SOLUTION OF OOPS

Assignment Solution of Object Oriented Programming

Compiled By: Er. Ashish Kr. Jha
Computer Science & Engineering Department

Chapter

Introduction, Class and Object

1) What is Object Oriented Programming? Differentiate procedure oriented and object oriented programming language. OR

List out characteristics of POP and OOP.

Object Oriented Programming is programming paradigm that represents concepts as objects that has data fields and associated procedures known as methods.

Procedure Oriented Programming (POP)	Object Oriented Programming (OOP)
1. Emphasis is on doing things not on data, means it is function driven	1. Emphasis is on data rather than procedure, means object driven
2. Main focus is on the function and procedures that operate on data	2. Main focus is on the data that is being operated
3. Top Down approach in program design	3. Bottom Up approach in program design
4. Large programs are divided into smaller programs known as functions	4. Large programs are divided into classes and objects
5. Most of the functions share global data	5. Data is tied together with function in the data structure
6. Data moves openly in the system from one function to another function	6. Data is hidden and cannot be accessed by external functions
7. Adding of data and function is difficult	7. Adding of data and function is easy
8. We cannot declare namespace directly	8. We can use name space directly, Ex: using namespace std;
9. Concepts like inheritance, polymorphism, data encapsulation, abstraction, access specifiers are not available.	9. Concepts like inheritance, polymorphism, data encapsulation, abstraction, access specifiers are available and can be used easily
10. ***Example***s: C, Fortran, Pascal, etc…	10. ***Example***s: C++, Java, C#, etc…

2) Explain Basic Concepts of OOP. OR

Explain Following terms.

Various concepts present in OOP to make it more powerful, secure, reliable and easy.

Object

- An object is an instance of a class.
- An object means anything from real world like as person, computer, bench etc...
- Every object has at least one unique identity.
- An object is a component of a program that knows how to interact with other pieces of the program.
- An object is the variable of the type class.
- For ***Example***, If water is class then river is object. **Class**
- A class is a template that specifies the attributes and behavior of things or objects.
- A class is a blueprint or prototype from which objects are created.
- A class is the implementation of an abstract data type (ADT).
- It defines attributes and methods.

Example:

```
class employee                          // Class
{
        char name[10];          // Data member int id;     // Data member
    public:
        void getdata()          // Member function {
            cout<<" enter name and id of employee:";
            cin>>name>>id;
        }
}a;                                     // Object Declaration
```

- In above ***Example*** class employee is created and 'a' is object of this class.
- Object declaration can be also done in main() function as follows:

```
int main() {
    employee a; }
```

Data Abstraction

- Just represent essential features without including the background details.
- They encapsulate all the 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 sometimes called 'methods' or 'member functions'.
- It is used to implement in class to provide data security.

Encapsulation

- Wrapping up of a data and functions into single unit is known as encapsulation.
- In C++ the data is not accessible to the outside world.
- Only those functions can access it which is wrapped together within single unit.

Inheritance

- Inheritance is the process, by which class can acquire the properties and methods of another class.
- The mechanism of deriving a new class from an old class is called inheritance.
- The new class is called **derived class** and old class is called **base class.**
- The derived class may have all the features of the base class.
- Programmer can add new features to the derived class.
- For ***Example***, Student is a base class and Result is derived class.

Polymorphism

- A Greek word Polymorphism means the ability to take more than one form.
- Polymorphism allows a single name to be used for more than one related purpose.
- The concept of polymorphism is characterized by the idea of 'one interface, multiple methods',
- That means using a generic interface for a group of related activities.
- The advantage of polymorphism is that it helps to reduce complexity by allowing one interface to specify a general class of action'. It is the compiler's job to select the specific action as it applies to each situation.
- It means ability of operators and functions to act differently in different situations.

  Example:

```
int total(int, int);
int total(int, int, float);
```

Static Binding

- Static Binding defines the properties of the variables at compile time. Therefore they can't be changed.

Dynamic Binding

- Dynamic Binding means linking of procedure call to the code to be executed in response to the call.
- It is also known as **late binding**, because It will not bind the code until the time of call at run time. In other words properties of the variables are determined at runtimes.
- It is associated with polymorphism and inheritance. **Message Passing**
- A program contains set of object that communicates with each other.
- Basic steps to communicate
  - **Creating classes** that define objects and their behavior.
  - **Creating objects** from class definition
  - **Establishing communication** among objects.
  - Message passing involves the object name, function name and the information to be sent.
  - ***Example***: employee.salary(name);
- In above statement employee is an object. salary is message, and name is information to be sent.

3) List out benefits of OOP.

- We can eliminate redundant code though **inheritance.**
- Saving development time and cost by using existing module.
- Build secure program by **data hiding.**
- It is easy to partition the work in a project based on object.
- Data centered design approach captures more details of a programming model.
- It can be easily upgraded from small to large system.
- It is easy to map objects in the problem domain to those in the program.
- Through message passing interface between objects makes simpler description with external system.
- Software complexity can be easily managed.

4) List out Applications of OOP.

- Applications of OOP technology has gained importance in almost all areas of computing including real-time business systems.
- Some of the ***Example***s are as follows:
  1. Real-time systems
  2. Simulation and modeling
  3. Object oriented database
  4. Artificial intelligence and expert systems
  5. Neural networks and parallel programming
  6. Decision support and office automation
  7. CIM/CAM/CAD systems.
  8. Distributed/Parallel/Heterogeneous computing
  9. Data warehouse and Data mining/Web mining

1) What is C++? Explain Structure of C++ Program.

- C++ is an object oriented programming language.
- It is a superset of C language and also called as extended version of C language.
- It was developed by Bjarne Stroustrup at AT&T Bell lab in New Jersey, USA in the early 1980's.
- Structure of C++ program is as follow.

• Include Files
• Class Declaration or Definition
• Member functions definitions
• Main function

- In any program first write header files like as iostream.h, conio.h, etc..as per requirement of program.
- After header file write class declaration or definition as per your planning.
- After class, define all member functions which are not define but declare inside the class.
- In last write main function without main function program execution is not possible.

Example:

```
#include <iostream> //Header File using namespace std;

class trial      //Class {
        int a; public:
        void getdata() {
                a=10; }
        void putdata(); };
void trial::putdata()    //Member Function outside class definition {
        cout<<"The value of a = "<<a; }
int main()      //Main Function {
        trial T; T.getdata(); T.putdata();
}
```

Output:

The value of a = 10

2) Explain following terms.

Namespace:

- It defines a scope for the identifiers that are used in a program.
- For using identifiers defined in namespace using directive is used as follows: using namespace std;
- **std** is the namespace where ANSI C++ standard class libraries are defined.
- All ANSI C++ programs must include this directive.
- This will bring all the identifiers defined in std to the current global scope.

Keywords:

- They are explicitly reserved identifiers and cannot be used as names for the program variables or other user-defined program elements. Ex: int, class, void etc.

Identifiers:

- They refer to the names of variables, functions, arrays, classes etc., created by the programmer.
- Each language has its own rules for naming these identifiers.
- Following are common rules for both C and C++:
    - Only alphabetic characters, digits and underscores are permitted.
    - The name cannot start with a digit.
    - Uppercase and lowercase letters are distinct.
    - A declared keyword cannot be used as a variable name.

Constants:

- Like variables, constants are data storage locations. But variables can vary, constants do not change.
- You must initialize a constant when you create it, and you can not assign new value later, after constant is initialized.
  - Defining constant **using #define:**

- #define is a preprocessor directive that declares symbolic constant.
  - ***Example*** syntax:

    #define PI 3.14
- Every time the preprocessor sees the word PI, it puts 3.14 in the text.
  - ***Example*****:** #include<iostream> using namespace std; #define PI 3.14

```
int main() {
        int r,area;  cout<<"Enter Radius :"; cin>>r;
        area=PI*r*r;
        cout<<"Area of Circle = "<<area; return 0;
}
```

Output:

Enter Radius :5

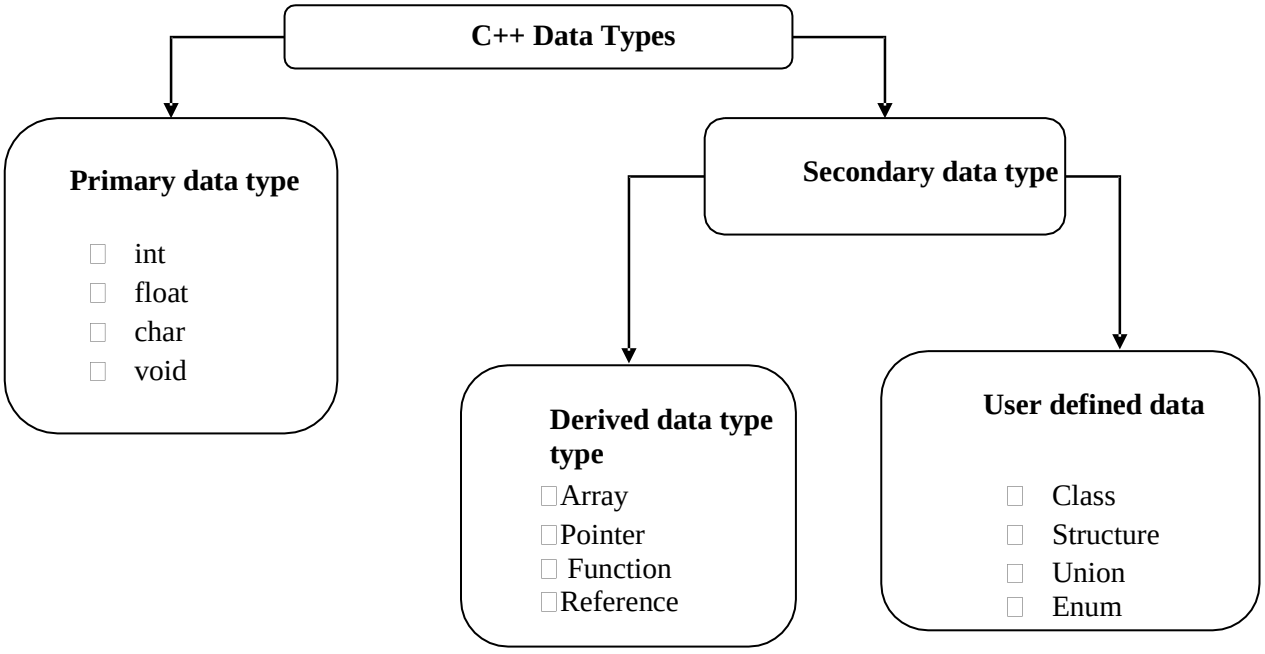
Area of Circle = 78.5

- Defining constant **using const keyword:**
- **'const'** keyword is used to declare constant variable of any type. We cannot change its value during execution of program.
  - Syntax: const Datatype Variable_Name=value; Ex: const int a=2;
  - Now **'a'** is a integer type constant.

3) Explain various Data types used in C++.

- C++ provides following data types.
- We can divide data types into three parts

1. Primary data type
2. Derived data type
3. User defined data type



Primary/Inbuilt Data types:

- The primary data type of C++ is as follow.

	Size (bytes)	Range
char	1	-128 to 127
unsigned char	1	0 to 255
short or int	2	-32,768 to 32,767
unsigned int	2	0 to 65535
long	4	-2147483648 to 2147483647
unsigned long	4	0 to 4294967295
float	4	3.4e-38 to 3.4e+308
double	8	1.7e-308 to 1.7e+308
long double	10	3.4e-4932 to 1.1e+4932

Derived Data types:

- Following derived data types.

