**OVERRIDING:**

If derived class defines same function as defined in its base class, it is known as function **overriding**. It is used to achieve runtime polymorphism.

**#include <iostream>**

**using namespace std;**

**class Animal**

**{**

**public:**

**void Speak()**

**{**

**cout << "Animals can speak"<<endl;**

**}**

**};**

**class Dog:public Animal**

**{**

**public:**

**void Speak()**

**{**

**cout << "Barks Barks"<<endl;**

**}**

**};**

**int main()**

**{**

**Dog d;**

**//here it calls the that function whose object is created**

**d.Speak();**

**return 0;**

**}**

**VIRTUAL FUNCTION:**

A virtual function is a member function of a base class that you redefine in a derived class. It is declared using the **virtual** keyword.

**#include <iostream>**

**using namespace std;**

**class Animal**

**{**

**public:**

**virtual void Speak()**

**{**

**cout << "Animals can speak"<<endl;**

**}**

**};**

**class Dog:public Animal**

**{**

**public:**

**void Speak()**

**{**

**cout << "Barks Barks"<<endl;**

**}**

**};**

**class Cat:public Animal**

**{**

**public:**

**void Speak()**

**{**

**cout << "Meowoo Meowoo"<<endl;**

**}**

**};**

**int main()**

**{ //pointer**

**Animal \*ptr;**

**Cat c;**

**//pointer hmesha apny andr address lyta**

**ptr = &c;**

**ptr->Speak();**

**return 0;**

**}**

**PURE VIRTUAL FUNCTION:**

A virtual function with no definition in base class but the function must be implemented by all its derived class. It is declared by the following way:

**Virtual void function() = 0;**

**ABSTRACT CLASS:**

If a class have at least one pure virtual function called Abstract Class. We cannot make object in any abstract class.

**INTERFACE:**

An interface is a special type of abstract class and all its function must be pure virtual function.

Every interface is an abstract class but not every abstract class is an interface.

e.g.

**#include<iostream>**

**using namespace std;**

**class Animal**

**{**

**public:**

**virtual void speak() = 0;**

**};**

**class Dog: public Animal{**

**public:**

**void speak()**

**{**

**cout<<"dog speaks";**

**}**

**};**

**int main()**

**{**

**Dog d;**

**d.speak();**

**}**

**TEMPLATE:**

A template is a blueprint for creating a generic class and function. In a simple way, you can create a single function or single class to work with different data type using template.

There are two types of templates:

Class Template

Function Template

**FUNCTION TEMPLATE:**

**#include<iostream>**

**using namespace std;**

**template<class S>**

**void getter(S a, S b)**

**{**

**cout<<a<<endl;**

**cout<<b;**

**}**

**int main()**

**{**

**getter("shiza","absa");**

**return 0;**

**}**

**CLASS TEMPLATE:**

**#include<iostream>**

**using namespace std;**

**template<class S>**

**class test{**

**public:**

**void getter(S a, S b)**

**{**

**cout<<a<<endl;**

**cout<<b;**

**}**

**};**

**int main()**

**{**

**test<string> sheeza;**

**sheeza.getter("shiza","absa");**

**return 0;**

**}**

**EXCEPTION HANDLING:**

To avoid program from crash, we will use exception handling.

**#include<iostream>**

**using namespace std;**

**int main()**

**{**

**int a, b, c;**

**a = 10;**

**b = 5;**

**try{**

**if(b==0)**

**{**

**throw c;**

**}**

**else**

**{**

**c = a/b;**

**cout<<c<<endl;**

**cout<<"Program Runs";**

**}**

**}**

**catch(...)**

**{**

**cout<<"solution not possible";**

**}**

**return 0;**

**}**

**ASSOCIATION:**

Association is a relationship between two classes where one class use another. It means there is almost always a link between objects. Each object has its own lifecycle and there is no parent or owner. It is represented as **has a** relationship.

E.g. relationship between patient and doctor.

**AssociatedClass**

{

**BaseClass** variableName;

..

//Other members of class

..

}

**RELATIONSHIP:**

* One to One
* One to Many
* Many to Many

**TYPES OF ASSOCIATION:**

* **Unary Association**

Association between two object of same type.

* **Binary Association**

Association between two different types of object.

* **Ternary Association**

Association between three different types of object.

**AGGREGATION:**

Aggregation is represented as **has a** Relationship. It is unidirectional association i.e. one way relationship e.g. department can have students but vice versa is not possible, Moreover the child can exist independently of the parent. And in aggregation, both the entities will survive independently.

