmarks(53,67);
        R.gettotal();
        R.putdata();
        R.putmarks();
        R.display();
}
/*Student Details :
Roll No : 1
Name    : Shalini
 Marks 1 : 53
 Marks 2 : 67
Total Marks   : 120
Average Marks : 60*/
```

Multiple Inheritance:

A class that inherits the attributes of two or more classes is known as multiple inheritance. For example if class D is derived from the three different classes A, B and C by three different mechanisms:

```
class D : public A, private B, protected C
{
……………
};
```

Multiple Inheritance can be explained with the help of the following program:

```
#include<iostream.h>

class A
{       protected:
        int a;
        public:
        void geta(int x)
        {
                a=x;
        }
};
class B
```

```
{       protected:
        int b;
        public:
        void getb(int x)
        {
                b=x;
        }
};
class C:public A,public B
{       protected:
        int c;
        public:
        void mul()
        {
                c=a*b;
        }
        void display()
        {
                cout<<"\na = "<<a;
                cout<<"\nb = "<<b;
                cout<<"\nc = "<<c;
        }
};
main()
{
        C C1;
        C1.geta(10);
        C1.getb(20);
        C1.mul();
        C1.display();
}
/*a = 10
b = 20
c = 200*/
```

Hybrid Inheritance:

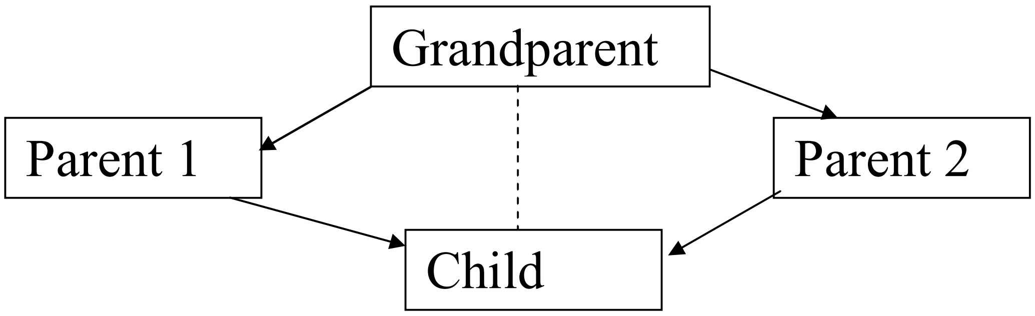
This type of inheritances may also be referred to as **mixed inheritances.** As the name suggests it is a mixture of all the previous kinds of mechanisms we saw namely single inheritance, multilevel inheritance and hierarchical inheritance. Hybrid Inheritance is implemented in the following program:

```
#include<iostream.h>
#include<string.h>
class Student
{       int RollNo;
        char name[20];
        public:
        void getdata(int r,char *n)
        {
                RollNo=r;
                strcpy(name,n);
        }
        void putdata()
        {
                cout<<"\nStudent Details : ";
                cout<<"\nRoll No : "<<RollNo;
                cout<<"\nName    : "<<name;
        }
};
class Test:public Student
{       protected:
        int m1,m2;
        public:
        void getmarks(int x,int y)
        {
                m1=x;
                m2=y;
        }
        void putmarks()
        {
                cout<<"\n Marks 1 : "<<m1;
                cout<<"\n Marks 2 : "<<m2;
        }
};
class Sports
{       protected:
        int score;
        public:
        void getscore(int x)
        {
                score=x;
        }
        void putscore()
        {
                cout<<"\nSport's score : "<<score;
        }
};
class Result : public Test,public Sports
{       int total;
        public:
        void gettotal()
        {
                total=m1+m2+score;
        }
        void display()
        {
                cout<<"\nTotal Marks   : "<<total;
                cout<<"\nAverage Marks : "<<total/3;
        }
```

```
};
main()
{       Result R;
        R.getdata(1,"Shalini");
        R.getmarks(53,67);
        R.getscore(66);
        R.gettotal();
        R.putdata();
        R.putmarks();
        R.putscore();
        R.display();
}
/*Student Details :
Roll No : 1
Name    : Shalini
Marks 1 : 53
Marks 2 : 67
Sport's score : 66
Total Marks   : 186
Average Marks : 62*/
```

Virtual Base class:

Consider a situation where all the three kinds i.e. multilevel, multiple and hierarchical inheritances are invoked.



The ***child*** has two direct base classes ***parent1*** and ***parent2*** which themselves have a common base class ***grandparent.*** The child inherits the features of ***grandparent*** via two separate paths. It can also inherit directly as shown by the broken line. The grandparent is sometimes referred to as indirect base class. Inheritance by the child as shown in above figure might pose some problems. All the public and protected members of grandparent are inherited into child twice first via parent1 and via parent2. This means child would have duplicate set of members inherited from grandparent. This introduces ambiguity and should be avoided. The duplication of inherited members due to these multiple paths can be avoided by making common base classes which is shown as follows:

```
#include<iostream.h>
#include<string.h>
class Person
```

```
{
        int code;
        char name[10];
        public:
        void getdata(int c,char *n)
        {
                code=c;
                strcpy(name,n);
        }
        void putdata()
        {
                cout<<"\nEmployee Details : ";
                cout<<"\nEmployee Code    : "<<code;
                cout<<"\nName : "<<name;
        }
};
class Account : virtual public Person
{       protected:
        float pay;
        public:
        void getpay(float p)
        {
                pay=p;
        }
};
class Admin : public virtual Person
{       protected:
        int exp;
        public:
        void getexp(int p)
        {
                exp=p;
        }
};
class Master: public Account,public Admin
{       public:
        void display()
        {
                cout<<"\nNo of yrs of experience : "<<exp;
                cout<<"\nSalary of the person            : "<<pay;
        }
};
main()
{       Master M;
        M.getdata(1,"Shalini");
        M.getpay(24000);
        M.getexp(8);
        M.putdata();
        M.display();
}
```

```
        /*Employee Details :
        Employee Code    : 1
        Name : Shalini
        No of yrs of experience : 8
        Salary of the person      : 24000*/
```

Constructors and Destructors in Hierarchical Inheritance:

Both base and derived class can have constructors and destructors. This section explains how constructors of base and derived class are called and how can parameters be passed to base class constructor functions. It is important to understand the order in which these functions are executed when an object of a derived class into existence and when it goes out of existence. The below program demonstrates the working of Constructors and Destructors in Hierarchical Inheritance.