1. Arrays
2. Function
3. Pointers

- We cannot use the derived data type without use of primary data type.
- **Array**: An array is a fixed-size sequenced collection of elements of the same data type.
- **Pointer**: Pointer is a special variable which contains address of another variable.
- **Function**: A Group of statements combined in one block for some special purpose.

User Defined Data types:

- We have following type of user defined data type in C++ language.
- The user defined data type is defined by programmer as per his/her requirement.
- **Structure:** Structure is a collection of logically related data items of different data types grouped together and known by a single name.
- **Union:** Union is like a structure, except that each element shares the common memory.
- **Class:** A class is a template that specifies the fields and methods of things or objects. A class is a prototype from which objects are created.
- **enum:** Enum is a user-defined type consisting of a set of named constants called enumerator.
- In other words enum is also used to assign numeric constants to strings
    - Syntax of enumeration: enum enum_tag {list of variables};
    - ***Example*** of enumeration: enum day-of-week {mon=1,tue,wed,thu,fri,sat,sun};

4) Describe various operators used in C++.

An operator is a symbol that tells the compiler to perform certain mathematical or logical operation.

1. Arithmetic Operators

Arithmetic operators are used for mathematical calculation. C++ supports following arithmetic operators

+	Addition or unary plus

-	Subtraction or unary minus
*	Multiplication
/	Division
%	Modulo division

2. Relational Operators

Relational operators are used to compare two numbers and taking decisions based on their relation. Relational expressions are used in decision statements such as if, for, while, etc…

<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to

==	is equal to
!=	is not equal to

3. Logical Operators

Logical operators are used to test more than one condition and make decisions

&&	logical AND (Both non zero then true, either is zero then false)
||	logical OR (Both zero then false, either is non zero then true)
!	logical NOT (non zero then false, zero then true)

4. Assignment Operators

Assignment operators are used to assign the result of an expression to a variable. C++ also supports shorthand assignment operators which simplify operation with assignment

=	Assigns value of right side to left side
+=	a += 1 is same as a = a + 1
-=	a -= 1 is same as a = a - 1
*=	a *= 1 is same as a = a * 1
/=	a /= 1 is same as a = a / 1
%=	a %= 1 is same as a = a % 1

5. Increment and Decrement Operators

These are special operators in C++ which are generally not found in other languages.

++	Increments value by 1. a++ is postfix, the expression is evaluated first and then the value is incremented. Ex. a=10; b=a++; after this statement, a= 11, b = 10. ++a is prefix, the value is incremented first and then the expression is evaluated.
--	Decrements value by 1. a-- is postfix, the expression is evaluated first and then the value is decremented. Ex. a=10; b=a--; after this statement, a= 9, b = 10. --a is prefix, the value is decremented first and then the expression is evaluated.

6. Conditional Operator

A ternary operator is known as Conditional Operator. *exp1?exp2:exp3* if exp1 is true then execute exp2 otherwise exp3

Ex: x = (a>b)?a:b; which is same as if(a>b)
x=a; else
x=b;

7. Bitwise Operators

Bitwise operators are used to perform operation bit by bit. Bitwise operators may not be applied to float or double.

&	bitwise AND
|	bitwise OR
^	bitwise exclusive OR
<<	shift left (shift left means multiply by 2)
>>	shift right (shift right means divide by 2)

8. Special Operators

&	Address operator, it is used to determine address of the variable.
*	Pointer operator, it is used to declare pointer variable and to get value from it.
,	Comma operator. It is used to link the related expressions together.
sizeof	It returns the number of bytes the operand occupies.
.	member selection operator, used in structure.
->	member selection operator, used in pointer to structure.

9. Extraction operator (>>)

Extraction operator (>>) is used with cin to input data from keyboard.

10. Insertion operator (<<)

Insertion operator (<<) is used with cout to output data from keyboard.

11. Scope resolution operator (::)

Scope resolution operator (::) is used to define the already declared member functions of the class.

5) Explain Memory Management Operators of C++ with *Example*.

- For dynamic memory management, C++ provides two unary operator 'new' and 'delete'.
- An object can be created by using new, and destroy by using delete, as and when required.
- Dynamic allocation of memory using new
- Syntax of **new :**
    - pointer_variable = new data_type;
- Here pointer_variable is a pointer of any data type.
- The new operator allocates sufficient memory to hold a data object.
- The pointer_variable holds the address of the memory space allocated.
- For ***Example***:
- p=new int; q=new float;
- Type of 'p' is integer and type of 'q' is float.
- We can combine declaration and initialization.
    - int *p=new int;
    - float *q=new float;
- We can dynamic allocate space for array, structure and classes by new. int *p=new int[10];
- Allocates a memory space for an array of size 10.
- p[0] will refer location of p[1] and p[1] will refer location of [2] and so on.
- Release memory using delete
- When a data object is no longer needed, it is destroyed to release the memory space for reuse.
- Syntax of delete: delete pointer_variable;
- The pointer_variable is the pointer that points to a data object created with new.
    - For ***Example***: delete p; delete q;
- To free a dynamically allocated array
    - delete [size] pointer_variable; delete [10]p;

6) What is reference variable? What is its major use?

- A reference variable provides an alias (alternative name) for a previously defined variable.
- This mechanism is useful in object oriented programming because it permits the manipulation of objects by reference, and eliminates the copying of object parameters back and forth.
- It is also important to note that references can be created not only for built-in data types but also for user-defined data types.
- Syntax: Data_type & reference_name = variable_name
- For ***Example*** :

```
int a=100;

int &b=a;     //Now both a and b will give same value.
```

7) Explain use of scope resolution operator (::) by giving *Example*.

- The scope resolution operator is used to resolve or extend the scope of variable.
- C++ is block structured language. We know that the same variable name can be used to have different meaning in different block.
- The scope resolution operator will refer value of global variable from anywhere (also from inner block).
- Without scope resolution operator all variable will refer local value.

***Example*:**

```
#include <iostream.h> int m=10;
int main() {
        int m=20; {
                int k=m; int m=30;
```

```
        cout<<"we are in inner block\n"; cout<<"k="<<k<<"\n"; cout<<"m="<<m<<"\n";
        cout<<"::m="<<::m<<"\n";
    }
    cout<<"we are in outer block\n"; cout<<"m="<<m<<"\n"; cout<<"::m="<<::m<<"\n"; return 0;
}
```

Output:

```
we are in inner block k=20
m=30
we are in outer block m=20
::m=20
```

8) Explain member dereferencing operators in short.

C++ provides three pointers to member operators to access the class member through pointer.

Operato	Function
::*	To declare a pointer to a member of a class.
.*	To access a member using object name and a pointer to that member.
->*	To access a member using a pointer to the object and a pointer to that member.

9) Explain implicit and explicit type conversion in short.

Implicit Conversion:

- Wheneer data types are mixed in an expression, C++ performs the conversions automatically. This process is known as implicit or automatic conversion.
- ***Example***: m= 5+2.5;
- For a binary operator, if the operands type differs, the compiler converts one of them to match with the other.
- Using the rule that the smaller type is converted to wider type.
- For ***Example*** if one operand is integer and another is float then integer is converted to float because float is wider than integer.
- In above ***Example*** answer means m, will be in float.

Explicit conversion:

- C++ permits explicit type conversion of variables or expressions using the type cast operator.
- Syntax: type_name (expression)
- ***Example***: average = sum/float(i);
- Alternatively we can use typedef to create an identifier of the required type and use it in the functional notation.

Example:

```
typedef  int * int_ptr; p = int_ptr(q);
```

***Example*:**

```
#include<iostream>
using namespace std;

int main() {
        int intvar=5;
        float floatvar=3.5;

        cout<<"intvar = "<<intvar; cout<<"\nfloatvar = "<<floatvar; cout<<"\nfloat(intvar) =
        "<<float(intvar); cout<<"\nint(floatvar) = "<<int(floatvar);
}
```

Output:

intvar = 5 floatvar = 3.5 float(intvar) = 5 int(floatvar) = 3

1) Explain Call by Value and Call by Reference with appropriate *Example*. OR

Write a program to interchange (swap) value of two variables.

Call By Value

- In call by value, value of variable is passed during function call.
- And copy this value in another variable at function definition.
- In call by value the original value in calling function will never change after execution of function.

Example:

```
#include<iostream> using namespace std;

void swp(int a, int b) {
        int temp=a; a=b;  b=temp;
}
int main() {
        int x,y;
        cout<<"enter value of a and b:"; cin>>x>>y;
        swp(x,y);
        cout<<"\nafter swapping a = "<<a<<"and b = "<<b; return 0;
}
```

Output:

```
Enter value of a and b: 4
5
After swapping a = 5 and b = 4
```

Call By Reference

- In call by reference, reference is passed during function call.
- The formal arguments in the called function become aliases to the actual function call.
- In call by reference the original value in calling function will change after execution of function.

Example:

```
#include<iostream> using namespace std;

void swap(int &a, int &b) // It is the only difference from above program {
        int temp=a; a=b;  b=temp;
}
int main() {
        int a,b;
        cout<<"Enter two numbers:"; cin>>a>>b;
        swap(a,b); cout<<"a="<<a<<"b="<<b; return 0;
}
```

Output:

```
Enter value of a and b: 4
5
After swapping a = 5 and b = 4
```

2) What is inline function? Explain with *Example*.

- The functions can be made inline by adding prefix inline to the function definition.
- An inline function is a function that is expanded in line when it is invoked.
- The complier replaces the function call with the corresponding function code.
- Inline function saves time of calling function, saving registers, pushing arguments onto the stack and returning from function.
- Preprocessor **macros** are popular in C, which has similar kind of advantages mentioned above.
- The major drawback of macros is that they are not really functions.
- Therefore, the usual error checking does not occur during execution of macros.
- We should be careful while using inline function. If function has very few lines of code and simple expressions then only it should be used.
- **Critical situations** for inline function:

  1) If a loop, a switch or a goto exists in function body. 2) If function is not returning any value.

3) If function contains static variables. 4) If function is recursive.

***Example*:**

```
inline int cube(int n) {
        return n*n*n; }
int main() {
        int c;
        c = cube(10); cout<<c;
}
```

- Function call is replaced with expression so c = cube(10); becomes c=10*10*10; at compile time.

3) Explain Function with Default Arguments with appropriate *Example*.

- C++ allows us to call a function without specifying all its arguments.
- In such cases, the function assigns a default value to the parameter.
- Default values are specified when the function is declared.
- We must add default arguments from right to left.
- We cannot provide a default value to a particular argument in the middle of an argument list.
- Default arguments are useful in situations where some arguments always have the same value. For ***Example***, passing marks.
- Legal and illegal default arguments
  - void f(int a, int b, int c=0);
  - void f(int a, int b=0, int c=0);
  - void f(int a=0, int b, int c=0);
  - void f(int a=0, int b, int c);
  - void f(int a=0, int b=0, int c=0);

***Example*:**

```
#include <iostream> using namespace std;
    void f(int a=0, int b=0)
    {
            cout<< "a= " << a << ", b= " << b<<endl; }
    int main() {
            f();  f(10); f(10, 99);
            return 0; }
```

Output:

a=0 , b=0 a=10 ,b=0 a=10, b=99

4) What is friend function? Explain with *Example*.

- A friend function is a function which is declared using **friend** keyword.
- It is not a member of the class but it has access to the private and protected members of the class.
- It is not in the scope of the class to which it has been declared as friend.
- It cannot access the member names directly.
- It can be declared either in public or private part of the class.
- It is not a member of the class so it cannot be called using the object.
- Usually, it has the objects as arguments.
- It is normal external function which is given special access privileges.