**#include<iostream>**

**using namespace std;**

**class Hostel{**

**public:**

**int street, houseno, room;**

**string city;**

**void hostelsetter(int sn, int hn, string ct, int roomno)**

**{**

**street = sn;**

**houseno = hn;**

**city = ct;**

**room = roomno;**

**}**

**};**

**class Student{**

**Hostel \*h;**

**string name;**

**public:**

**void setter(string sname, Hostel \*hostel)**

**{**

**name = sname;**

**h = hostel;**

**}**

**void getter()**

**{**

**cout<<"Name: "<<name<<endl;**

**cout<<"Room #: "<<h->room;**

**}**

**};**

**int main()**

**{**

**Hostel h;**

**h.hostelsetter(1,1,"lahore",200);**

**Student s;**

**s.setter("maham", &h);**

**s.getter();**

**}**

**COMPOSITION:**

Composition is a restricted form of Aggregation in which two entities are highly dependent on each other. It represents **part-of** relationship. If the one object destroy then the other will destroy automatically.

**#include<iostream>**

**using namespace std;**

**class Hostel{**

**public:**

**int room;**

**};**

**class Student{**

**Hostel h;**

**string name;**

**public:**

**void setter(string sname, int room1)**

**{**

**name = sname;**

**h.room = room1;**

**}**

**void getter()**

**{**

**cout<<"Name: "<<name<<endl;**

**cout<<"Room #: "<<h.room<<endl;**

**}**

**};**

**int main()**

**{**

**Student s;**

**s.setter("maham", 102);**

**s.getter();**

**}**

**OPERATOR OVERLOADING:**

We can change the way operators work for user-defined types like objects and structures. This is known as operator overloading.

It helps us to do Obj3=Obj1+Obj2 instead of using setter functions.

**#include <iostream>**

**using namespace std;**

**class Addition**

**{**

**int num1;**

**int num2;**

**public:**

**Addition()**

**{**

**}**

**Addition(int x, int y)**

**{**

**num1 = x;**

**num2 = y;**

**}**

**void setNum1(int n1)**

**{**

**num1 = n1;**

**}**

**void setNum2(int n2)**

**{**

**num2 = n2;**

**}**

**Addition operator + (Addition &obj)**

**{**

**Addition temp;**

**temp.setNum1(num1+obj.num1);**

**temp.setNum2(num2+obj.num2);**

**return temp;**

**}**

**void display()**

**{**

**cout << num1 <<endl;**

**cout << num2 <<endl;**

**}**

**};**

**int main()**

**{**

**Addition a1(2, 4);**

**Addition a2(2,1);**

**Addition a3 = a1+a2;**

**a3.display();**

**return 0;**

**}**

**Types**

* Unary Operator Overloading
* Binary Operator Overloading
* Stream Insertion << and Stream Extraction >>

**Stream Overloading**

**#include <iostream>**

**using namespace std;**

**class Person**

**{**

**int id;**

**public:**

**friend istream& operator >>(istream& in, Person& obj);**

**friend ostream& operator <<(ostream& out, Person& obj);**

**};**

**istream& operator >>(istream& in, Person& obj) //extraction operator overloading**

**{**

**cout << "Enter ID"<<endl;**

**in >> obj.id;**

**return in;**

**}**

**ostream& operator <<(ostream& out, Person& obj) //extraction operator overloading**

**{**

**out<< obj.id;**

**return out;**

**}**

**int main()**

**{**

**Person p1;**

**cin>>p1;**

**cout<<p1;**

**return 0;**

**}**

**DIAMOND PROBLEM**

Diamond problem occurs when we inherit more than one base class in the same derived class and all these base classes also inherit another but same single class (parent class).

And Virtual inheritance solves that problem that ensures that only one copy or instance of the base class’s is inherited by the grandchild derived class.

**#include<iostream>**

**using namespace std;**

**class Animal {**

**string food;**

**public:**

**void Eat(string food1)**

**{**

**food = food1;**

**}**

**void display()**

**{**

**cout<<"Bat eats "<<food;**

**}**

**};**

**// Two classes virtually inheriting Animal:**

**class Mammal: virtual public Animal {**

**public:**

**int Breathe()**

**{**

**cout<<"mammel here"<<endl;**

**return 0;**

**}**

**};**

**class WingedAnimal: virtual public Animal {**

**void Flap() {}**

**};**

**class Bat: public Mammal, WingedAnimal**

**{**

**};**

**int main()**

**{**

**Bat b;**

**b.Eat("choclate");**

**b.display();**

**return 0;**

**}**