```
#include<iostream.h>
#include<string.h>
class Staff
{       int empcode;
        char name[20];
        public:
        Staff(int e,char *n)
        {
                cout<<"\nConstructor Staff called";
                empcode=e;
                strcpy(name,n);
        }
        void display()
        {
                cout<<"\nEmployee Code : "<<empcode;
                cout<<"\nName                  : "<<name;
        }
        ~Staff()
        {
                cout<<"\nDestructor of Staff called";
        }
};
class Typist : public Staff
{       float speed;
        public:
        Typist(int e,char *n,float s):Staff(e,n)
        {
                cout<<"\nConstructor of Typist called";
                speed=s;
        }
        void displayT()
        {
                cout<<"\nSpeed   :"<<speed;
        }
```

```
		~Typist()
		{
			cout<<"\nDestructor of Typist called";
		}
};
class Admin: public Staff
{	char post[20];
	public:
	Admin(int e,char *n,char *n1):Staff(e,n)
	{
		cout<<"\nConstructor of Admin called";
		strcpy(post,n1);
	}
	void displayT()
	{
		cout<<"\nPost	:"<<post;
	}
	~Admin()
	{
		cout<<"\nDestructor of Admin called";
	}
};
class Teacher:public Staff
{	int exp;
	public:
	Teacher(int e,char *n,int s):Staff(e,n)
	{
		cout<<"\nConstructor of Teacher called";
		exp=s;
	}
	void displayT()
	{
		cout<<"\nExperience	:"<<exp;
	}
	~Teacher()
	{
		cout<<"\nDestructor of Teacher called";
	}
};
main()
{	Typist T(1,"Ajay",80);
	T.display();
	T.displayT();
	Admin A(2,"Arun","manager");
	A.display();
	A.displayT();
	Teacher T1(3,"Shalini",8);
	T1.display();
	T1.displayT();
}
```

```
/*Constructor Staff called
Constructor of Typist called
Employee Code : 1
Name                    : Ajay
Speed   :80
Constructor Staff called
Constructor of Admin called
Employee Code : 2
Name                    : Arun
Post   :manager
Constructor Staff called
Constructor of Teacher called
Employee Code : 3
Name                    : Shalini
Experience   :8
Destructor of Teacher called
Destructor of Staff called
Destructor of Admin called
Destructor of Staff called
Destructor of Typist called
Destructor of Staff called*/
```

Container class:

Inheritance is a mechanism of deriving certain properties of one class into another. C++ supports yet another way of inheriting properties of one class into another. This approach takes a view that an object can be a collection of many other objects. A class can contain objects of other classes as its members as shown below:

```
class A
{
…………….
…………….
};
class B
{
…………….
…………….
};
class C
{
A a;
B b;
};
```

- All objects of C class will contain the objects ***'a'*** and ***'b'***. This kind of relationship is called containership or nesting
- Creation of an object that contains another object is very different than the creation of an independent object.

- Its constructor creates an independent object when it is declared with arguments. On the other hand a nested object is created in two stages. First the member objects are created using their respective constructors and then the other ordinary members are created.
- This means constructors of all the member objects should be called before its own constructor body is executed.

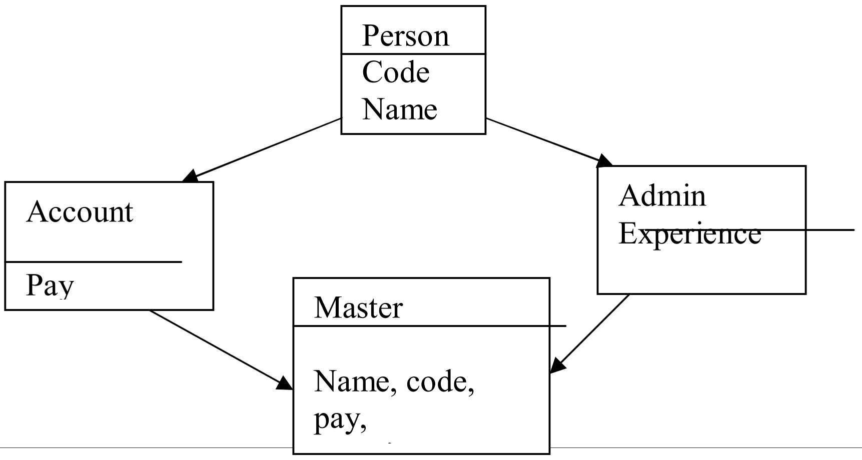
Function Overriding:

Method overriding is redefining the method or member function in the derived class with the same name as that of the base class. When a member function in the base class is overridden in the derived class the object of the derived class cannot access the functions original definition from the base. It can only access the new definition implemented in the derived class. Let us explain function overriding with the help of below example:

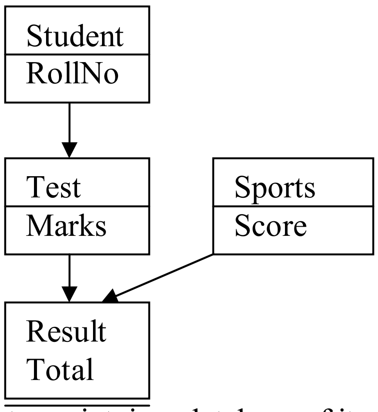
```
#include<iostream.h>
#include<string.h>
class employee
{       int empno;
        char name[20];
        public:
        void set(int e,char *s)
        {
                empno=e;
                strcpy(name,s);
        }
        void displayH()
        {
                cout<<"\nEmployee Information : ";
        }
        void display()
        {
                cout<<"\nEmp No  : "<<empno;
                cout<<"\nName   : "<<name;
        }
};
class Manager:public employee
{       int no_of_projects;
        public:
        void set(int e,char *n,int e1)
        {
                no_of_projects=e1;
                employee::set(e,n);
        }
        void displayH()
        {
                cout<<"\nManager Information : ";
        }
        void displayP()
        {
                cout<<"\nNo of projects  : "<<no_of_projects;
        }
};
main()
{       Manager M;
        M.set(1,"shalini",5);
        M.displayH();
        M.display();
        M.displayP();
}
/*Manager Information :
Emp No  : 1
Name   : shalini
No of projects  : 5*/
```

Exercise:

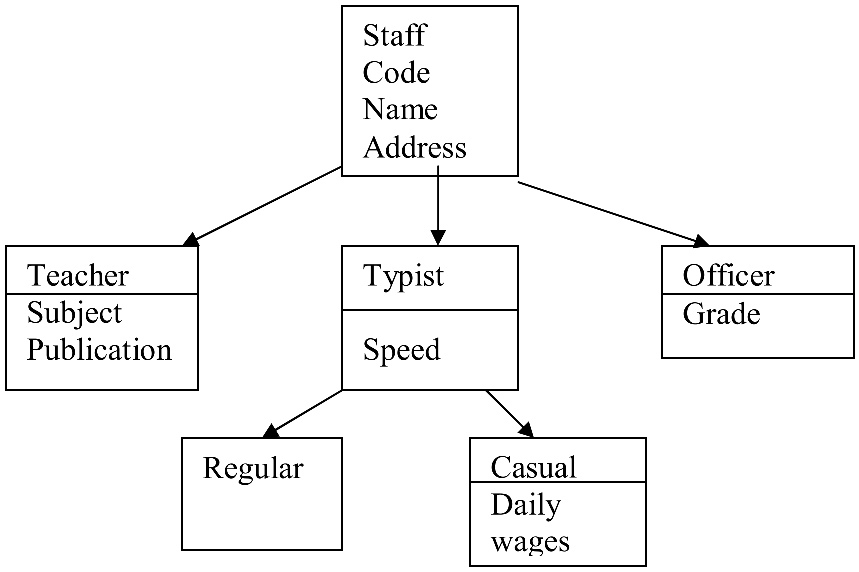
1. What does inheritance mean in C++.
2. State giving reasons whether the statement is true/false: “Inheritance aids data hiding”.
3. Describe the syntax of single inheritance?
4. What are the various types of Inheritance? What ambiguity is faced during multiple inheritance and how it is resolved?
5. What are the different forms of Inheritance? Give an example of each?
6. Write a short note on visibility of members in Inheritance.
7. Describe the syntax of multiple inheritance when do we use such an inheritance?
8. When do we use protected visibility specifier to a class member?
9. Explain inheritance in object oriented programming in C++ in terms of multiple inheritance, level of inheritance. Deriving classes in public, private and protected mode. Use suitable programs and class definitions for explaining the concepts?
10. What is virtual base class?
11. Consider a class network given below. The class master derives information from both account and admin classes which in turn derive information from the class person. Define all the four classes and write a program to create, update and display the information contained in master objects.



12.An educational institution wishes to maintain a database of its students. The database is divided into number of classes whose inheritance relationships are shown below. The figure also shows the minimum information required for each class. Specify all classes, define functions to create database and retrieve information as and when required.



13. An educational institution wishes to maintain a database of its employees. The database is divided into number of classes whose inheritance relationships are shown below. The figure also shows the minimum information required for each class. Specify all classes, define functions to create database and retrieve information as and when required.



Operator Overloading:

Operator overloading provides a flexible for the creation of new definitions for the creation of new definitions for most of the C++ operators. We can overload all he C++ operators except the following:

- Sizeofoperator()
- Conditional operator : (? :)
- Scope resolution operator (: :)
- Class member accessing operator '.' And '*'

The operator symbols can be overloaded means a different semantics can be attached to these symbols. Here as operator symbols are standard and defined by C++ we cannot change their syntax of usage. When the operator is overloaded its original meaning defined by C++ is not changed or lost. For example a '+' operator can be overloaded to concatenate two string operands but still it can be used two add two integers in a normal way.

The key to overload an operator is to associate the operator function with a class object. The operator thus can be given different meaning but in the context of the class for which it is defined. The operator function can be defined as follows:

```
<return_type> classname :: operator <operator symbol> (<argument_list>)
{
        //function body
}
```

Unary Operator Overloading:

We know that unary operators work with a single operand. For example increment operator '++' increases the value by one. Refer the following program:

```
#include<iostream.h>
class unary
{
        int x;
        public:
        unary(int a)
        {
                x=a;
        }
        void operator++()
        {
                x++;
        }
        void display()
        {
                cout<<"\nx="<<x;
        }
};
main()
{
        unary u(2);
        u.display();
        u++;
        u.display();
}
/*x=2
 x=3*/
```

Binary Operator Overloading:

Binary Operator requires working on two operands the object for which it is invoked and the other operand is the argument explicitly passed to it.

```
Sample Sample :: operator +(Sample obj)
{
………………
………………..
}
```

For example the overloaded binary + operator take the object argument of type 'Sample'. It adds the current object's data member from which it is invoked to the argument's data members and stores the result in a temporary object and returns the object as a resultant value of the addition. Refer the following program:

```
#include<iostream.h>
class Complex
{
        float real,imaj;
        public:
        Complex()
        {
                cout<<"\nDefault Constructor called";
        }
        Complex(float r,float i)
        {
                real=r;
                imaj=i;
                cout<<"\nParameterized Constructor called";
        }
        void display()
        {
                cout<<endl<<real<<"+i"<<imaj;
        }
        Complex operator+(Complex B)
        {
                Complex temp;
                temp.real=real+B.real;
                temp.imaj=imaj+B.imaj;
```

```
			return(temp);
		}
		~Complex()
		{
			cout<<"\nDestructor called";
		}
};
main()
{
		Complex A(2,5),B(5,7);
		Complex C;
		C=A+B;
		A.display();
		B.display();
		C.display();
}
/*Parameterized Constructor called
Parameterized Constructor called
Default Constructor called
Default Constructor called
Destructor called
2+i5
5+i7
7+i12
Destructor called
Destructor called
Destructor called*/
```

Overloading Relational Operator:

Relational operators mean to relate two things and return the resultant Boolean value. They are binary in nature as the work with two operands. The commonly known relational operators are ‘<’,’>’,’<=’,’>=’,’= =’ and ‘!=’. Relational operators can be overloaded. Refer the following program:

```
#include<iostream.h>
class abc
{
		int x;
		public:
		abc()
		{
			cout<<"\nDefault Constructor Called";
		}
		abc(int a)
		{
			x=a;
		}
		void display()
		{
```

```
                cout<<x<<endl;
        }
        int operator <(abc n)
        {
                if(x<n.x)
                        return(-1);
                else
                        return(1);
        }
};
main()
{
        abc a1(15),a2(7);
        cout<<"\na1 =";
        a1.display();
        cout<<"\na2 =";
        a2.display();
        int b=a1<a2;
        if(b<0)
                cout<<"\na1 is less than a2";
        else if(b>0)
                cout<<"\na1 is greater than a2";
}
/*
a1 =15
a2 =7
a1 is greater than a2
a1 =5
a2 =7
a1 is less than a2*/
```

Implement a class having the following members:

Private Members:

i) Two-dimensional array variable A of 10 X 10 integer elements.

ii) Number of rows and columns of the array – M and N

Public members are incorporated as member functions and the list is as follows:

i) Functions for reading a matrix from console

ii) Functions for displaying a matrix

iii) Functions for addition of two matrices

iv) Function which overloads (*) operator for multiplication of two matrices.

Write a suitable main program to test the member functions.

