OBJECT ORIENTED PROGRAMMING

(with C++)

Unit-3

Inheritance

Dr. Gaurav Gupta
Associate Professor (IT Deptt.)

Unit-III

Inheritance, Class hierarchy, derivation – public, private & protected; aggregation, composition vs classification hierarchies
polymorphism, categorization of polymorphic techniques, method polymorphism, polymorphism by parameter, operator overloading, parametric polymorphism,
generic function – template function, function name overloading, overriding inheritance methods, run time polymorphism.

What is Inheritance ?

- Inheritance permits the reusability of software i.e.
- Inheritance provides a powerful way to extend the capabilities of existing classes, and to design programs using hierarchical relationships.
- The new inheriting class is called ***derived*** or ***sub*** class and inherited class is called ***base*** or ***super*** class.
- The derived class can add other features of its own, so it becomes a specialized version of the base class.

Need of Inheritance ?

- Specialisation - Extending the functionality of an existing class
- Generalisation - Sharing commonality between two or more classes e.g.
  - Some Objects share the same attributes
  - Some Objects share the same methods
  - Some Objects require similar functionality

Forms of Inheritance

(a) Single Inheritance
(b) Multiple Inheritance
(c) Hierarchical Inheritance
(d) Multilevel Inheritance
(e) Hybrid Inheritance

How to inherit a class?

1. Define a base class of which all properties are to be inherited.
2. Define a derived class as:

class derived_cls_name : **access-specifier** base_cls_name

{ BODY OF THE DERIVED CLASS } ;

3. Access specifier can be either public, protected or private (if nothing is specified then default access is private)

   Access specifier also called visibility mode which specifies whether the features of the base class are privately or publicly derived.

Inheritance...

- If access specifier is **private** then

1. All public and protected members of base class become private members of derived class.
2. Private members of base class remain as private to it and hence not accessible to derived class.
3. All public and protected members of a class can be accessed by its own objects using the dot operator.
4. No member of base class is accessible to the objects of the derived class.

- If access specifier is **public**

1. All public members of the base class become public members of derived class.
2. All protected members of base class become protected members of derived class.
3. Private members of base class are still private to base class ,thus in-accessible to derived class.

Inheritance using public access specifier

```
#include"iostream.h"
#include"conio.h"
#include<string.h>
class student
{
int rollno; char name[12];
public:
 void get_details()
 {  cout<<"enter roll no and name";
    cin>>rollno>>name; }
 void display()
 {  cout<<"roll no is "<<rollno<<"and name is "<<name;}
  }; // Base class ends here
```

```
class cse_student :public student
{
char sub1[10],sub2[10];
public:
cse_student() // CONSTRUCTOR
{ strcpy(sub1,"OOPS");
  strcpy(sub2,"PROGRAMMING");}
void sub_display()
{ cout<<"and subjects are"<<sub1<<"and"<<sub2;
} };
```

```
int main (void)
{
cse_student c1;
c1.get_details();
c1.display();
c1.sub_display();
return 0;
}
```

Public members of base class are accessed as public of derived

Output

- enter roll no and name 12 ABC
- roll no is 12
- and name is ABC
- and subjects are OOPS and PROGRAMMING

Inheritance using private access specifier

- If access specifier is **private** then

1. All public and protected members of base class become private members of derived class.
2. Private members of base class remain as private to it and hence not accessible to derived class.
3. All public and protected members of a class can be accessed by its own objects using the dot operator.
4. No member of base class is accessible to the objects of the derived class.

Inheritance using private access specifier

```
#include"iostream.h"
#include"conio.h"
#include<string.h>
class student
{
int rollno; char name[12];
public:
void get_details()
{ cout<<"enter roll no and name";
  cin>>rollno>>name;}
void display()
{ cout<<"roll no is "<<rollno<<"and name is"<<name;}
};
```

```
class cse_student :private student
{
char sub1[10],sub2[10];
public:
cse_student()          // CONSTRUCTOR
{  strcpy(sub1,"OOPS");
   strcpy(sub2,"PROGRAMMING");
   get_details();
}
void sub_display()
{  display();
  // cout<<"rollno is "<<rollno; // Rollno still in accessible
    cout<<"and subjects are"<<sub1<<"and"<<sub2;}
};
```

Now private member of cse_student

```
int main (void)
{
cse_student c1;
//c1.get_details();
//c1.display();
c1.sub_display();
return 0;
}
```