Syntax:

```
class ABC {
        public: ……
                friend void xyz(void); // declaration ……
};
```

Example: #include<iostream> using namespace std; class numbers

{

```
        int num1, num2; public:
                void setdata(int a, int b); friend int add(numbers N);
};
void numbers :: setdata(int a, int b) {
        num1=a; num2=b;
}
int add(numbers N) {
        return (N.num1+N.num2); }
int main() {
        numbers N1; N1.setdata(10,20); cout<<"Sum = "<<add(N1);
}
```

Output: Sum = 30

- add is a friend function of the class numbers so it can access all members of the class(private, public and protected).
- Member functions of one class can be made friend function of another class, like…

```
class X {
        …… int f();
};  class Y {
            …… friend int X :: f();
};
```

- The function f is a member of class X and a friend of class Y.
- We can declare all the member functions of one class as the friend functions of another class.
- In such cases, the class is called a friend class, like class X is the friend class of class Z

```
class Z {
        …… friend class X;
        …….
};
```

5) Explain function overloading with *Example*.

- Function overloading is compile time polymorphism.
- Function overloading is the practice of declaring the same function with different signatures.
- The same function name will be used with different number of parameters and parameters of different type.
- Overloading of functions with different return types is not allowed.
- Compiler identifies which function should be called out of many using the type and number of arguments.
- A function is overloaded when same name is given to different functions.
- However, the two functions with the same name must differ in at least one of the following,
  a) The number of parameters
  b) The data type of parameters
  c) The order of appearance

Example:

```
#include <iostream> using namespace std; class Math
{
        public:
        void Add(int num1, int num2) //Function 1 {
                cout<<num1 + num2 <<endl; }
        void Add(float num1, float num2) //Function 2 {
                cout<<num1 + num2 <<endl; }
```

```
        void Add(int num1, int num2, int num3) //Funciton 3 {
                cout<<num1 + num2 + num3 <<endl; }
};
int main() {
        Math m;
        m.Add(10,20); \\ Calls function 1 m.Add(10.15, 25.70); \\ Calls function 2 m.Add(1,2,3);
        \\ Calls function 3
}
```

Output: 30 35.85 6

6) Explain operator overloading with *Example*.

- Operator overloading is compile time polymorphism.
- The operator overloading provides mechanism to perform operations on user defined data type.
- We can give special meaning to any operators in which program it is implemented.
- Rules for operator overloading

1. Only existing operator can be overloaded.
2. The overloaded operator must have at least one operand that is user defined type.
3. We cannot change the basic meaning and syntax of an operator.
4. We cannot use **friend** function to overload certain operators. However member function can be used to overload them.
5. Unary operators, overloaded by means of a member function, take no explicit argument s and return no explicit value, but, those overloaded by means of a friend function, take one reference argument.
6. Binary operators overloaded through a member function take one explicit argument and those which are overloaded through a friend function take two explicit arguments.
7. When using binary operators overloaded through a member function, the left hand operand must be an object of the relevant class.
8. We cannot overload following operators.

Operator	Name
. and .*	Class member access operator
::	Scope Resolution Operator
sizeof()	Size Operator
?:	Conditional Operator

- ***Example* for Unary Operator Overloading**

```
#include <iostream> using namespace std;

class sample {
        int a,b; public:
        void getdata()
```

```
        {
                a=10; b=20'

        }
        void operator -()    //Unary Member Function {
                a = a - 5; b = b - 5;
        }
       void disp() {
                cout<<"\nThe value of a="<<a; cout<<"\nThe value of b="<<b;

        } };

int main() {
        sample S; S.getdata();
       -S;                    //Call Unary Member Function S.disp();
        getch(); return 0;
}
```

- ***Example* for Unary Operator Overloading using Friend function.**

```
#include <iostream> using namespace std;

class sample {
        int a,b; public:
        void getdata() {
                a=10; b=20;

        }
        friend sample operator +(sample A)    //Unary Friend Function {
                A.a = A.a + 5; A.b = A.b + 5; return A;
        }
        void disp() {
                cout<<"\nThe value of a="<<a; cout<<"\nThe value of b="<<b;
        } };
int main() {
        sample S; S.getdata();
        S=+S;                    //Call Unary Friend Function S.disp();
        getch(); return 0;
}
```

- ***Example* for Binary Operator Overloading**

```
#include <iostream> using namespace std; class complex
{
        float real, imag; public:
        complex(float _real, float _imag) // constructor {
                real = _real; imag = _imag;
        }
        void disp() {
                cout<<"The value of real="<<real; cout<<"The value of imag="<<imag;
        }
        void operator +(complex c) //Binary Member function {
               real = real + c.real; imag = imag + c.imag;
        } };
int main() {
      complex x(4,4); complex y(6,6);
        x + y; // Call Binary Member Function x.disp();
          getch(); return 0;
}
```

- ***Example* of Binary operator overloading using friend function**

```
#include <iostream> using namespace std; class complex
{
        float real, imag; public:
        complex(float _real, float _imag) // constructor {
                real = _real; imag = _imag;
        }
        void disp() {
                cout<<"The value of real="<<real; cout<<"The value of imag="<<imag;
        }
        friend complex operator +(complex c, complex d) //Binary Friend function {
                d.real = d.real + c.real; d.imag = d.imag + c.imag; return d;
        } };
int main() {
      complex x(4,4); complex y(6,6);
        complex z = x + y; // Call Binary Friend Function z.disp();
}
```

1) Explain basic concept of class.

- A class is a template that specifies the attributes and behavior of things or objects.
- A class is a blueprint or prototype from which objects are created.
- A class is the implementation of an abstract data type (ADT).
- It defines attributes and methods.

Object declaration:

- In following ***Example*** class employee is created and 'a' is object of this class.

```
class item

{

        // data members and member functions

}a;
```

- In above syntax class name is item, and a is object of that class
- Object declaration can be also done in main() function as follows:

```
int main() {
    item a; }
```

Accessing class member:

- Private members of the class can only accessed by the members with in that class.
- Public members of the class can be accessed outside the class also.
- For accessing class member outside class can be done by using dot operator and object of that class using following syntax,

```
object-name.function-name(actual-arguments);
```

- Member functions of the class can be declared outside class definition also as follows,
- **Syntax:**

```
class item {
public:
        void getdata(); };
void item::getdata();
```

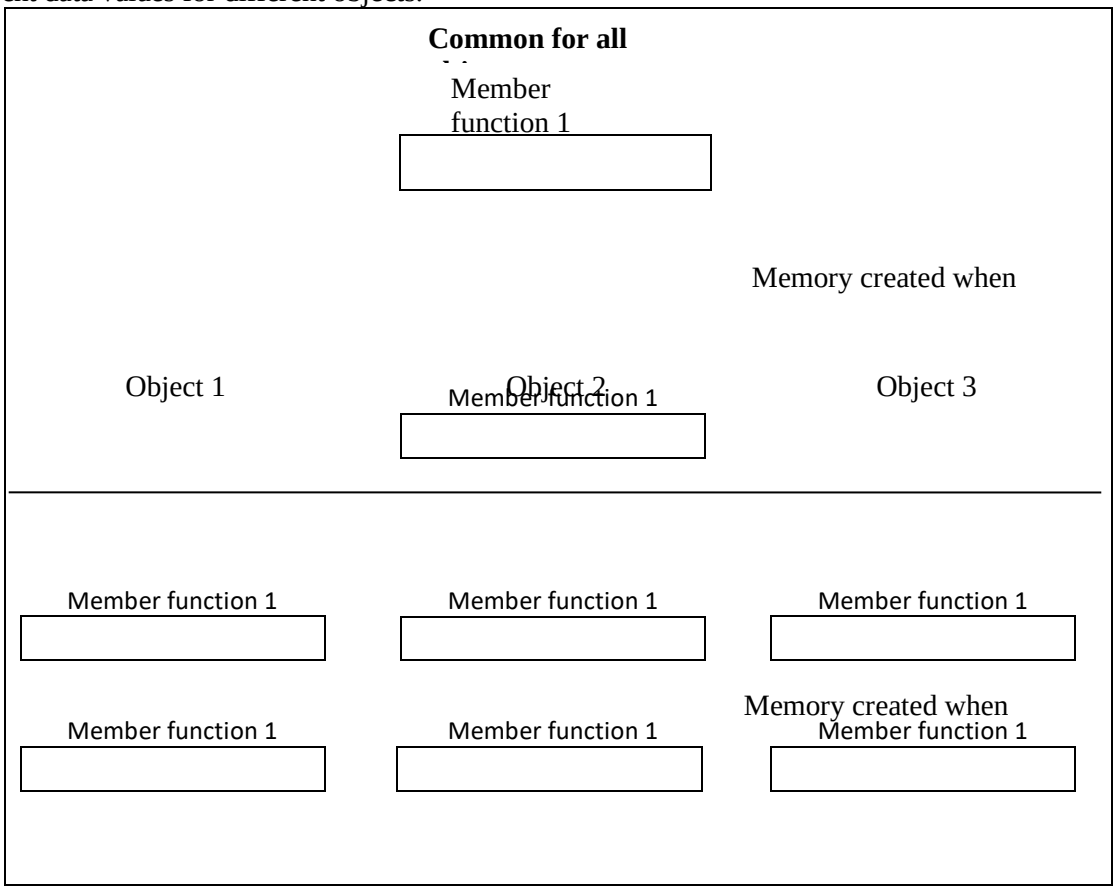
***Example*:** #include<iostream> using namespace std;

```
class employee                          // class {
        char name[10]; // data member int id;       // data member
    public:
        void getdata(); // prototype declaration
```

```
                    void putdata(); // prototype declaration };
void employee::getdata() // member function {
        cout<<"Enter name and id of employee: "; cin>>name>>id;
}

void employee::putdata() // member function {
        cout<<"Display name and id of employee: "; cout<<name<<id;
}
int main()
{
        employee x;
       x.getdata(); x.putdata();
}
```

2) Explain memory allocation of objects with *Example*.

- The member functions are created and placed in the memory space only once when they are defined as part of a class specification.
- No separate space is allocated for member functions when the objects are created.
- Only space for member variable is allocated separately for each object.
- A separate memory location for the objects happens, because the member variables will hold different data values for different objects.



***Example*:**

```
class item {
        public:
        int id, cost; void getdata();
};
int main() {
        item x,y,z; }
```

- In above ***Example*** each object x, y and z has separate space for both id and cost. Means value of id and cost can be different for each object.
- But no default space is allocated to function when object is declared.
- Required separate space is allocated to function during calling of that function.

3) Explain Public and Private Access modifier (specifier) for C++ classes.

Public:

- Public members of the class are accessible by any program from anywhere.
- There are no restrictions for accessing public members of a class.
- Class members that allow manipulating or accessing the class data are made public.

Private:

- Private members of the class can be accessed within the class and from member functions of the class.
- They cannot be accessed outside the class or from other programs, not even from inherited class.
- Encapsulation is possible due to the private access modifier.
- If one tries to access the private members outside the class then it results in a compile time error.
- If any other access modifier is not specified then member default acts as Private member.

***Example*:** #include<iostream> using namespace std;

```
class ABC {
        public:
                int a;

        private: int b;

};
```

```
int main() {
        ABC x; x.a=5;
        cout<<"Value of public variable = "<<x.a; }
```

Output: 5

- In above program b is private member of class ABC.
- So we cannot use it with object of ABC outside the class, Means we cannot use x.b in main as x.a shown above.