```
#include<iostream.h>
class Matrix
{
        int x[4][4];
        int r,c;
```

```
public:
Matrix()
{
	for(r=0;r<4;r++)
	{
		for(c=0;c<4;c++)
		{
			x[r][c]=0;
		}
	}
}
void read()
{
	for(r=0;r<4;r++)
	{
		cout<<"Enter the "<<r<<"row :";
		for(c=0;c<4;c++)
		{
			cin>>x[r][c];
		}
	}
}
void display()
{
	for(r=0;r<4;r++)
	{
		cout<<endl;
		for(c=0;c<4;c++)
		{
			cout<<x[r][c]<<" ";
		}
	}
}
Matrix operator +(Matrix M)
{
	Matrix temp;
	for(r=0;r<4;r++)
	{
		for(c=0;c<4;c++)
		{
			temp.x[r][c]=x[r][c]+M.x[r][c];
		}
	}
	return(temp);
}
Matrix operator *(Matrix M)
{
	Matrix temp;
	int k;
	for(r=0;r<4;r++)
```

```
			{
				for(c=0;c<4;c++)
				{
					for(k=0;k<4;k++)
					{
						temp.x[r][c]+=x[r][k]*M.x[k][c];
					}
				}
			}
			return(temp);
		}
};
main()
{
		Matrix A,B,C;
		A.read();
		B.read();
		C=A+B;
		A.display();
		B.display();
		cout<<"\nAfter addition :";
		C.display();
		cout<<"\nAfter Multiplication :";
		C=A*B;
		C.display();
}
/*Enter the 0row :1 2 3 4
Enter the 1row :5 6 7 8
Enter the 2row :9 10 11 12
Enter the 3row :13 14 15 16
Enter the 0row :1 2 3 4
Enter the 1row :5 6 7 8
Enter the 2row :9 10 11 12
Enter the 3row :13 14 15 16
1 2 3 4
5 6 7 8
9 10 11 12
13 14 15 16
1 2 3 4
5 6 7 8
9 10 11 12
13 14 15 16

After addition :2 4 6 8
10 12 14 16
18 20 22 24
26 28 30 32

After Multiplication :90 100 110 120
202 228 254 280
```

```
314 356 398 440
426 484 542 600   */
```

Static and Dynamic Binding:

Polymorphism means one name multiple forms. We have already seen how the concept of polymorphism is implanted using the overloaded functions and operators. The overloaded member functions are selected fro invoking matching arguments both type and number. The compiler knows the information at compile time and therefore the compiler is able to select the appropriate function for a particular call at the compile time itself. This is called as ***early binding*** or ***static binding***. Also known as compile time polymorphism, early binding simply means that an object is bound to its function call at compile time.

Let us consider a situation where the function name and prototype is the same in the base and derived class. For example consider the following class definition:

```
class X
{
        int a;
        public:
        void show( )   // show( ) is the base class
        {
        ……………
        }
};
class Y : public X
{
        int b;
        public:
        void show( )      //show( ) is the derived class
        {
        ……………
        }
};
```

The prototype of the show ( ) is the same in both the places, the function is not overloaded and therefore static binding does not apply. We have seen earlier that in such situations we may use the scope resolution operator to specify the class while invoking the functions with derived class objects.

It would be nice if the appropriate member functions could be selected while the program is running. This is known as run time polymorphism. C++ supports a mechanism known as virtual function to achieve run time polymorphism.

At run time, when it is known what class objects are under consideration the appropriate version of the function is invoked. Since the function is linked with a particular class much later after compilation, this process is termed as ***late binding*** or ***dynamic binding.***

Dynamic binding is the powerful feature of C++. This requires the use of pointer to objects

Pointer to Objects:

Pointer to Objects can be defined and used the same way we use pointer to structure variables. For example see the following code:

```
#include<iostream.h>
class Employee
{
        int empno;
        char name[20];
        float salary;
        public:
        void getdata()
        {
                cout<<"\nEnter the data for employee :"<<endl;
                cout<<"\nEnter the employee no :";
                cin>>empno;
                cout<<"\nEnter the name :";
                cin>>name;
                cout<<"\nEnter the salary :";
                cin>>salary;
        }
        void putdata()
        {
                cout<<"\nDisplay the details :";
                cout<<"\nEmployee Number        : "<<empno;
                cout<<"\nEmployee Name          : "<<name;
                cout<<"\nEmployee Salary        : "<<salary;
        }
};
main()
{
        Employee E;
        E.getdata();
        E.putdata();
}
/*
Enter the data for employee :
Enter the employee no :1
Enter the name :shalini
Enter the salary :6000
Display the details :
Employee Number        : 1
Employee Name          : shalini
Employee Salary        : 6000*/
```

The ***this*** pointer:

The member function of every object has access to pointer names ***this*** which points to the object itself. When we call a member function it comes into existence with the value of this set to the addresses of the object for which it was called. The ***this*** pointer can be treated like any other pointer to an object.

Using a ***this*** pointer any member can find out the addresses of the object to which it is a member. It can also be used to access the data in the object it points to. The following program shows the working of ***this*** pointer.

Virtual Functions:

When we use the same function in both the base and derived classes the function in base class is declared as ***virtual*** the keyword ***virtual*** in C++ determines which function to use at run time based on the type of object pointed to by the base pointer rather than the type of pointer. Thus by making the base pointer to point different objects we can execute different versions of the virtual function. The following program illustrates the concept:

```
#include<iostream.h>
#include<string.h>
class Vehicle
{
        protected:
        char make[20];
        float milage;
        public:
        Vehicle(char *m,float m1)
        {
                strcpy(make,m);
                milage=m1;
        }
        virtual void display()
        {
           cout<<"\nVehicle Details :";
        }
};
class TWheeler : public Vehicle
{
        char clutch;
        public:
        TWheeler(char *m,float m1,char c):Vehicle(m,m1)
        {
                clutch=c;
        }
        void display()
        {
           cout<<"\nMake   : "<<make;
           cout<<"\nMilage : "<<milage;
           cout<<"\nClutch : "<<clutch;
        }
};
class FWheeler : public Vehicle
```