$\left.\begin{array}{l} \\ \\ \end{array}\right\}$ private member of base class in-accessible

Output

```
enter roll no and name
12
neerja

roll no is 12 and name is neerja

and subjects are OOPS and PROGRAMMING
```

“Protected” Access Specifier

- A protected member of a class behaves similar to private members (***that is accessible only to members of the class)*** BUT
- If the class is inherited by some class then protected members of base class may be accessible to derived class (***different from PRIVATE MEMBERS OF BASE CLASS)***
- Thus protected members of a class are despite being private to it can be inherited by some other class.

```
#include"iostream.h"
#include"conio.h"
class student
{
protected:
int rollno; char name[12];
void get_details()
{ cout<<"enter roll no and name";
  cin>>rollno>>name;}
public:
student()
{ get_details();    }
 void display()
{ cout<<endl<<"roll no is "<<rollno<<"and name is
   "<<name<<endl; } };
```

```
class it_student :public student
{
char sub1[10],sub2[10];
public:
it_student()
{
strcpy(sub1,"OOPS");
strcpy(sub2,"PROGRAMMING");
get_details(); // valid as get_details is PROTECTED and
   not PRIVATE
}
void display1()
{
cout<<"rollno and name are"<<rollno<<"\t"<<name;
cout<<endl<<"and subjects are"<<sub1<<"and"<<sub2;
}};
```

```
int main (void)
{
clrscr();
student s;
    //s.get_details(); // get_details is private to student
s.display();        // valid being PUBLIC to student
it_student c1;
    //c1.get_details(); //get_details is private to cse_student
c1.display();       //valid ,display() is public to
                       //cse_student
c1.display1();
return 0;
}
```

Output

- enter roll no and name123 angel

- roll no is 123and name is angel
- enter roll no and name45 angel2
- rollno and name are45   angel2

  and subjects are OOPS and PROGRAMMING

Inheriting a class in protected mode

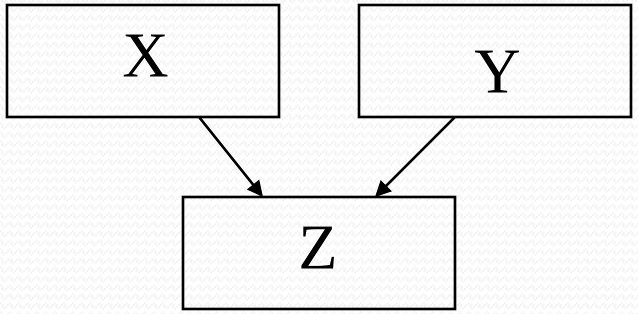
- Both public and protected members of base class become protected members of derived class.
- Private members of base class remain private to it and hence in-accessible to derived class.

Visibility of inherited members

Base class visibility	Derived class visibilty		
	Public derivation	Private derivation	Protected Derivation
Private	Not inherited	Not Inherited	Not Inherited
Protected	Protected	Private	Protected
Public	Public	Private	Protected

More on inheritance….

Derived class may inherit traits from more than one class (multiple inheritance)



Multiple Inheritance

Class Z:access_specifier X, access_specifier Y…

Example of multiple inheritance

```
#include"iostream.h"
#include"conio.h"
class X
{
protected:
int a;
public:
void getX (int m)
{a=m;}
void display()
{
cout<<"The number in X is
   "<<a<<endl;}
};
```

```
class Y
{
protected:
int b;
public:
void getY (int n)
{b=n;}
void display1()
{
cout<<"The number in Y is
   "<<b<<endl;
}};
```

class Z: public X, public Y

```
{
protected:
int c;
public:
Z(int k){
getX(1);
getY(2);
c=k;
}
void disp()
{
display();
display1();
cout<<"the number in Z is  <<c;
}
};
```

```
   int main (void)
{
clrscr();
Z obj(3);
obj.disp();
return 0;
}
```