4) Explain Static data members and static member functions with *Example*.

Static data members:

- Data members of the class which are shared by all objects are known as static data members.
- **Only one copy** of a static variable is maintained by the class and it is common for all objects.
- Static members are declared inside the class and defined outside the class.
- It is initialized to zero when the first object of its class is created. No other initialization is permitted.
- It is visible only within the class but its lifetime is the entire program.
- Static members are generally used to maintain values common to the entire class.

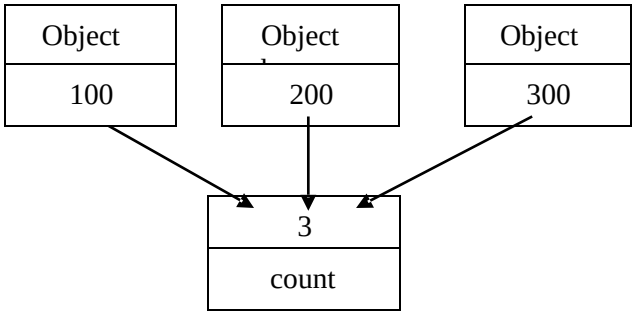
Static member functions:

- Static member functions are associated with a class, not with any object.
- They can be invoked using class name, not object.
- They can access only static members of the class.
- They cannot be virtual.
- They cannot be declared as constant or volatile.
- A static member function can be called, even when a class is not instantiated.
- There cannot be static and non-static version of the same function.
- A static member function does not have this pointer.

***Example*:**

```
#include<iostream> using namespace std;

class item {
        int number;
        static int count;                    // static variable declaration public:
                void getdata(int a) {
                        number = a; count++;
                }
```

```
                static void getcount() // the only difference from above program {
                        cout<<"value of count: "<<count; }
};
int item :: count;      // static variable definition

int main() {
        item a,b,c;
        a.getdata(100);
        b.getdata(200);
        c.getdata(300);
        a.getcount();
        b.getcount();
        c.getcount();
        getch();
        return 0; }
```



Output:
value of count:3 value of count:3 value of count:3

5) What is constructor? List out characteristics of constructors.

- A constructor is a "special" member function which initializes the objects of class.

Properties of constructor:

- Constructor is invoked automatically whenever an object of class is created.
- Constructor name must be same as class name.
- Constructors should be declared in the public section because private constructor cannot be invoked outside the class so they are useless.
- Constructors do not have return types and they cannot return values, not even void.
- Constructors cannot be inherited, even though a derived class can call the base class constructor.
- Constructors cannot be virtual.
- An object with a constructor cannot be used as a member of a union.
- They make implicit calls to the operators **new** and **delete** when memory allocation is required.

6) Explain types of constructor with *Example*.

There are mainly **three types** of constructors as follows:

1. **Default constructor:**
   - **Default constructor** is the one which invokes by default when object of the class is created.
   - It is generally used to initialize the value of the data members.
   - It is also called no argument constructor.

***Example*:**

```
class integar
{
    int m,n; public:
            integer() // Default constructor {
                m=n=0; }
};
```

2. Parameterized constructor

- Constructors that can take arguments are called **parameterized constructors.**
- Sometimes it is necessary to initialize the various data elements of different objects with different values when they are created.
- We can achieve this objective by passing arguments to the constructor function when the objects are created.

***Example*:**

```
class integer {
    int m,n; public:
    integer(int x,int y) // Parameterized constructor {
            m =x; n=y;
    } };
```

3. Copy constructor

- A **copy constructor** is used to declare and initialize an object from another object.
- For ***Example***, integer(integer &i); **OR** integer I2(I1);
- Constructor which accepts a reference to its own class as a parameter is called copy constructor.

***Example*:**

```
class integer {
      int m, n;

      public:
      integer(rectangle &x) // Copy constructor {
          m = x.m; n = x.n;
      } };
```

***Example*:**

```
#include<iostream> using namespace std;

class rectangle {
      int length, width;

      public:
      rectangle() // Default constructor {
            length=0; width=0;
      }
      rectangle(int _length, int _width) // Parameterized constructor {
            length = _length; width = _width;
      }
      rectangle(rectangle &_r) // Copy constructor {
            length = _r.length; width = _r.width;
      } ……
```

```
        // other functions for reading, writing and processing can be written here
        …… };

int main() {
        rectangle r1; // Invokes default constructor
        rectangle r2(10,10); // Invokes parameterized constructor rectangle r3(r2); // Invokes copy
        constructor
}
```

7) Explain destructor with *Example*.

- Destructor is used to destroy the objects that have been created by a constructor.
- Destructor is a member function whose name must be same as class name but is preceded by a tilde (~).
- Destructor never takes any argument nor it returns any value nor it has return type.
- Destructor is invoked automatically by the complier upon exit from the program.
- Destructor should be declared in the public section

***Example*:**

```
#include<iostream.h> using namespace std;

class rectangle {
        int length, width; public:
        rectangle() //Constructor {
                length=0; width=0;
        }
        ~rectangle() //Destructor {

        }
        // other functions for reading, writing and processing can be written here
};
int main()
{
        rectangle x; // default constructor is called for this object
        }
```

8) Explain use of objects as function arguments with *Example*.

Like any other data type function may be used as a function argument. It can be done in following two ways:

1. A copy of the entire object is passed to the function
   - This method is call pass by value.
   - Since copy of the object is passed to the function, any changes made to the object inside the function do not affect the object used to call the function.
2. Only the address of the object is transferred to the function.
   - This method is called pass-by-reference.
   - When an address of the object is passed, 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.
   - This method is **more efficient** because it requires passing only addresses of the object, not an entire object.
   - ***Example*:**

```
#include<iostream> using namespace std; clas time
{
        int hours; int minutes; public:
                void gettime(int h, int m) {
                        hours=h; minutes=m;
                }
                void puttime(void) {
                        cout<<hours<<" hours and";



                        cout<<minutes<<"minutes"<<"\n"; }
                void sum(time, time); //declaration with objects as arguments };

void time::sum(time t1, time t2) //t1,t2 are objects {
        minutes = t1.minutes + t2.minutes; hours = minutes/60;
        minutes = minutes % 60;
        hours =hours + t1.hours + t2.hours; }

int main() {
        time t1, t2, t3; t1.gettime(2,45); //get t1 t2.gettime(3,30); //get t2 t3.sum(t1,t2); //t3 = t1 +
        t2
        cout<<"t1 = "; t1.puttime(); //display t1 cout<<"t2 = "; t2.puttime(); //display t2
        cout<<"t3 = "; t3.puttime(); //display t3

        return 0; }
```

9) Explain operator overloading with *Example*.

- Operator overloading is compile time polymorphism.
- The operator overloading provides mechanism to perform operations on user defined data type.
- We can give special meaning to any operators in which program it is implemented.
- Rules for operator overloading

1. Only existing operator can be overloaded.
2. The overloaded operator must have at least one operand that is user defined type.
3. We cannot change the basic meaning and syntax of an operator.
4. We cannot use **friend** function to overload certain operators.
5. Unary operators, overloaded by means of a member function, take no explicit arguments and return no explicit value, But, those overloaded by means of a friend function, take one reference argument.
6. Binary operators overloaded through a member function take one explicit argument and those which are overloaded through a friend function take two explicit arguments.
7. When using binary operators overloaded through a member function, the left hand operand must be an object of the relevant class.
8. We cannot overload following operators.

Operator	Name
. and .*	Class member access operator
::	Scope Resolution Operator
sizeof()	Size Operator
?:	Conditional Operator

- ***Example* for Unary Operator Overloading**

```
#include <iostream> using namespace std;

class sample {
        int a,b;

        public:
        void getdata() {
                a=10; b=20;

        }
        void operator -()     //Unary Member Function {
                a = a - 5; b = b - 5;
        }
        void disp() {
                cout<<"\nThe value of a="<<a; cout<<"\nThe value of b="<<b;
        } };

int main() {
        sample S; S.getdata();
        -S;                         //Call Unary Member Function S.disp();
        getch(); return 0;
}
```

Output:
The value of a=5 The value of b=15

- ***Example* for Unary Operator Overloading using Friend function.**

```
#include <iostream> using namespace std;

class sample {
        int a,b;

        public:
        void getdata() {
                a=10; b=20;

        }
        friend sample operator +(sample A)     //Unary Friend Function {
                A.a = A.a + 5; A.b = A.b + 5; return A;



        }
        void disp() {
                cout<<"\nThe value of a="<<a; cout<<"\nThe value of b="<<b;
        } };
int main() {
        sample S; S.getdata();
        S=+S;                       //Call Unary Friend Function S.disp();
        return 0; }
```

Output:
The value of a=15 The value of b=25

- ***Example* for Binary Operator Overloading**

```
#include <iostream> using namespace std;

class complex {
        float real, imag; public:
        complex(float _real, float _imag) // constructor {
                real = _real; imag = _imag;
        }
        void disp() {
                cout<<"The value of real="<<real; cout<<"The value of imag="<<imag;
        }
        void operator +(complex c) //Binary Member function {
               real = real + c.real; imag = imag + c.imag;
        } };
int main() {
       complex x(4,4); complex y(6,6);
        x + y; // Call Binary Member Function x.disp();
          getch(); return 0;
}
```

Output:
The value of real=10 The value of imag=10

- ***Example* for Unary Operator Overloading using Friend function.**

```
#include <iostream> using namespace std;

class complex {


        float real, imag; public:
        complex(float _real, float _imag) // constructor {
                real = _real; imag = _imag;
        }
        void disp() {
                cout<<"The value of real="<<real; cout<<"The value of imag="<<imag;
        }
        friend complex operator +(complex c, complex d) //Binary Friend function {
                d.real = d.real + c.real; d.imag = d.imag + c.imag; return d;
        } };
 int main() {
       complex x(4,4); complex y(6,6);
        complex z = x + y; // Call Binary Friend Function z.disp();
 }
```

Output:
The value of real=10 The value of imag=10

10) List out various type conversion techniques? Explain basic to class type conversion with *Example*.

- C++ provides mechanism to perform automatic type conversion if all variable are of basic type.
- For user defined data type programmers have to convert it by using constructor or by using casting operator.
- Three type of situation arise in user defined data type conversion.
  1. Basic type to Class type
  2. Class type to Basic type
  3. Class type to Class type

- Now we will see each situation with ***Example***

1. Basic type to Class type

- The basic type to class type conversion is done by using constructor. ***Example*:**

```
#include <iostream> using namespace std;

class sample {
        float a; public: sample(){}
        sample(int x) //Constructor to convert Basic to Class type {
                a=x;



        }
        void disp() {
                cout<<"The value of a="<<a; }
};
int main() {
       int a=10; sample S;
         S=a;          //Basic to class type conversion S.disp();
         return 0; }
```

Output:

The value of a=10

11) Explain type conversion for class to basic type with *Example*.

- The Class type to Basic type conversion is done by using Casing Operator.
- The casting operator function should satisfy the following conditions.
  1. It must be a class member.
  2. It must not specify a return type.
  3. It must not have any arguments.

***Example*:**

```
#include <iostream> using namespace std;

class sample {
        float a; public:
        sample() {
                a=10.23; }
        operator int(); };
sample:: operator int()     //Casting operator function {
        int x; x=a; return x;
}
int main() {
        sample S;
        int y= S;
        // Class to Basic conversion
         cout<<"The value of y="<<y;
        getch(); return 0;
}
```

Output:

The value of y=10

12) Explain type conversion for class type to another class type with *Example*.