```
{
        char ac;
        public:
        FWheeler(char *m,float m1,char c):Vehicle(m,m1)
        {
                ac=c;
        }
        void display()
        {
            cout<<"\nMake   : "<<make;
            cout<<"\nMilage : "<<milage;
            cout<<"\nAC     : "<<ac;
        }
};
main()
{
        Vehicle *b;
        TWheeler T("Hero Honda",70,'y');
        b=&T;
        b->display();
        FWheeler F("Maruti",17,'y');
        b=&F;
        b->display();
}
/*void display()*/
/*Vehicle Details :
Vehicle Details :*/
/*With using virtual: virtual void display()
Make   : Hero Honda
Milage : 70
Clutch : y
Make   : Maruti
Milage : 17
AC     : y*/
/*After using pure virtual:virtual void display()=0
Make   : Hero Honda
Milage : 70
Clutch : y
Make   : Maruti
Milage : 17
AC     : y*/
```

Rules for ***virtual functions***:

Virtual functions though are very useful for dynamic polymorphism there are some restrictions that we need to follow while using them. They are:

1. The virtual functions must be members of the same class.
2. They should not be the static members.

3. A virtual function in a base class must be defined. It could be defined with empty open and close brackets.
4. Do not make constructor virtual.
5. A pointer to derived class type cannot be used to access the object of the base type while a base class pointer could be used to point to a derived class object.
6. Prototypes declared in the base class and the other derived classes should be identical.
7. A virtual function can be friend function for another class.

Pure functions and Abstract classes:

We do not create object of vehicle class but instead create objects of the derived class. So by declaring pure virtual functions which when declared in the base class becomes an abstract class if not instantiated. A pure virtual function is a do nothing function and can be defined as:

Virtual <return type> <function_name( )> = 0;

See that the definition of the function is not even empty it is completely avoided by setting it equal to zero. That means the pure virtual function is declared in such a way has no definition relative to the base class. In such cases compiler wants the function is redefined in all the derived classes.

Example:

Create a base class called shape. Use this class to store two data type values that could be used to compute area of figures. Derive two specific classes called triangle and rectangle from the base shape. Add to the base class a member function getdata ( ), base class data member and another function display( ) compute and display the area of figures. Make display( ) as a virtual function and redefine the function in the derived class to suit their requirements. It should calculate

Area of rectangle = x * y   Area of triangle = ½ * x* y

```
#include<iostream.h>
class Shape
{
        protected:
        float x,y;
        public:
        void getdata(float a,float b)
        {
                x=a;
                y=b;
        }
        virtual void display()
        {
                cout<<"\nArea :";
        }
};
class Rectangle : public Shape
{
        public:
        void getdata(float a,float b)
        {

        }
        void display()
        {
                cout<<"\n Area of Rectangle : "<<x*y;
        }
};
class Triangle : public Shape
{
        public:
        void getdata(float a,float b)
        {

        }
        void display()
        {
                cout<<"\n Area of Triangle : "<<x*y*0.5;
        }
};
main()
{
        Triangle T;
        Rectangle R;
        Shape *ptr;
        ptr=&T;
        ptr->getdata(5,7);
        ptr->display();
        ptr=&R;
        ptr->getdata(7,8);
        ptr->display();
}/* Area of Triangle : 17.5
 Area of Rectangle : 56*/
```

Exercise:

1. Explain operator overloading with an example.
2. Explain with the example unary and binary scope resolution operator.
3. Create a class FLOAT that contains one float data member; overload all the four arithmetic operators so that they operate on the objects of FLOAT.
4. Explain operator overloading with and without using a friend function?
5. Explain the set of rules for operator overloading in C++?

6. What is an operator function? Describe the syntax of an operator function?
7. Describe a class POLAR which describes a point in the plane using polar coordinates ‘radius’ and ‘angle’. Use the overloaded ‘+’ operator to add two objects of POLAR with the help of the following formulae:

   x = r * cos(a)   a = tan (x/y)

   y = rsin(a)   y = sqrt(x*x +y *y)
8. Write a program in C++ to overload ‘+=’ operator to add two distances given in feet and inches.
9. Write a program to define a NUM class and overload the following operators:

   i) Increment operator   ii) Unary Minus

   iii) Binary Plus, Minus, Multiplication and Division
10. Show the general rules of new ( ) and delete ( ). What are some advantages and disadvantages of using them instead of malloc( ) and free ( ).
11. Write a short note on new ( ) and delete ( ) operators.
12. Create a class MATRIX of size m & n. Define all operations on matrix type of objects.
13. Develop an object-oriented program that determines the number of days in a given semester. Input to the program is year, month and day information of the first and last days of a semester. The program should accept the date information as a single string instead of accepting the year, month and day information separately. The input string is in the MM/DD/YYYY format.
14. Write a class to represent a vector (series of float values). Include different member functions to perform the following tasks:

   i) To create a vector   ii) To modify values of a given vector

   iii) To multiply by a scalar value   iv) To display the vector in the form (10,20,30….)

   Write a program to test the class
15. Write a C++ program called delchar( ) that can be used to delete characters from string. The function should take 3 arguments string name, number of characters to delete and the starting position in the string where characters should be deleted.

Function Overloading:

1. What do you mean by overloading a function? When do we use this concept?
2. Write a class to represent a vector (series of float values). Include different member functions to perform the following tasks:

   i) To create a vector   ii) To modify values of a given vector

   iii) To multiply by a scalar value   iv) To display the vector in the form (10,20,30….)

Write a program to test the class.

3. Overload ‘A’ for calculating power function value. Write a program (e.g. 8 = 2^3)
4. Write a program in C++ to overload a function squareroot( ) in three ways so that it returns squareroot of an integer, long int and double. Also write the output.
5. Write a C++ program which consists of ADD ( ) that adds two values and return their sum. Provide overloaded functions to work with integer float and string data types and test that they work with the following program segment with function references.
6. What is the difference between function overloading and overriding functions?

Virtual Functions:

1. State giving reasons whether the following statement is true or false: “Pure virtual functions force programmer to redefine virtual functions inside the derived classes”.
2. When do we make use of virtual function “pure”? What are the implications of making a function a pure virtual function?
3. Differentiate between member function and virtual function?
4. What is a virtual function? Why do we need virtual function?
5. What is polymorphism? How is run time polymorphism achieved? Explain with example?
6. Write a program using polymorphism that will accept a geometric figure and display its shape and calculate its area?
7. Explain with example how is polymorphism achieved at compile time and run time?
8. What is pure virtual function? Explain with example?

Solved University Question Papers: (Dec. 2007 and May 2008)

Dec. 2007:

1. a) Explain storage class with one programming example?

The variable's storage class tells us:

- Where the variable would be stored
- What will be the initial value of the variable if a value is assigned well and good or lse it holds some garbage as the default value?
- What is the scope of the variable; that is the scope in which function the value of the variable is available?
- What is the life of the variable; that is how long would the variable exist?

There are four storage classes:

1) Automatic Storage Class:

- Storage of variable is Memory
- Default Initial value is an unpredictable value which is often garbage.
- Scope of variable is local to the block in which the variable is defined.
- Life of variable till the control remains within block in which the variable is defined.