Output

- The number in X is 1
- The number in Y is 2
- the number in Z is 3

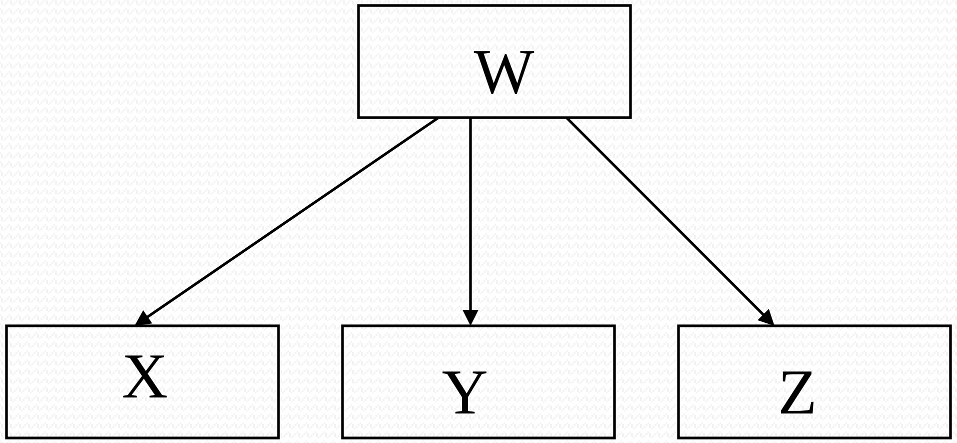
Ambiguity

```
class M
{ public: void display(){cout<<“ class M”;}
};
class N
{ public: void display(){cout<<“ class N”;}
};
class P : public M , public N
{ public: void display(){cout<<“ class P”;}
};
```

In this case function of derived class **overrides** the inherited function so when object of P calls display function it simply calls display of P. To invoke display of M & N we use scope resolution operator.

```
 int main ()
{ P p;
p.display();
p.M::display();
p.N::display();
p.P::display();
 return 0;
}
```

- Inheriting one class by more than one class leads to **Hierarchical inheritance.**



Hierarchical Inheritance

Example

```
#include<iostream.h>
#include<conio.h>
#include<string.h>
class college
{
protected:
char name[12];
void disp()
{
   cout<<"name of college
    is"<<name<<endl;}
public:
college()
{cout<<"constructor of college is
    called"<<endl;
strcpy (name, "PIET");
}};
```

```
class it_dept : public college
{
int faculty;
public:
void get_faculty()
{
cout<<"enter no of faculty
    members  in IT" <<endl ;
cin>>faculty; }
void display()
{ disp();  // now a protected
    member of it_dept
  cout<<"no of teachers in IT
    "<<faculty<<endl;
 }
};
```

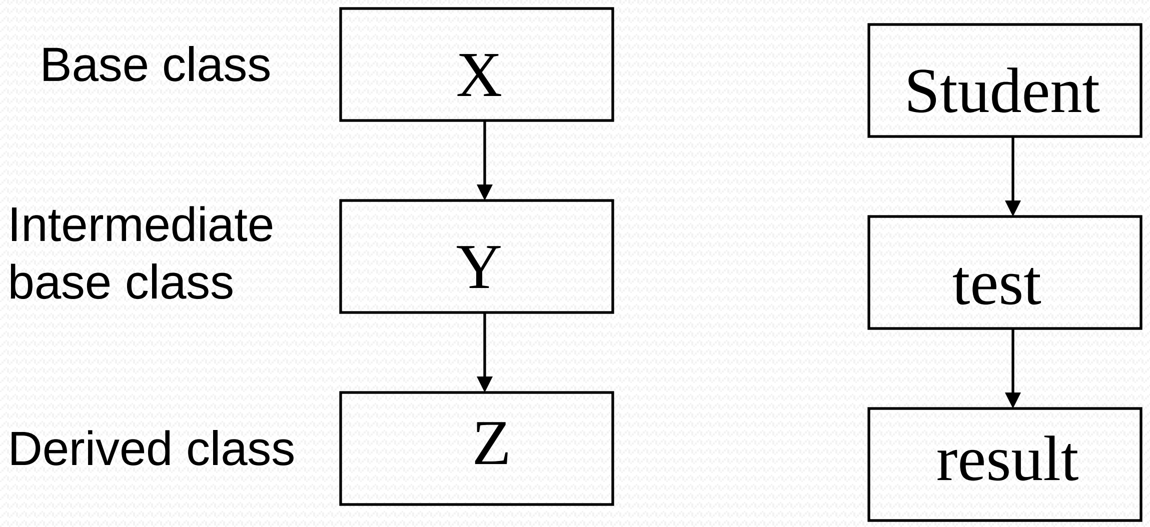
```
class ece_dept : public
   college
{
int faculty;
public:
void get_faculty()
{
cout<<"enter no of faculty
   members  in ECE"<<endl;
cin>>faculty;}
void display()
{ disp(); // now a protected
   member of ece_dept
  cout<<"no of teachers in
   ECE "<<faculty<<endl;}};
```

```
void main()
{   clrscr();
    it_dept C;
   ece_dept E;
   C.get_faculty();
   E.get_faculty();
    C.display();
    E.display();
}
```