- The class to class conversion is done by both constructor and casting operator.
- If conversion take place at source class, then by casting operator.
  operator destination_class_name() //Definition of Casting operator {
  //Create object of destination class //write statement to convert value
  // Return objecct of destination class }
- If conversion take place at destination class, then by constructor.

  Constructor_of_Destination_Class(Source_Class Object_Name) {
  //Statement for conversion }
- We cannot convert at a time in both source and destination class.

***Example*:**

```
#include <iostream> using namespace std;

class invent2; // destination class declared

class invent 1;                 // source class {
                int code;   // item code
                int items;   // no. of items
                float price;   // cost of each item public:
                invent1( int   a, int   b , float   c) {
                        code = a; items = b; price = c;
                }
                void putdata() {
                        cout <<"code:" << code << "\n"; cout <<"items:" << items << "\n";
                        cout <<"value:" << price << "\n";
                }

                int getcode() {return code;}  int getitems(){return items;} float getprice(){return
                price;}

                operator float(){return (items * price);} };

class invent2           // destination class {


                int code; float   value;
        public:
                invent2 () {
                      code = 0; value =0; }
                invent2(float x, float y) {
                     code=x; value =y;
                }
                void putdata () {
                        cout   <<"code:" << code << "\n"; cout <<"value:" << price << "\n";
                }

                invent2(invent1 p) {
                                code = p.getcode();
                                value= p.getitems()*p.getprice(); }
};

int main() {
        invent s1(100,5,140.0); invent2 d1;
        float   total_value; total_value =s1; d1=s1;
        cout<<"product details-invent1 type"<<"\n"; s1.putdata();
```

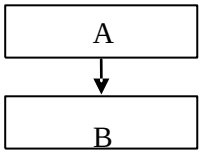
```
cout << ”\nstock value”<<”\n”;
cout << “value =” <<total_value<<”\n\n”;

cout <<”product details-invent2 type”<<”\n”; d1.putdata();
return 0; }
```

1) Explain types of inheritance with *Example*.

- Inheritance is the process, by which class can acquire the properties and methods of another class.
- The mechanism of deriving a new class from an old class is called inheritance.
- The new class is called derived class and old class is called base class.
- The derived class may have all the features of the base class and the programmer can add new features to the derived class.

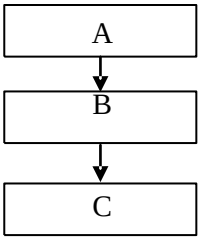
Types of Inheritance:

Single Inheritance



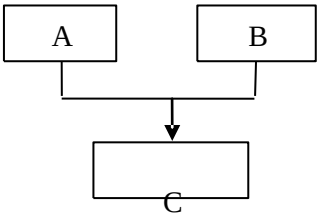
- If a class is derived from a single class then it is called single inheritance.
- Class *B* is derived from class *A*

Multilevel Inheritance



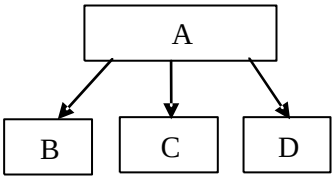
- A class is derived from a class which is derived from another class then it is called multilevel inheritance.
- Here, class *C* is derived from class *B* and class *B* is derived from class
- *A*, so it is called multilevel inheritance.

Multiple Inheritance



If a class is derived from more than one class then it is called *multiple inheritance*.
Here, class *C* is derived from two classes, class *A* and class *B*.

Hierarchical Inheritance



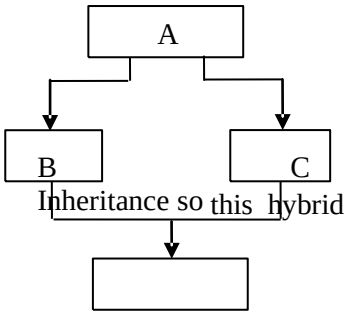
If one or more classes are derived from one class then it is called hierarchical inheritance.

Here, class *B*, class *C* and class *D* are derived from class *A*.

Hybrid Inheritance

It is a combination of any above inheritance types. That is either

multiple or multilevel or hierarchical or any other combination.



Here, class B and class C are derived from class A and class D is derived from class B and class C.
class A, class B and class C is ***Example*** of Hierarchical Inheritance
and class B, class C and class D is ***Example*** of Multiple Inheritance so this hybrid inheritance is combination of Hierarchical and Multiple Inheritance. D

***Example*:**

```
#include<iostream> using namespace std;

class A {
        public:
        void dispA() {
                cout<<"class A method"<<endl; }
};

class B : public A                  //    Single Inheritance - class B is derived from class A {
        public:
        void dispB() {
                cout<<"class B method"; }
};

class C : public B                  // Multilevel Inheritance - class C is derived from class B {
        public:   void dispC() {
                cout<<"class C method"<<endl; }
};

class D {
        public:
        void dispD() {
                cout<<"class D method"<<endl; }
};

class E: public A, public D //Multiple Inheritance:class E is derived from class A {          // and D
        public:
        void dispE()




        {
                cout<<"class E method"; }
};

class F: public B, public C     //Hybrid Inheritance:class F is derived from class B {          // and C
        public:
        void dispF() {
```

```
                    cout<<"class F method"; }
};

int main() {
        A a; B b; C c; D d; E e; F f;
        b.displayA(); f.displayF(); f.displayA();
}
```

Output:

- class A method class A method class D method class C method

    - Class B and class E are derived from class A so it is ***Example*** of Hierarchal Inheritance.
    - Class *F* is derived from class *B* and class *C*, class *B* is derived from class *A* so *displayA()* is not a member of class *F* then also we can access it using object of class *F*.

2) List out visibility of inherited members in various categories of inheritance.

Base class visibility	**Derived class visibility**		
	Public derivation	**Private derivation**	**Protected derivation**
Private	Not inherited	Not inherited	Not inherited
Protected	Protected	Private	Protected
Public	Public	Private	Protected

3) Explain protected access modifier for class members.

Protected:

- This access modifier plays a key role in inheritance.
- Protected members of the class can be accessed within the class and from derived class but cannot be accessed from any other class or program.
- It works like public for derived class and private for other programs

***Example*:**

```
#include<iostream> using namespace std;

class A {
        protected: int a;

        public:
                void getdata() {
                        cout<<"Enter value of a:"; cin>>a;
                } };

class B:public A {
        public:
                void show() {
                        cout<<"Value of a is:"<<a; }
};

int main() {
        B x; x.getdata(); x.show();
}
```

Output:

Enter value of a:5 Value of a is:5

4) What is overriding in C++. Explain with suitable *Example*.

- If base class and derived class have member functions with same name and arguments.
- If you create an object of derived class and write code to access that member function then, the member function in derived class is only invokes.
- Means the member function of derived class overrides the member function of base class. This is called function overriding or method overriding in C++.

***Example*:**

```
#include<iostream> using namespace std;
class A {
        public:
                void display() {
                        cout<<"This is parent class"; }
};
class B:public A {
        public:
                void display() // overrides the display() function of class A {
                        cout<<"\nThis is child class"; }
};

int main() {
        B x;
        x.display(); // method of class B invokes, instead of class A }
```

Output:

This is child class

5) Explain virtual base class with *Example*.

- It is used to prevent the duplication/ambiguity.
- In hybrid inheritance child class has two direct parents which themselves have a common base class.
- So, the child class inherits the grandparent via two seperate paths. it is also called as indirect parent class.
- All the public and protected member of grandparent are inherited twice into child.

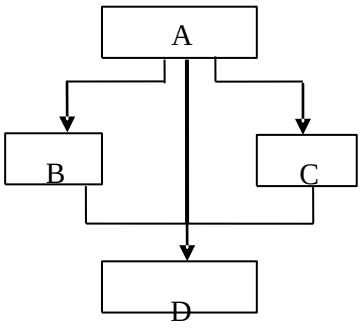


Figure: Multipath Inheritance

We can stop this duplication by making base class virtual.

For ***Example***:

```
class A {
        public: int i;
};

class B : virtual public A {
            public: int j;
};

 class C: virtual public A {
                public: int k;
 };

 class D: public B, public C {
                public: int sum;
 };
```

The keywords virtual and public may be used in either order.

If we use virtual base class, then it will inherit only single copy of member of base class to child class.

6) List out various situations for the execution of base class constructor in inheritance.

Method of Inheritance	Order of Execution
Class B:public A { };	A(); base constructor B(); derived constructor

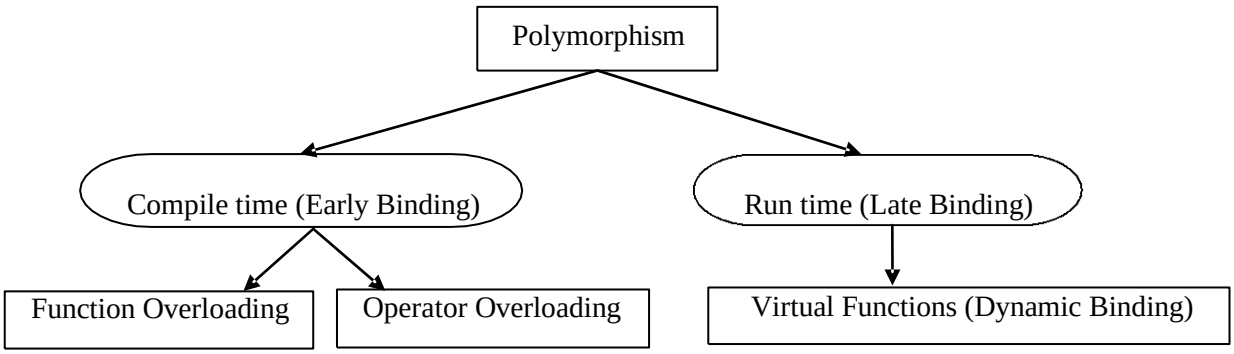
Class A:public B, public C
{
};
	B(); base(first)
	C(); base(second)
	A(); derived

Class A:public B, virtual public C
{
};
	C(); virtual base
	B(); ordinary base
	A(); derived

1) Write a short note on Polymorphism.

- Polymorphism means the ability to take more than one form.
- It allows a single name to be used for more than one related purpose.
- It means ability of operators and functions to act differently in different situations.

Different types of polymorphism are



Compile time:

- Compile time polymorphism is function and operator overloading.

Function Overloading:

- Function overloading is the practice of declaring the same function with different signatures.
- The same function name will be used with different number of parameters and parameters of different type.
- Overloading of functions with different return type is not allowed.

Operator Overloading:

- Operator overloading is the ability to tell the compiler how to perform a certain operation based on its corresponding operator's data type.
- Like + performs addition of two integer numbers, concatenation of two string variables and works totally different when used with objects of time class.

Dynamic Binding (Late Binding):

- Dynamic binding is the linking of a routine or object at runtime based on the conditions at that moment.
- It means that the code associated with a given procedure call is not known until the time of the call.
- At run-time, the code matching the object under current reference will be called.

Virtual Function:

- Virtual function is a member function of a class, whose functionality can be over-ridden in its derived classes.
- The whole function body can be replaced with a new set of implementation in the derived class.
- It is declared as virtual in the base class using the virtual keyword.

2) Explain pointers to objects with *Example*.

- A pointer can point to an object created by a class.