```
#include<iostream.h>
main()
{
    auto int i,j;
    cout<<"\ni = "<<i;
    cout<<"\nj = "<<j;
    i=2;
    j=3;
    cout<<"\ni = "<<i;
    cout<<"\nj = "<<j;
}
/*i = 1032
j = 1050
i = 2
j = 3*/
```

2) Register Storage Class :

- Storage of variable is Memory
- Default Initial value is an unpredictable value which is often garbage.
- Scope of variable is local to the block in which the variable is defined.
- Life of variable till the control remains within block in which the variable is defined

```
#include<iostream.h>
main()
```

```
{
    register int i;
    for(i=1;i<=5;i++)
            cout<<i<<" ";
}
/*1 2 3 4 5*/
```

A value stored in a CPU register can always be accessed faster than the one which is stored in memory. Therefore if a variable is used at many places in a program it is better to declare its storage class as register

3) Static Storage Class:

- Storage of variable is CPU register.
- Default Initial value is zero.
- Scope of variable is local to the block in which the variable is defined.
- Life of variable is outside the block in which the variable is defined, value of the variable persists between different function calls.

```
#include<iostream.h>
void inc();
main()
{
    inc();
    inc();
    inc();
    inc();
}
void inc()
{
    static int i;
    int x=1;
    cout<<"\n"<<i<<" "<<x;
    i++; x++;
}
/*0 1
1 1
2 1
3 1*/
```

4) External Storage Class:

- Storage of variable is Memory.
- Default Initial value is zero.
- Scope of variable is local global.
- Life of variable as long as the program’s execution does not come to an end.

External variables are declared outside all functions yet are available to all functions yet are available to all functions that care to use them.

```
#include<iostream.h>
int i;
void inc();
void dec();
main()
{
    cout<<"\ni ="<<i;
    inc();
    inc();
    dec();
    dec();
}
void inc()
{
    ++i;
    cout<<"\nOn Incrementing : "<<i;
}
void dec()
{
    --i;
    cout<<"\nOn Decrementing :"<<i;
}
/*i =0
On Incrementing : 1
On Incrementing : 2
On Decrementing :1
On Decrementing :0*/
```

b) Write a program to reverse the digit of a positive integer number?

```
#include<iostream.h>
main()
{
    int n,d,rev=0;
    cout<<"\nEnter n : ";
    cin>>n;
    while(n>0)
    {
            d=n%10;
            rev=rev*10+d;
            n=n/10;
    }
    cout<<"\nThe reverse of number is :"<<rev;
}
/*Enter n : 3487
The reverse of number is :7843*/
```

2. a) What is hybrid inheritance? What are the problems in it? How it is resolved? Give one programming example?

Already explained in Hybrid inheritance section

b) Write a program to reverse a sentence (do not reverse the words).

3. a) Write a program to shift the elements of a single dimensional array in the right direction from one position. If the given array is [76 35 55 43 22] then after execution of the program it should be [22 76 35 55 43].

```
#include<iostream.h>
#include<conio.h>
main()
{
        int x[5]={76,35,55,43,22};
        int t;
        clrscr();
        cout<<"\nDisplay the data :";
        int n=5;
        for(int i=0;i<n;i++)
        {
                cout<<x[i]<<" ";
        }
        t=x[n-1];
        for(i=n-1;i>0;i--)
        {
                x[i]=x[i-1];
        }
        x[0]=t;
        cout<<endl<<"\nMoving the array : ";
        for(i=0;i<n;i++)
        {
                cout<<x[i]<<" ";
        }
}
/*Display the data :76 35 55 43 22
Moving the array : 22 76 35 55 43*/
```

b) Write a program to find LCM and GCD of two non negative integers.

```
#include<iostream.h>
main()
{
    int a,b,x,m,n;
    cout<<"\nEnter m and n :";
    cin>>m>>n;
    a=m;
    b=n;
    if(a<b)
            x=a;
    else
            x=b;
    while(x>0)
    {
```

```
                if(a%x==0 && b%x==0)
                        break;
                else
                        x--;
        }
        cout<<"\nG.C.D of "<<m<<" and "<<n <<"is :"<<x;
        cout<<"\nL.C.M of "<<m<<" and "<<n <<"is :"<<m*n/x;
}
/*Enter m and n :45 95
G.C.D of 45 and 95is :5
L.C.M of 45 and 95is :855*/
```

4. a) Define a structure “Hockey” consisting of the following elements :

i) Player Name ii) Name of Country iii) No. of matches played iv) No. of goals scored

Write a program to read records of ‘n’ players and to prepare following lists:

i) List prepared according to player’s name

ii) List prepared according to country’s name

iii) List prepared according to number of matches played.

iv) List prepared according to number of goals scored.

```
#include<iostream.h>
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
struct hockey
{
        char name[20],country[20];
        int nm,goals;
}h[10];
main()
{
        int n,choice,ans,j;
        struct hockey temp;
        cout<<"\nEnter n :";
        cin>>n;
        for(int i=0;i<n;i++)
        {
                cout<<"\nEnter name of player : ";
                gets(h[i].name);
                cout<<"\nEnter name of country :";
                gets(h[i].country);
                cout<<"\nEnter no.of matches played : ";
                cin>>h[i].nm;
                cout<<"\nEnter no.of goals : ";
                cin>>h[i].goals;
        }
        do
        {
```

```
cout<<"\n1. List according to Player's name ";
cout<<"\n2. List according to Country's name ";
cout<<"\n3. List according to no. of matches played ";
cout<<"\n4. List according to no. of goals achieved ";
cout<<"\n5. Exit ";
cout<<"\nEnter choice :";
cin>>choice;
switch(choice)
{
        case 1: for(i=0;i<n;i++)
                        {
                       for(j=i+1;j<n;j++)
                       {
                                        if((strcmp(h[i].name,h[j].name)>0))
                                                {
                                                        temp=h[i];
                                                        h[i]=h[j];
                                                        h[j]=temp;
                                                }
                                        }
                        }
                        for(i=0;i<n;i++)
                        {
                                        cout<<"\nPlayer : "<<h[i].name;
                                        cout<<"\nCountry : "<<h[i].country;
                                        cout<<"\nMatches : "<<h[i].nm;
                                        cout<<"\nGoals :"<<h[i].goals;
                        }
                        break;
        case 2: for(i=0;i<n;i++)
                        {
                                        for(j=i+1;j<n;j++)
                                        {
                                        if((strcmp(h[i].country,h[j].country)>0))
                                                {
                                                        temp=h[i];
                                                        h[i]=h[j];
                                                        h[j]=temp;
                                                }
                                        }
                        }
                        for(i=0;i<n;i++)
                        {
                                        cout<<"\nPlayer : "<<h[i].name;
                                        cout<<"\nCountry : "<<h[i].country;
                                        cout<<"\nMatches : "<<h[i].nm;
                                        cout<<"\nGoals :"<<h[i].goals;
                        }
                        break;
        case 3: for(i=0;i<n;i++)
```