OUTPUT

```
constructor of college is called
constructor of college is called
enter no of faculty members  in IT
12
enter no of faculty members  in ECE
36
name of college is PIET
no of teachers in IT 12
name of college is PIET
no of teachers in ECE 36
```

- deriving one class from other derived class leads to **multi level inheritance.**



Multilevel Inheritance

Example

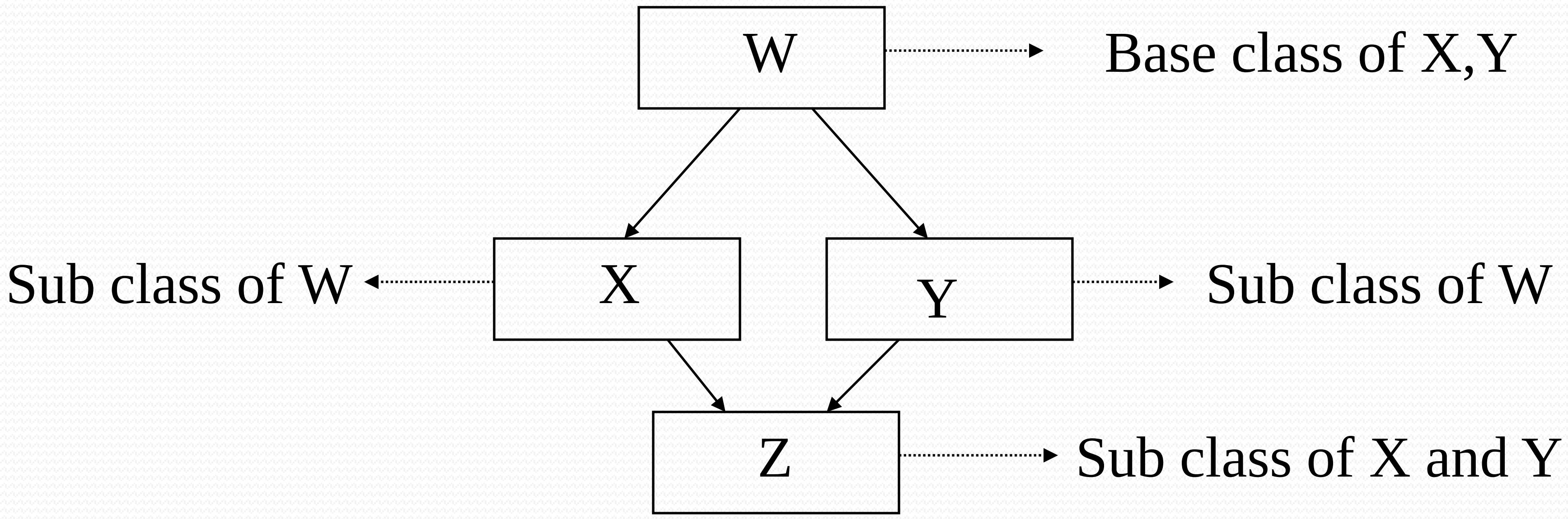
```
#include<iostream.h>
#include<conio.h>
#include<string.h>
class student
{
 protected:
 int roll_number;
 public:
void get_number(int a)
{
  roll_number=a;}
 void put_number()
{cout<<“Roll number
   is:"<<roll_number<<“\n”;
}};
```

```
class test : public student
{
protected:
float sub1,sub2;
public:
void get_marks(float x, float y)
{
sub1=x;
sub2=y;}
void put_marks()
{   cout<<“Marks in sub1=
    "<<sub1<<endl;
cout<<“Marks in sub2=
    "<<sub2<<endl;
 }
};
```

```
class result : public test
{
 float total;
public:
void display()
{
 total=sub1+sub2;
 put_number();
 put_marks();
 cout<<"Total ="<< total
   <<endl;
}};
```

```
int main()
{   clrscr();
    result Abc;
    Abc.get_number(111);
  Abc.get_marks(75.0,59.5);
Abc.display();
 return 0;
}
```