- Object pointers are useful in creating objects at run time.
- We can also use an object pointers to access the public members of an object. For ***Example***,

```
item x;

item *ptr = &x;
```

Here pointer ptr is initialized with the address of x. x.display();

Above call is equivalent to,

```
ptr->show();
```

***Example*:**

```
#include<iostream> using namespace std;

class item {
        int code; float price;
    public:
        void getdata(int a, float b) {
            code =a; price=b;
        }

        void show(void) {
            cout<<"code"<<code<<"\n": cout<<"price"<<price<<"\n";
        } };
const int size =2; int main()
{
    item *p =new item[size]; item *d = p;
    int x, i; float y;

    for(i=0; i<size; i++) {
        cout<<"input code and price for item"<<i+1; cin>>x>>y;
        p->getdata(x, y); p++;
    }

    for(i=0; i<size; i++) {
        cout<<"item:"<<i+1<<"\n";
```

```
            d->show(); d++;
        }
        return 0; }
```

3) Explain 'this' pointer with *Example*.

- **'this'** pointer represent an object that invoke or call a member function.
- It will point to the object for which member function is called.
- It is automatically passed to a member function when it is called. It is also called as implicit argument to all member function.
- For ***Example***: S.getdata();
- Here **S** is an object and **getdata()** is a member function. So, **'this'** pointer will point or set to the address of object **S.**
- Suppose **'a'** is private data member, we can access it only in public member function like as follows a=50;
- We access it in public member function by using **'this'** pointer like as follows this->a=50;
- Both will work same.

***Example*:**

```
#include <iostream> using namespace std;

class sample {
        int a;

        public: sample() {
                a=10; }
        void disp(int a) {
                cout<<"The value of argument a="<<a; cout<<"\nThe value of data member
                a="<<this->a;
        } };

int main() {
        sample S; S.disp(20); return 0;
}
```

Output:

The value of argument a=20

The value of data member a=10

- The most important **advantage** of **'this'** pointer is, If there is same name of argument and data member than you can differentiate it.
- By using **'this'** pointer we can access the data member and without **'this'** we can access the argument in same function.

4) Explain pointer to derived class.

We can use pointers not only to the base objects but also to the objects of derived classes.

A single pointer variable can be made to point to objects belonging to different classes.

For
***Example*:**

```
                    //pointer to class B type variable //base object
B *ptr              // derived object
B b;                // ptr points to object b
D d;
ptr =&b;
```

In above ***Example*** **B** is base class and **D** isa derived class from **B,** then a pointer declared as a pointer to **B**

and point to the object b.

We can make **ptr** to point to the object **d** as follow

```
ptr = &d;
```

We can access those members of derived class which are inherited from base class by base class pointer. But we cannot access original member of derived class which are not inherited by base class pointer.
We can access original member of derived class which are not inherited by using pointer of derived class. ***Example*:**

```
#include <iostream> using namespace std;

class base {
        public: int b;
        void show() {
                cout<<"\nThe value of b"<<b; }
};
class derived:public base {
        public: int d;
        void show() {
                cout<<"\nThe value of b="<<b <<"\nThe value of d="<<d;
        } };

int main() {
        base B; derived D; base *bptr; bptr=&B;
        cout<<"\nBase class pointer assign address of base class object";
```

```
        bptr->b=100; bptr->show(); bptr=&D;
        bptr->b=200;
        cout<<"\nBase class pointer assign address of derived class object"; bptr->show();
        derived *dptr; dptr=&D;
        cout<<"\nDerived class pointer assign address of derived class object"; dptr->d=300;
        dptr->show(); return 0;
}
```

Output:

Base class pointer assign address of base class object The value of b100
Base class pointer assign address of derived class object The value of b200
Derived class pointer assign address of derived class object The value of b=200
The value of d=300

5) Explain virtual function with *Example*.

- It is a run time polymorphism.
- Base class and derived class have same function name and base class pointer is assigned address of derived class object then also pointer will execute base class function.
- To execute function of derived class, we have to declare function of base class as virtual.
- To declare virtual function just uses keyword virtual preceding its normal function declaration.
- After making virtual function, the compiler will determine which function to execute at run time on the basis of assigned address to pointer of base class.

Rules for virtual function

1. The virtual functions must be member of any class.
2. They cannot be static members.
3. They are accessed by using object pointers.
4. A virtual function can be a friend of another class.
5. A virtual function in a base class must be defined, even though it may not be used.
6. If two functions with the same name have different prototypes, C++ considers them as overloaded functions, and the virtual function mechanism is ignored.
7. We cannot have virtual constructors, but we can have virtual destructors.
8. The derived class pointer cannot point to the object of base class.
9. When a base pointer points to a derived class, then also it is incremented or decremented only relative to its base type. Therefore we should not use this method to move the pointer to the next object.
10. If a virtual function is defined in base class, it need not be necessarily redefined in the derived class. In such cases, call will invoke the base class.

***Example*:**

```
#include <iostream> using namespace std;

class base {
        public:

         void disp() {
                cout<<"\nSimple function in base class"; }
        virtual void show() {
                cout<<"\nVirtual function of Base class"; }
};

class derived: public base {
        public:  void disp() {
                cout<<"\nSame name with simple function of base class in derived class";
        }
        void show() {
               cout<<"\nSame name with virtual function of base class in derived class";
        } };

int main() {
        base B; derived D; base *bptr; bptr=&B;
        cout<<"\nBase class pointer assign address of base class object"; bptr->disp();
        bptr->show(); bptr=&D;
        cout<<"\nBase class pointer assign address of derived class object"; bptr->disp();
        bptr->show(); return 0;
}
```

Output:

- Base class pointer assign address of base class object
- Simple function in base class Virtual function of Base class
- Base class pointer assign address of derived class object Simple function in base class
- Same name with virtual function of base class in derived class

6) Explain Run time polymorphism with *Example*.

- We must access virtual functions through the use of a pointer declared as a pointer to the base class.
- We can also use object name and dot operator to call virtual functions.
- But, **runtime polymorphism using virtual functions** is achieved only when a virtual function is accessed through a pointer to the base class.

***Example*:**

```
#include<iostream> #include<cstring> using namespace std;

class media {
        protected:
                char title[50]; float price;
        public:
                media(char *s, float a) {
                strcpy(title, s); price = a;
```

```
        }

        virtual void display() { } };

class book: public media {
        int pages; public:
        book(char *s, float a, int p):media(s, a) {
                pages = p; }
        void display(); };

class tape:public media {
        Float time; Public:
        Tape(char *s, float a, float t):media(s, a) {
        Time = t; }
        Void display(); };

void book::display() {
    cout<<"\nTitle:"<<title; cout<<"\nPages:"<<pages; cout<<"\nPrice:"<<pages;
}
void tape::display() {
    cout<<"\nTitle:"<<title;



    cout<<"\nPlay time:"<<pages; cout<<"\nPrice:"<<pages;
}
int main() {
    char * title=new char[30]; float price, time;
    int pages;

    // book details
    cout<<"\n enter book details\n"; cout<<"\n title: ";
    cin>>title; cout<<"\n price: "; cin>>price; cout<<"\n pages: "; cin>>pages;

    book book1(title, price, pages);

    cout<<"\n enter tape details \n"; cout<<"Title: ";
    cin>>title;

    cout<<"Price: "; cin>>price;
    cout<<"Play time (mins):"; cin>>time;
    tape tape1(title, price, time);

    media* list[2]; list[0] = &book1; list[1] = &tape1;

    cout<<"\n MEDIA DETAILS";

    cout<<"\n ……BOOK……"; list[0] -> display();

    cout<<"\n ……TAPE……"; list[i] -> display();

    return 0; }
```

7) Explain pure virtual functions.

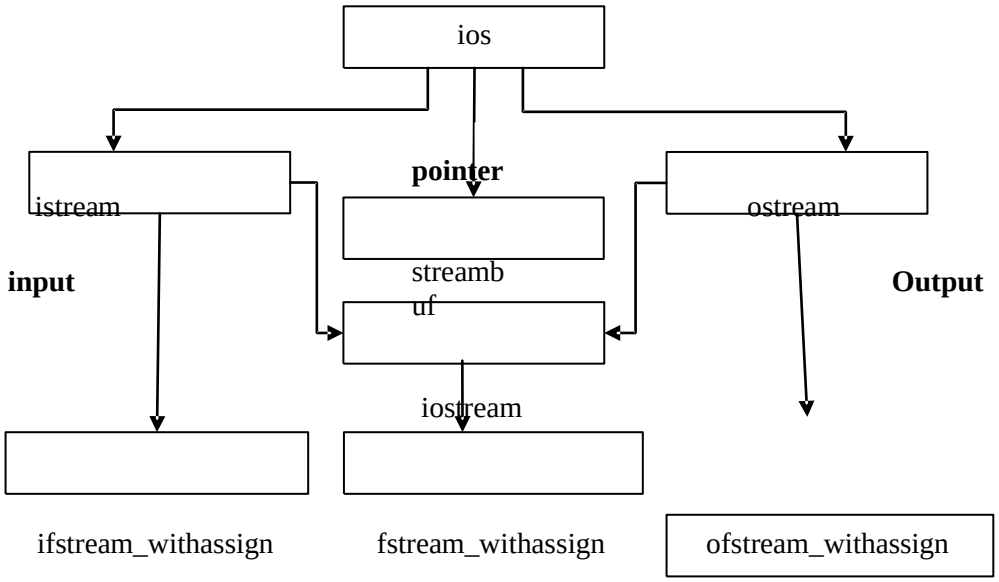
- A pure virtual function means ‘do nothing’ function.
- We can say empty function. A pure virtual function has no definition relative to the base class.

- Programmers have to redefine pure virtual function in derived class, because it has no definition in base
- class.
- A class containing pure virtual function cannot be used to create any direct objects of its own. This type of class is also called as abstract class.

Syntax:

virtual void display() = 0; OR virtual void display() {}

1) C++ Stream Classes



2) List out and explain Unformatted I/O Operations.

- C++ language provides a set of standard built-in functions which will do the work of reading and displaying
  - data or information on the I/O devices during program execution.
- Such I/O functions establish an interactive communication between the program and user.

Functio	Syntax	Use
cout	cout<<" "<<" ";	To display character, string and number on output device.
cin	cin>> var1>>var2;	To read character, string and number from input device.
get(char*)	char ch;	To read character including blank space, tab and newline

	cin.get(ch);	character from input device. It will assign input character to its argument.
get(void)	char ch; ch=cin.get();	To read character including blank space, tab and newline character from input device. It will returns input character.
put()	char ch; cout.put(ch);	To display single character on output device. If we use a number as an argument to the function put(), then it will convert it into character.
getline()	char name[20]; int size=10; cin.getline(name,size);	It is used to reads a whole line of text that ends with a newline character or size -1 character. First argument represents the name of string and second argument indicates the number of character to be read.
write()	char name[20]; int size=10; cout.write(name,size);	It is used to display whole line of text on output device. First argument represents the name of string and second argument indicates the number of character to be display.

Example of **cin** and **cout**:

```
#include <iostream> using namespace std; int main()
{
    int a;
    cout<<"Enter the number"; cin>>a;
    cout<<"The value of a="<<a; return 0;
}
```

Example of **get(char*)**, **char(void)** and **put()**:

```
#include <iostream>
using namespace std; int main()
{
    int a=65; char ch;
    cin.get(ch);   //get(char*) cout.put(ch);     //put() ch=cin.get(); //get(void) cout.put(ch);
    cout.put(a);
    return 0; }
```

Example of **getline()** and **write()**:

```
#include <iostream>
using namespace std; int main()
{
    int size=5; char
    name[50];
    cin.getline(name,size);
    cout.write(name,size);          //getline()
    return 0;                       //write
}
```

3) List out and explain functions and manipulators used for Formatted I/O operations.