```
                                {
                                        for(j=i+1;j<n;j++)
                                        {
                                            if(h[i].nm<h[j].nm)
                                            {
                                                temp=h[i];
                                                h[i]=h[j];
                                                h[j]=temp;
                                            }
                                        }
                                }
                                for(i=0;i<n;i++)
                                {
                                        cout<<"\nPlayer : "<<h[i].name;
                                        cout<<"\nCountry : "<<h[i].country;
                                        cout<<"\nMatches : "<<h[i].nm;
                                        cout<<"\nGoals :"<<h[i].goals;
                                }
                                break;
                      case 4:for(i=0;i<n;i++)
                                {
                                        for(j=i+1;j<n;j++)
                                        {
                                            if(h[i].goals<h[j].goals)
                                            {
                                                temp=h[i];
                                                h[i]=h[j];
                                                h[j]=temp;
                                            }
                                        }
                                }
                                for(i=0;i<n;i++)
                                {
                                        cout<<"\nPlayer : "<<h[i].name;
                                        cout<<"\nCountry : "<<h[i].country;
                                        cout<<"\nMatches : "<<h[i].nm;
                                        cout<<"\nGoals :"<<h[i].goals;
                                }
                                break;
                      case 5:exit(0);
                  }
                  cout<<"\nDo you wish to continue :(y/n) :";
                  cin>>ans;
          }
          while(ans==1);
      }
      /*Enter n :3
      Enter name of player : Watson
      Enter name of country :usa
      Enter no.of matches played : 50
```

```
Enter no.of goals : 40
Enter name of player : John
Enter name of country :Italy
Enter no.of matches played : 100
Enter no.of goals : 70
Enter name of player : Arnold
Enter name of country :England
Enter no.of matches played : 20
Enter no.of goals : 30
1. List according to Player's name
2. List according to Country's name
3. List according to no. of matches played
4. List according to no. of goals achieved
5. Exit
Enter choice :1
Player : Arnold
Country : England
Matches : 20
Goals :30
Player : John
Country : Italy
Matches : 100
Goals :70
Player : Watson
Country : usa
Matches : 50
Goals :40
Do you wish to continue :(y/n) :1
1. List according to Player's name
2. List according to Country's name
3. List according to no. of matches played
4. List according to no. of goals achieved
5. Exit
Enter choice :2
Player : Arnold
Country : England
Matches : 20
Goals :30
Player : John
Country : Italy
Matches : 100
Goals :70
Player : Watson
Country : usa
Matches : 50
Goals :40
Do you wish to continue :(y/n) :1
1. List according to Player's name
2. List according to Country's name
3. List according to no. of matches played
```

```
4. List according to no. of goals achieved
5. Exit
Enter choice :3

Player : John
Country : Italy
Matches : 100
Goals :70
Player : Watson
Country : usa
Matches : 50
Goals :40
Player : Arnold
Country : England
Matches : 20
Goals :30
Do you wish to continue :(y/n) :1

1. List according to Player's name
2. List according to Country's name
3. List according to no. of matches played
4. List according to no. of goals achieved
5. Exit
Enter choice :4

Player : John
Country : Italy
Matches : 100
Goals :70
Player : Watson
Country : usa
Matches : 50
Goals :40
Player : Arnold
Country : England
Matches : 20
Goals :30
Do you wish to continue :(y/n) :5*/
```

b) What is the difference between declaration and definition?

5. a) What is the output of the following program?

```
#include<iostream.h>
main()
{
    int a,b,ab;
    a=2;b=3;ab=4;
    char ch='c';
    cout<<ch<<" "<<(++ch)<<endl;
    cout<<a<<" "<<(++a)<<endl;
    cout<<b<<" "<<(++b)<<endl;
    cout<<ab<<" "<<(++ab)<<endl;
    cout<<a<<" "<<(!!a)<<endl;
    }
```

/*d d
3 3
4 4
5 5
3 1*/

b) Write a program to read elements of a square matrix and to transpose the matrix.

   Use a single two dimensional array.

```
  #include<iostream.h>
  main()
  {
    int x[3][3],i,j,temp;
    for(i=0;i<3;i++)
    {
            cout<<"\nEnter data for "<<i<<"row :";
            for(j=0;j<3;j++)
            {
                    cin>>x[i][j];
            }
    }
    for(i=0;i<3;i++)
    {
            for(j=0;j<3;j++)
            {
                    cout<<x[i][j]<<" ";
            }
            cout<<endl;
    }
    for(i=0;i<3;i++)
    {
            for(j=i+1;j<3;j++)
            {
                    if(i!=j)
                    {
                            temp=x[i][j];
                            x[i][j]=x[j][i];
                            x[j][i]=temp;
                    }
            }
    }
    for(i=0;i<3;i++)
    {
            for(j=0;j<3;j++)
            {
                    cout<<x[i][j]<<" ";
            }
            cout<<endl;
    }
  }
  /*Enter data for 0row :1 2 3
  Enter data for 1row :4 5 6
  Enter data for 2row :7 8 9
  1 2 3
  4 5 6
  7 8 9
  1 4 7
  2 5 8
  3 6 9*/
```

6. a) What is copy constructor? Give one programming example of it?

   Already explained in Constructor section

   b) Write a program top find value of y using recursive function where y = x ^{n}, x and y are real number and 'n' is an integer.

```
#include<iostream.h>
int power(int x,int n)
{
    if(n==0)
            return(1);
    else
            return(x*power(x,n-1));
}
main()
{
    int x,n;
    cout<<"\nEnter x and n : ";
    cin>>x>>n;
    cout<<"\nPower of "<<x<<" raised to "<<n<<" is :"<<power(x,n);
}
/*Enter x and n : 5 3
Power of 5 raised to 3 is :125*/
```

7. a) Write a program to find how many objects of a class have been created using static member function?

   A static function can have access to only static members declared in the same class.

   The **static** function **showcount()** displays number of objects created till that moment.

   A count of number of objects created is maintained by the **static** variable count.

   Already explained in the static member section.

7. b) Write a program to find area of circle, area of rectangle and area of triangle using overloaded constructor function?

```
#include<iostream.h>

class Area
{
		float r,b,h;
		int x,y;
		public:
		Area(float a)
		{
			r=a;
			cout<<"\nArea of circle :"<<3.14*r*r;
		}
		Area(float d,float e,float f)
		{
			b=d;
			h=e;
			cout<<"\nArea of triangle : "<<d*e*f;
		}
		Area(int a,int b)
		{
			x=a;
			y=b;
			cout<<"\nArea of rectangle : "<<x*y;
		}
	};
	main()
	{
		clrscr();
		Area A1(3.5);
		Area A2(7.5,8.6,0.5);
		Area A3(5,8);
	}
	/*Area of circle :38.465
	Area of triangle : 32.25
	Area of rectangle : 40*/
```

May 2008:

1. a) Explain call-by value and call-by reference with one programming example each?

   Already explained in the Pointer section

   b) What is the output of the following code?