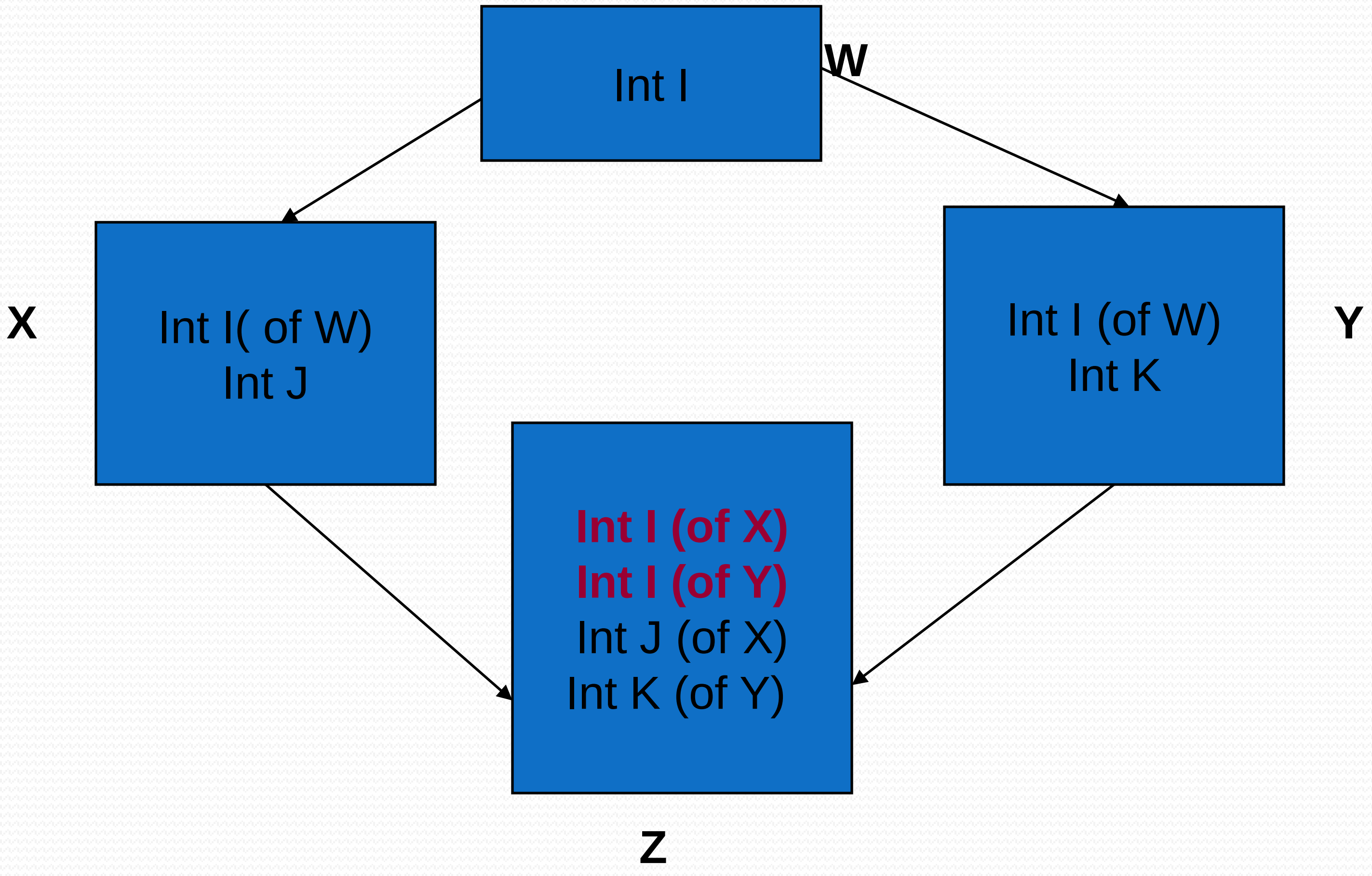
- When more than one inheritance is used, it is called **hybrid inheritance.**



Hybrid Inheritance

Virtual base class

- In case of hybrid inheritance shown, two copies of W-members are created in class Z.



```
#include"iostream.h"
#include"conio.h"
class W
{
public:
int i;
};
class X : public W
{ public:
  int j;
};
class Y: public W
{
public:
int k;
};
class Z: public X, public Y
{
public:
int sum;
};
```

```
int main (void)
{
clrscr();
Z obj;
obj.X::i=1;
obj.j=2;obj.k=3;
obj.sum=obj. X::i+ obj.j+obj.k;
// obj.i will be ambiguous
cout<<"the sum is "<<obj.sum;
return 0;
  }
```