We can format input and output by following methods.

1. **ios** class funstions and flags.
2. Manipulators.

3. User-defined output functions. Now we will see each method in detail.

The ios format functions are shown in below table:

Functi	Syntax	Use
width()	cout.width(size);	To specify the required field size for displaying an output value.
precision( )	cout.precision(2);	To specify the number of digits to be displayed after the decimal point of a float value.
fill()	cout.fill('character');	To specify a character that is used to fill the unused portion of
setf()	cout.setf(arg1, arg2);	To specify format flags that can control the form of output display such as left or right justification.
unsetf()	cout.resetiosflags()	To clear the flags specified.

In setf() we can provide one or two argument.

cout.setf(arg1, arg2);

The arg1 is formatting flags defined in the ios class. And arg2 is known as bit field specifies the group to which the formatting flags belong.

The flags and bit field are shown below

Format required	Flag (arg1)	Bit-field (arg2)
Left justified output	ios::left	ios::adjustfield
Right justified output	ios::right	ios::adjustfield
Padding after sign or base indicator (like +##20)	ios::internal	ios::adjustfield
Scientific notation	ios::scientific	ios::floatfield
Fixed point notation	ios::fixed	ios::floatfield
Decimal base	ios::doc	ios::basefield
Octal base	ios::oct	ios::basefield
Hexadecimal base	ios::hex	ios::basefield

The flags without field bit are shown below

Flag	Meaning
ios::showbase	Use base indicator on output.
ios::showpos	Print + before positive numbers.
ios::showpoint	Show trailing decimal point and zeros.
ios::uppercase	Use uppercase letters for hex output.
ios::skipus	Skip white space on input.
ios::unitbuf	Flush all streams after insertion.
ios::stdio	Flush stdout and stderr after insertion.

Example of **ios** functions:

```
#include <iostream> #include <math>  using namespace std; int main()
{
    cout.fill('*'); cout.setf(ios::left,ios::adjustfield); cout.width(10);
    cout<<"value"; cout.setf(ios::right,ios::adjustfield); cout.width(15);
    cout<<"SQRT OF VALUE"<<"\n"; cout.fill('.'); cout.precision(4); cout.setf(ios::showpoint);
    cout.setf(ios::showpos);
    cout.setf(ios::fixed,ios::floatfield);


    for(int i=1;i<=10;i++)
    {
        cout.setf(ios::internal, ios::adjustfield); cout.width(5);
        cout<<i;


        cout.setf(ios::right, ios::adjustfield);
        cout.width(20); cout<<sqrt(i)<<"\n";
    }
    cout.setf(ios::scientific, ios::floatfield);

    cout<<"\nSQRT(100)="<<sqrt(100)<<"\n"; return 0;
}
```

Output:

```
value********SQRT OF VALUE +...1.............+1.0000 +...2.............+1.4142 +...3.............+1.7321
+...4.............+2.0000 +...5.............+2.2361 +...6.............+2.4495 +...7.............+2.6458
+...8.............+2.8284 +...9.............+3.0000 +..10.............+3.1623

SQRT(100)=+1.0000e+01
```

The manipulators are shown in below table:

Manipulators	Use
setw()	To specify the required field size for displaying an output value.
setprecision()	To specify the number of digits to be displayed after the decimal point of a float value.
setfill()	To specify a character that is used to fill the unused portion of a field.
setiosflags()	To specify format flags that can control the form of output display such as left or right justification.
resetiosflags()	To clear the flags specified.

Manipulators are used to manipulate the output in specific format.

Example for manipulators

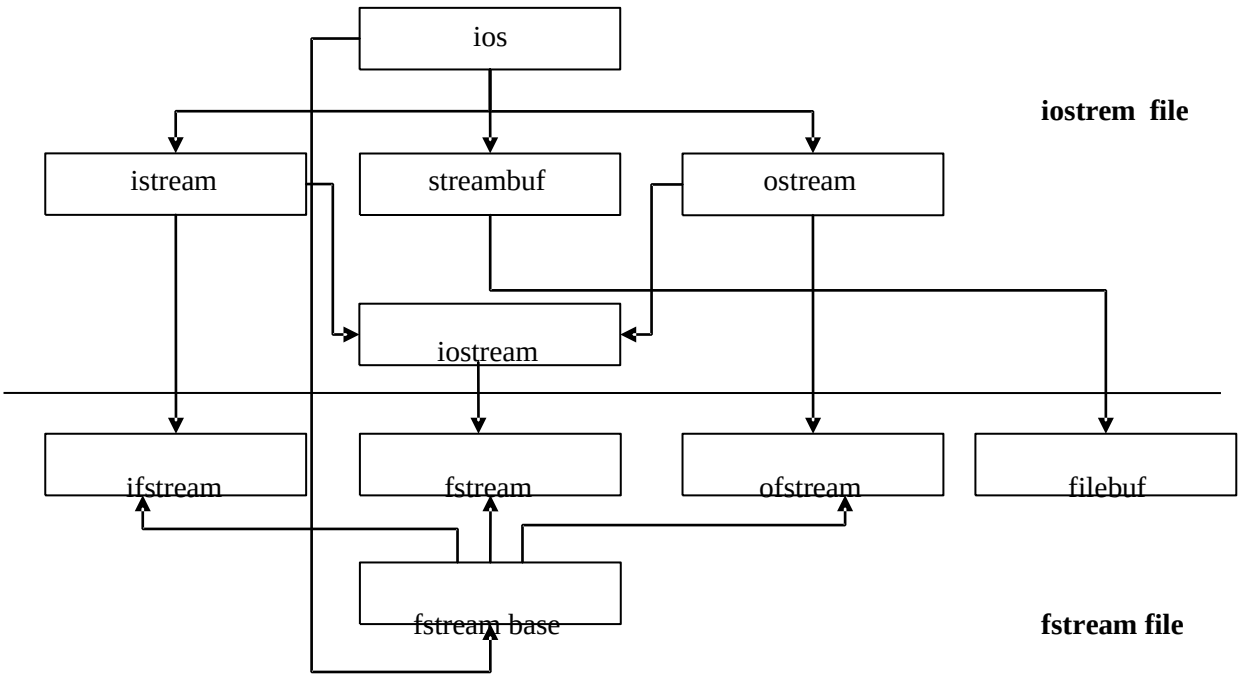
```
#include <iostream> #include <iomanip> using namespace std; int main()
{
    cout.setf(ios::showpoint); cout<<setw(5)<<"n"
        <<setw(15)<<"Inverse of n" <<setw(15)<<"Sum of terms\n\n";

    double term,sum=0;
```

```
for(int n=1;n<=10;n++) {
        term=1.0/float(n);



        sum=sum+term;


        cout<<setw(5)<<n <<setw(14)<<setprecision(4) <<setiosflags(ios::scientific)<<term
            <<setw(13)<<resetiosflags(ios::scientific) <<sum<<endl;
}
return 0; }
```

Output:

```
n   Inverse of n Sum of terms

1    1.0000e+00       1.0000 2         5.0000e-01      1.5000 3         3.3333e-01      1.8333 4
     2.5000e-01      2.0833 5 2.0000e-01      2.2833 6         1.6667e-01      2.4500 7
     1.4286e-01      2.5929 8 1.2500e-01      2.7179 9         1.1111e-01      2.8290
10   1.0000e-01       2.9290
```

4) Explain file stream classes with iostream classes in short.



Class	Contents
filebuf	Its purpose is to set the file buffers to read and write. Contains **Openprot** constant used in the **open()** of file stream classes. Also contain **close()** and **open()** as members.
fstreambase	Provides operations common to the file streams serves as a base for **fstream, ifstream** and **ofstream** classes. Contains **open()** and **close()** functions.
ifstream	Provides input operations. Contains **open()** with default input mode. Inherits the functions **get(), getline(), read(), seekg()** and **tellg()** functions from **istream.**
ofstream	Provides output operations. Contains **open()** with default output mode. Inherits **put(), seekg(), tellp(),** And [illegible] functions from ostream.
fstream	Provides support for simulations input and output operations. Contains **open()** with default input mode. Inherits all the functions from **istream** and **ostream** class through iostream.

5) List out various File mode parameters and explain in short.

Parameter	Meaning
ios::app	Append to end of file.
ios::ate	Go to end of file on opening.
ios::binary	Binary file.
ios::in	Open file for reading only.
ios::nocreate	Open fails if the file does not exist.
ios::noreplace	Open file if the file already exists.
ios::out	Open file for writing only.
ios::trunc	Delete the contents of the file it it exists.

- Opening a file in **ios::out** mode also opens it in the **ios::trunc** mode by default.
- Both ios::app and ios::ate take us to the end of the file when it opened. The difference between the two parameters is that the ios::app allows us to add data to the end of file only, while ios::ate mode permits us to add data or modify the existing data anywhere in the file. In both the cases, a file is created by the specified name, if it does not exist.
- Creating a stream using ifstream implies input and creating a stream using ofstream implies output. So, in these cases it is not necessary to provide the mode parameters.
- The mode can combine two or more parameters using the bitwise OR operator shown as follows. Fout.open("data", ios::app | ios::nocreate);
- This opens the file in the append mode but fails to open the file if it does not exist.

6) Explain File pointing functions in short.

Each file has two pointers one is getpointer to input and second one is putpointer to output.

Functions for manipulation of file pointer

Functions	Meaning
seekg()	Moves get pointer (input) to specified location.
seekp()	Moves put pointer (output) to specified location.
tellg()	Gives the current position of the get pointer.
tellp()	Gives the currint position of the put pointer.

For ***Example***:

infile.seekg(20);

int p=fileout.tellp();

We can also pass two argument in the seekg() and seekp() functions as below. seekg(offset, refposition); seekp(offset, refposition);

The parameter offser represent the number of bytes the file pointer is to be moved from the location specified by the parameter refposition.
For refposition the ios class provides following constants.

ios::beg	Starting of the file.
ios::c	Current position of the pointer.
ios::end	End of the file.

***Example*:**

```
#include<iostream> #include<fstream> #include<cstring> using namespace std;

class emp {
        char name[30]; int ecode;

        public: emp()
       { }

        emp(char *c, int c) {
                strcpy(name,n); ecode=c;
        } };


void main() {
        emp e[4]; e[0]=emp("amit",1); e[1]=emp("joy",2); e[2]=emp("rahul",3);
        e[3]=emp("vikas",4);

        fstream file;
        file.open("empolyee.dat", ios::in | ios::out);

        int i; for(i=0;i<4;i++)
        file.write((char *) &e[i], sizeof(e[i])); file.seekg(0, ios::end);
        int end=file.tellg();

        cout<<"number of objecs stored in employee.dat is"<<sizeof(emp); }
```

7) W. A. P. for reading and writing class objects.

```
#include<iostream> #include<fstream>

using namespace std;

class inventory {
        char name[10]; int code;  float cost;

        public:
                void readdata(); void writedata();
};

void inventory::readdata() {
        cout<<"Enter name"<<endl; cin>>name;
        cout<<"Enter code"<<endl; cin>>code;
        cout<<"Enter price/cost"<<endl; cin>>cost;
}

void inventory::writedata() {
        cout<<"Name ="<<name; cout<<"Name ="<<code; cout<<"Name ="<<cost;
}

int main() {
        inventory item[3]; fstream file;
        file.open("stock.txt",ios::in |ios::out);
        cout<<"Enter details of 3 items"; for(int i=0;i<3;i++)
        {
                item[i].readdata();
                file.write((char *)&item[i], sizeof(item[i])); }

        file.seekg(0); cout<<"output"; for(int i=0;i<3;i++) {
                file.read((char *)&item[i],sizeof(item[i])); item[i].writedata();
        } file.close(); return 0;
}
```

8) W. A. P. that write content of file1 in file2 but omits uppercase latters.

```
#include<iostream> #include<fstream>

using namespace std; int main()
{

        fstream f1,f2;

        f1.open("abc.txt",ios::in); f2.open("xyz.txt", ios::in);

        char ch;

        while(file) {
                f1.get(ch);

                if(ch<65 || ch<90) {
                        f2.put(ch); }

                return 0; }
}
```

1) Explain Function and Class Templates with appropriate *Example*.