```
     i) #include<iostream.h>
   main()
   {
    int a=3;
    cout<<" "<<a<<endl;
    cout<<" "<<a++<<endl;
    cout<<" "<<++a<<endl;
   }
```

/* 3
3
5*/

```
   ii) #include<iostream.h>
   main()
   {
    int x=4,y=9;
    int z;
    z=x+++--y+y;
    cout<<"\nValue = "<<z<<endl;
    z=--x+x+y--;
    cout<<"\nValue = "<<z<<endl;
   }
```

/*Value = 20
Value = 16*/

```
   iii) #include<iostream.h>
   main()
   {
    int a,b,c;
    a=2;b=5;c=10;
    cout<<"\nValue = "<<(a+b*-c)<<endl;
    cout<<"\nValue = "<<(-c/b*c-a)<<endl;
    cout<<"\nValue = "<<(-a+++b%a)<<endl;
   }
```

/*Value = -48
Value = -22
Value = -1*/

2. a) What is object oriented programming? List basic concepts of it? Explain each concept?

   Already explained in the Object Oriented Programming section

   b) Write a program to check whether a number entered is an Armstrong number?

```
  #include<iostream.h>
  main()
```

```
{
 int m,n,x,sum=0;
 cout<<"\nEnter a number :";
 cin>>n;
 m=n;
 while(n>0)
 {
   x=n%10;
   sum=sum+(x*x*x);
   n=n/10;
 }
 if(m==sum)
   cout<<"\nArmstrong number is : "<<m;
 else
   cout<<"\n Not an Armstrong number is : "<<m;
}
/*Enter a number :153
Armstrong number is : 153
Enter a number :132
Not an Armstrong number is : 132*/
```

3. a) Explain storage classes with one programming example?

Already explained above.

b) Write a C++ program to display:

```
 * * * *
   * * *
     * *
       *
```

```
#include<iostream.h>
main()
{
int i,j;
for(i=1;i<=4;i++)
{
  for(j=4;j>=i;j--)
  {
          cout<<" * ";
  }
  cout<<endl;
}
}
```

4. a) Write a program to read and display elements of a square matrix and then transpose the matrix. Use single two dimensional matrix throughout the program?

Already explained in the paper Dec 2007

b) Explain bitwise operators in C++?

5. a) Explain recursion and write a recursive function for calculating factorial of n number?

```
#include<iostream.h>
```

```
int fact(int n)
{
if(n==0)
  return(1);
else
  return(n*fact(n-1));
}
main()
{
int n;
cout<<"\nEnter n : ";
cin>>n;
cout<<"\nFactorial of "<<n<<" is :"<<fact(n);
}
/*Enter n : 7
Factorial of 7 is :5040*/
```

b) Write a program to find how many objects of a class have been created using static member function?

Already explained in the above section.

6. a) Write a C++ program to find volume of cube, cylinder and rectangle using method overloading?

b) Define structure within a structure consisting of the following elements:

i) Employee Code ii) Employee Name iii) Salary iv) Date of Joining

```
#include<iostream.h>
struct date
{
  int dd,mm,yy;
};
struct employee
{
  int empcode;
  char empname[20];
  float salary;
  struct date doj;
};
main()
{
  struct employee emp[5];
  int i,n;
  cout<<"\nEnter no. of records :";
  cin>>n;
  for(i=0;i<n;i++)
  {
          cout<<"\nEnter Employee Code : ";
          cin>>emp[i].empcode;
          cout<<"\nEnter Employee Name : ";
```

```
                cin>>emp[i].empname;
                cout<<"\nEnter Employee Salary :";
                cin>>emp[i].salary;
                cout<<"\nEnter Employee Date of Joining :";
                cin>>emp[i].doj.dd>>emp[i].doj.mm>>emp[i].doj.yy;
        }
        for(i=0;i<n;i++)
        {
                cout<<"\nEmployee Code : ";
                cout<<emp[i].empcode;
                cout<<"\nEmployee Name : ";
                cout<<emp[i].empname;
                cout<<"\nEmployee Salary :";
                cout<<emp[i].salary;
                cout<<"\nEmployee Date of Joining :";
                cout<<emp[i].doj.dd<<"/"<<emp[i].doj.mm<<"/"<<emp[i].doj.yy;
        }
      }
      /*Enter no. of records :3
      Enter Employee Code : 1
      Enter Employee Name : shalini
      Enter Employee Salary :4000
      Enter Employee Date of Joining :19 07 2002
      Enter Employee Code : 2
      Enter Employee Name : amar
      Enter Employee Salary :7000
      Enter Employee Date of Joining :19 06 1996
      Enter Employee Code : 3
      Enter Employee Name : abhi
      Enter Employee Salary :8000
      Enter Employee Date of Joining :07 03 2000
      Employee Code : 1
      Employee Name : shalini
      Employee Salary :4000
      Employee Date of Joining :19/7/2002
      Employee Code : 2
      Employee Name : amar
      Employee Salary :7000
      Employee Date of Joining :19/6/1996
      Employee Code : 3
      Employee Name : abhi
      Employee Salary :8000
      Employee Date of Joining :7/3/2000*/
```

7. a) Write a program to generate Fibonacci series.

```
#include<iostream.h>
void main()
{
      int prev=1,pre=1,next;
      int n;
```

```
        cout<<"\nEnter no.of terms :";
        cin>>n;
        cout<<prev<<" "<<pre;
        n=n-2;
        while(n>0)
        {
          next=prev+pre;
          cout<<" "<<next;
          prev=pre;
          pre=next;
          n--;
        }
}
/*Enter no.of terms :10
1 1 2 3 5 8 13 21 34 55*/
```

b) Write a program to check whether a given number is prime number or not and display respectively?

```
#include<stdio.h>
main()
{
int i,n;
printf("\n Enter n :");
scanf("%d",&n);
for(i=2;i<n;i++)
{
if(n%i ==0)
  break;
}
if(i==n)
printf("%d is a prime number",n);
else
printf("%d is not a prime number",n);
}
/* Enter n :11
11 is a prime number
 Enter n :15
15 is not a prime number*/
```

c) Explain difference between declaration and definition?

Already explained above.

d) Explain the concept of constructor and destructor in C++.

Already explained in the Constructor section