OUTPUT

```
The sum is 6
```

If X and Y derives class W as a **virtual** class then only one copy of W-members will be inherited by X and Y.

```
#include"iostream.h"
#include"conio.h"
class W
{
public:
int i;
};
class X : virtual public W
{ public:
int j;
};
class Y: public virtual W
{
public:
int k;
};
class Z: public X, public Y
{
public:
int sum;
};

int main(void)
{
clrscr();
Z obj;
//obj.X::i=1;
obj.i=1;
obj.j=2;obj.k=3;
obj.sum= obj.i + obj.j + obj.k;
cout<<"the sum is "<<obj.sum;
return 0;
  }
```

Abstract Classes

- An abstract class is one that is not used to create objects.
- An abstract class should have atleast one pure virtual function
- An abstract class is only designed only to act as a base class(to be inherited by other classes).

```
class  ABC
{
Public:
Virtual void fn()=o;
//pure virutal function
};
```

```
Class XYZ : public ABC
{public:
void fun()
{
//body of function
};
};
```

Constructors in inheritance

- Constructors play an important role in initializing objects.
- In case of inheritance the base class constructors is first called of n den the derived class constructor is called.
- When base class contains **only default constructor** then derived class **may or may not have** the constructor.
- When base class contains a constructor with one or more arguments then it is **mandatory** for the derived class to have a constructor and pass the arguments to the base class constructors.

More about constructors….

- When an object of derived class is created

1. In case of **multi level inheritance** constructors are called in the order of their derivation.

```
 class derived1 : public base
{       };
class derived2: public derived1 {  };
```

That is base class constructor is called first, then constructor for derived1 class is called, then constructor for derived2 class is called.

- In case of **multiple inheritance**, constructors are called in the order specified from left to right .
- class derived: public base1,public base2

then

derived d;

will create an object of derived class by calling constructor for base classes base1 and then base2

Destructors in inheritance

- When an object of derived class is destroyed

1. In case of **multi level inheritance** ,destructors are called in the reverse order of their derivation

class derived : public base1,public base2{ };

Then destructor for derived is called then destructor for base2 is called then lastly destructor defined in base1 will be called.

Passing arguments to base class

- When an object of derived class is created, constructors of base class is also invoked.
- If initialization of only derived class variables are to be done then arguments to the constructor can be passed in usual way.
- But if base class constructor (invoked implicitly) also require some arguments then …….?

- Constructor of derived class should be defined as

Derived constructor (arg_list) :
  base1(arg_list1),base2(arg_list)……

{ body of the constructor }

- Arg_list for base classes will contain those arguments that are required by them,but are passed while calling derived class constructor.

```
#include<iostream.h>
#include<conio.h>
#include<string.h>
class college
{
protected:
int id ;
char name[12];
void disp()
{
cout<<"name of college is
    "<<name<<endl;
}
public:
college(int a)
{
id=a;
cout<<"constructor of college is
    called"<<endl;
strcpy (name,"PIET");
}
};
```

```
class  it_dept : public college
{
int faculty;

public:

it_dept (int a, int b) : college (a)

{  faculty=b; }

void display()
{ disp();  // now a protected member of
                 it_dept
  cout<<"initialized college id is "<<id;
  cout<<"no of teachers in IT "
     <<faculty<<endl;
 } };
int main()
{   clrscr();
    it_dept C(10,12);
    C.display();
    return 0;
}
```

Argument for base constructor

OUTPUT

- constructor of base college is called
- initialized college id is 10
- name of college is PIET
- no of teachers in CSE 12

- The constructors for virtual base classes are invoked before any non virtual base classes.

METHOD OF INHERITANCE	ORDER OF EXECUTION
class B : public A	
{	
};	
class A : public B, public C	
{	
};	
class A : public B, virtual public C	

Initializing the class objects

```
 constructor (arglist) : initialization-section
{
 assignment-section
}
Example: class ABC{ int a;
 public : ABC(int x)
{ a=x;
 cout<< “” ABC Constructor & values”<< a;}
};
class XYZ
{ int  b,c;
 public: XYZ(int i, int j): a(i),b(2*j) { c=j;
 cout<< “XYZ Constructor & values”<<b<<c;}
};
 main()
{ XYZ x(2,3,4)
}
```

Member Classes : Nesting Of classes

```
 class alpha { };
 class beta { };
 class gamma
{ alpha a;
 beta b;
.........
};
```

All objects of gamma class will contain the objects a,b; This kind of relationship is called **containership** or **nesting.**

A class with the main purpose of holding objects is commonly called a ***container***.