C++ templates are a powerful mechanism for code reuse, as they enable the programmer to write code that behaves the same for any data type.

By template we can define generic classes and functions.

It can be considered as a kind of macro. When an object of a specific type is defined for actual use, the template definition for that class is substituted with the required data type.

Function Template:

Suppose you write a function printData:

```
void printData(int value) {
    cout<<"The value is "<<value; }
```

Now if you want to print double values or string values, then you have to overload the function:

```
void printData(float value) {
    cout<<"The value is "<<value; }
void printData(char *value) {
    cout<<"The value is "<<*value; }
```

To perform same operation with different data type, we have to write same code multiple time.

C++ provides templates to reduce this type of duplication of code.

```
template<typename T> void printData(T value) {
    cout<<"The value is "<<value; }
```

We can now use printData for any data type. Here T is a template parameter that identifies a type. Then, anywhere in the function where T appears, it is replaced with whatever type the function is instantiated.

For ***Example***:

```
int i=3;  float d=4.75;
char *s="hello"; printData(i); // T is int printData(d); // T is float printData(s); // T is string
```

Function Templates with Multiple Parameters:

We can use more than one generic data type in the template statement, using comma seperated list as shown below. For ***Example***:

```
template <class T1, class T2> void printData(T1 a, T2 b)
{




        cout<<"\na="<<a<<"\tb="<<b; }
int main() {
        printData(12,'N'); printData(12.5,34); return 0;
}
```

2) Explain class template. Also write a C++ program for class template with multiple parameters.

Class Template:

Suppose you want to perform addition of two integer number in class

```
class add {
        int a,b,c; public: add()
```

```
        {
                a = 10; b = 20;
              c = a + b; }
        void putdata() {
                cout<<"\nThe sum="<<c; }
};
```

If you later decide you also want to perform addition of two float numbers in class then you have to write separate class for addition of two numbers.

```
class add {
        float a,b,c; public:  add()
        {
                a = 10.45; b = 20.67;  c = a + b;
        }
        void putdata() {
                cout<<"\nThe sum="<<c; }
};
```

It means we have to write same class two time because of only different data type. We can solve this problem by using template as follow.

```
template<class T> class Sample
{
        T a,b,c; public:
        void getdata() {
                cout<<"Enter the value of a & b\n"; cin>>a>>b;




        }


        void sum() {
                c=a+b; }
        void putdata() {
                cout<<"\nThe sum="<<c; }
};

int main() {
        clrscr();
        Sample <int> S1; S1.getdata(); S1.sum(); S1.putdata(); Sample <float> S2; S2.getdata();
        S2.sum(); S2.putdata(); getch();
        return 0; }
```

We will pass data type when we create object of class. So, now template variable „T‟ is replace by data type which we passed with object.

Class Templates with Multiple Parameters:

We can use more than one generic data type in a class template. They are declared as a comma separated list within the template specification as shown below.

```
template <class T1, class T2> class classname
{
      /*Statement 1; Statement 2;
          . .
        Statement n; */ };
```

***Example*:**

```
template<class T1, class T2> class Sample
{
        T1 a; T2 b; public:
        Sample(T1 x,T2 y) {
                a=x; b=y;
        }
        void disp() {
                 cout<<"\na="<<a<<"\tb="<<b;




        } };
int main() {
        Sample <int,float> S1(12,23.3); Sample <char,int> S2('N',12); S1.disp();
        S2.disp(); return 0;
}
```

3) What is exception handling? Explain how to handle an exception with appropriate *Example*. OR Explain how to handle exceptions using try, catch and throw mechanism.

In programming two most common types of error are logical error and syntax error.

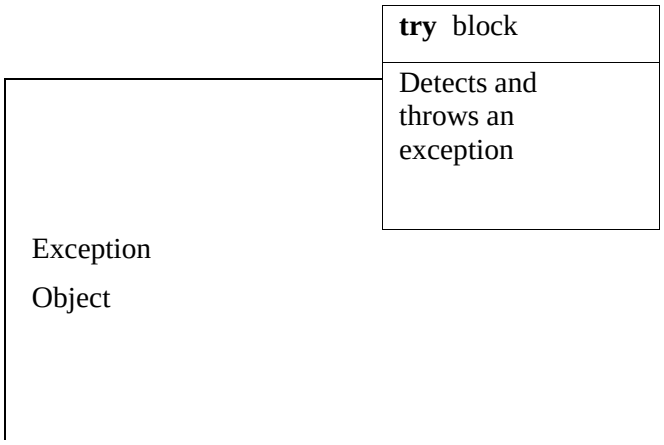
The syntax error is detected during compilation of program, but the logical error will detect during execution of program. So, it is very difficult to handle logical error.
The exception handling provides mechanism to handle logical error during execution of program. Steps to handle logical error:

1. Find the problem (Hit the exception)
2. Inform that an error has occurred (Throw the exception) 3. Receive the error information (Catch the exception)
4. Take corrective actions (Handle the exception)

Exception Handling Mechanism:

This mechanism is built upon three keyword: try, throw and catch.

try is used to preface a block of statement which may generate exceptions. When an exception is detected, it is thrown using a throw statement.
A catch block is used to catches the exceptions thrown by throw statement and take appropriate action. The relationship between try, throw and catch block is as shown in below figure.



Syntax:

```
try {
    // Set of Statments; throw exception;
    // Set of Statements; }
catch(type arg) {
    // Set of Statements; }
```

***Example*:**

```
#include <iostream> using namespace std;

int main() {
    int a,b;
    cout<<"Enter the value of a and b\n"; cin>>a>>b;
    try {
        if(b != 0)
            cout<<"The result(a/b)="<<a/b; else
            throw(b); }
    catch(int x) {
        cout<<"Exception caught b="<<x; }
    return 0; }
```

When the value of b is zero at that time exception will throw and this exception will catch in catch block and print the message value is zero.

4) Explain multiple catch statements with appropriate *Example*.

It is possible that a program segment has more than one condition to throw an exception.

For these situations we can associate more than one catch statement with a try block.

When an exception is thrown, the exception handlers are searched in order for an appropriate match. The fist handler that yields a match is executed.
After executing the handler, the control goes to the first statement after the lat catch block for that try.
Note: It is possible that arguments of several catch statements match the type of an exception.
In such cases, the first handler that matches the exception type is executed. ***Example*:**

```
#include<iostream> using namespace std;

void test(int x) {
    try

    {
        if(x==1) {
            throw(x); }
        else if(x==0) {
            throw 'x'; }
        else if(x==-1) {
            throw 1.0; }
    }

    catch(char c) {
        cout<<"Caught a character"<<endl; }
    catch(int m) {
        cout<<"Caught an integer"<<endl; }
    catch(float n) {
        cout<<"Caught a double"<<endl; }
    cout<<"End of the try catch system"<<endl; }

int main() {
    cout<<"Testing multiple catch"<<endl; cout<<"x==1"<<endl;
```

```
test(1); cout<<"x==0"<<endl; test(0);
cout<<"x==-1"<<endl; test(-1); cout<<"x==2"<<endl; test(2);
return 0; }
```

5) Explain Rethrowing an Exception with appropriate *Example*.

A handler may decide to rethrow the exception caught without processing it.

In such situations, we may simply invoke throw without any arguments as shown below:

```
throw;
```

This causes the current exception to be thrown to the next enclosing try/catch sequence and is caught by a catch statement listed after that enclosing try block.

***Example*:**

```
#include<iostream> using namespace std;

void divide(double x, double y)

{
        cout<<"Inside function \n"; try
        {
                if (y==0) throw y; else
                cout<<"Division ="<<x/y<<"\n"; }
        catch(double) {
                cout<<"caught double inside the function \n"; throw;
        }
        cout<<"End of function \n\n"; }

int main() {
        cout<<"Inside main"; try
        {
                divide(10.5,2.5); divide(4,0);
        } catch(double) {
                cout<<"caught double inside main \n"; }
        cout<<"End of main\n";

        return 0; }
```

6) Explain short introduction of STL.

Alexander Stepanov and Meng Lee of Hewlett-jPackard developed a set of general-purpose templatized

classes and function for storing and processing of data.

The collection of these generic classes is called **Standard Template Library(STL).** The STL has now become a part of the ANSI standard C++ library.
Using STL, we can save considerable time and effort, and lead to high quality programs.

All these benefits are possible because we are basically "reusing" the well-written and well-tested components defined in the STL.
STL components which are now part of the standard C++ library are defined in the namespace **std.**
We must therefore **using namespace** directive, to intend to use the Standard C++ library.

```
using namespace std;
```

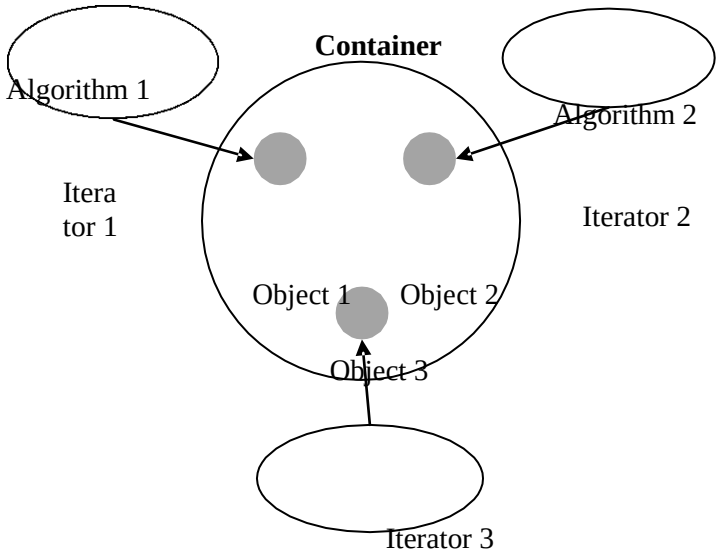
7) List out components of STL and explain in short.

There are three core components of STL as follows: 1. Containers

2. Algorithms
3. Iterators

These three components work in conjunction with one another to provide support to a variety of programming solution.

The relationship between the three components is shown in following figure,



Algorithm 3

Container:

A container is an object that actually stores data.

It is a way data is organized in memory.

The STL containers are implemented by template classes and therefore can be easily customized to hold different types of data.

Algorithm:

An algorithm is a procedure that is used to process the data contained in the containers.

The STL includes many different kinds of algorithms to provide support to taks such as initializing, searching, copying, sorting, and merging.
Algorithms are implemented by template functions.

Iterator:

The iterators is an object(like a pointer) that points to an element in a container. We can use iterators to move through the contents of containers.
Iterators are handled just like pointers. We can increment or decrement them.
Iterators connect algorithms with containers and play a key role in the manipulation of data stored in the containers