Association, Aggregation & Composition

Association is a relationship where all objects have their own lifecycle and there is no owner.

E.g. Teacher & Student

Aggregation is a specialized form of Association in which one object is a part of another.

A aggregates B =
B is part of A, but their lifetimes may be different

e.g. cars and wheels, engine, etc

Composition is a specialised form of Aggregation in which one object is an integral part of another.

A composes B =
B is part of A, and their lifetimes are the same

e.g. person and brain, lung, etc

Aggregation: Classes within classes

Inheritance is often called a "kind of" relationship e.g. if a class B is derived by inheritance from a class A, we can say that "B is a *kind of A.*"

Aggregation is called a "has a" relationship. Aggregation is also called a "part-whole" relationship e.g. the book is part of the library.

Aggregation may occur when one object is an attribute of another e.g. an object of class A is an attribute of class B:

```
class A
{
};
class B
{
    A objA; // define objA as an object of class A
};
```

Composition: A Stronger Aggregation

It has all the characteristics of aggregation, plus two more:

• The part may belong to only one whole.

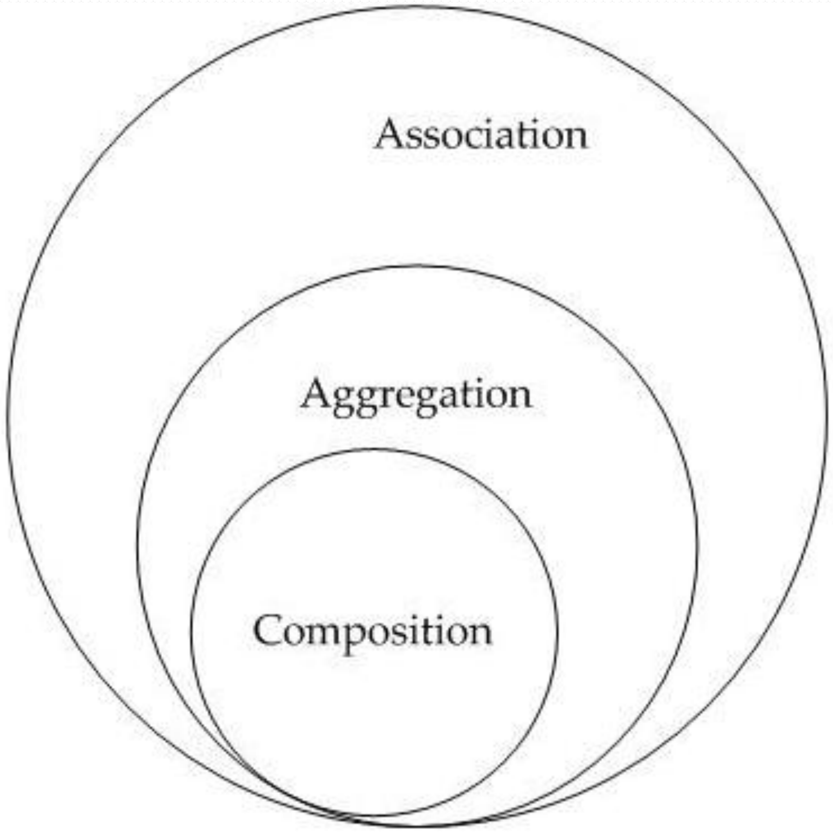
• The lifetime of the part is the same as the lifetime of the whole.

Aggregation is a "has a" relationship, while composition is a "consists of" relationship.

Even a single object can be related to a class by composition e.g. in a car there is only one engine.

Association, Aggregation & Composition

In both aggregation and composition **object of one class "owns" object of another class**. In **Composition,** the object of class that is owned by the object of it's owning class **cannot live on it's own**(Also called "death relationship"). It will always live as a part of it's owning object where as in **Aggregation** the dependent object is **standalone** and can exist even if the object of owning class is dead.



Classification Hierarchies

By organising classes into Classification Hierarchies we can:

- add extra dimensions to encapsulation by grouping ADT's
- and enable classes to inherit from other classes
- thus extending the attributes and methods of the class

The inheriting class may then also add extra functionality and attributes to produce a new object

Assignment

Q1. What are the mode of inheritance?

Q2. What is function overriding and how they differ from function overloading?

Q3. When would we use inheritance?

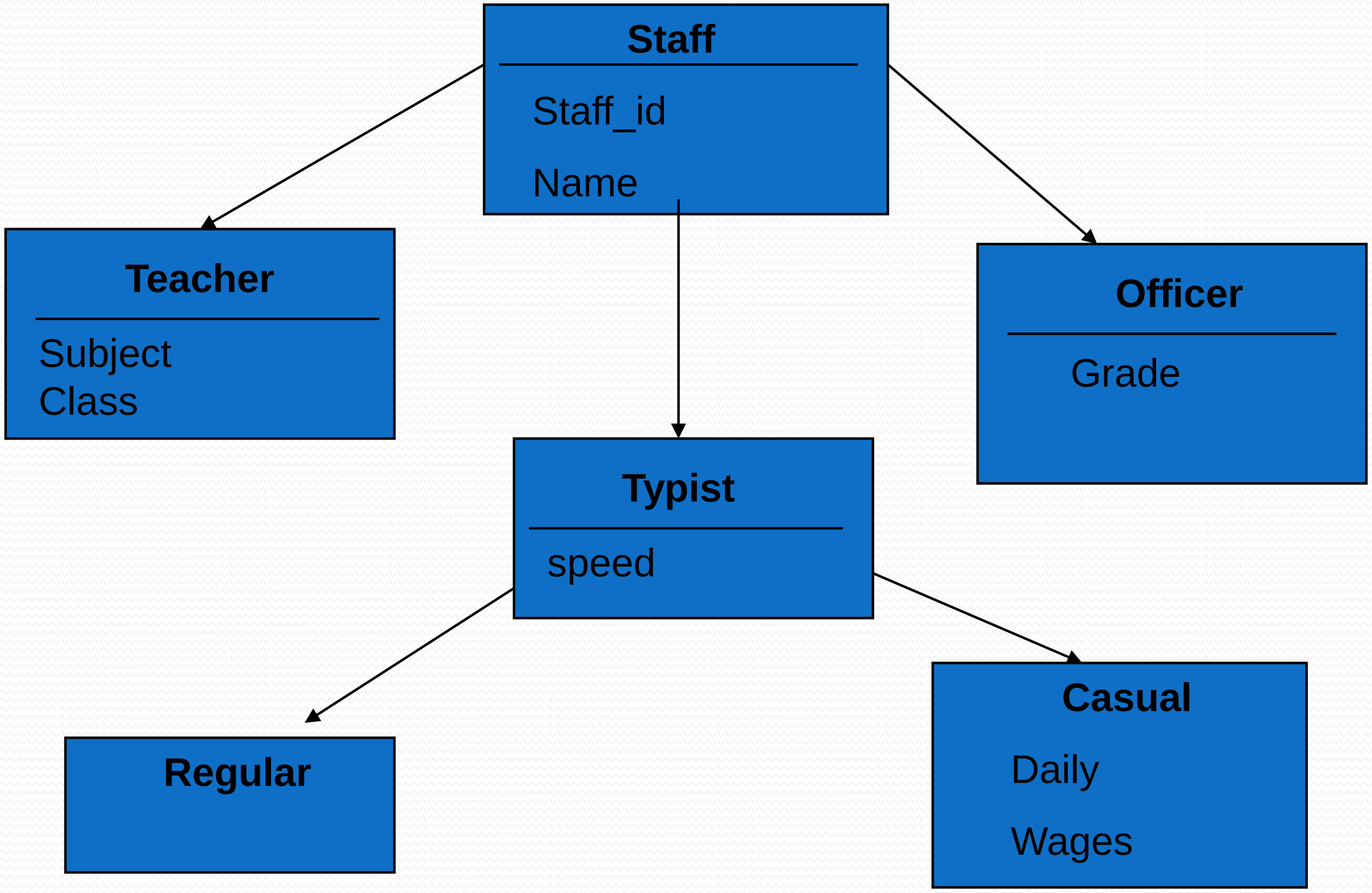
Q4. Why derived class cant access private things from base class.

Q5. When do we make a class virtual?

Q6. How constructors and destructors are work in inheritance?

Q7. What are abstract classes?With example

Q8. Write a program using inheritance which is shown in diagram.


Staff
Staff_id
Name
Teacher
Subject
Class
Officer
Grade
Typist
speed
Regular
Casual
Daily
Wages