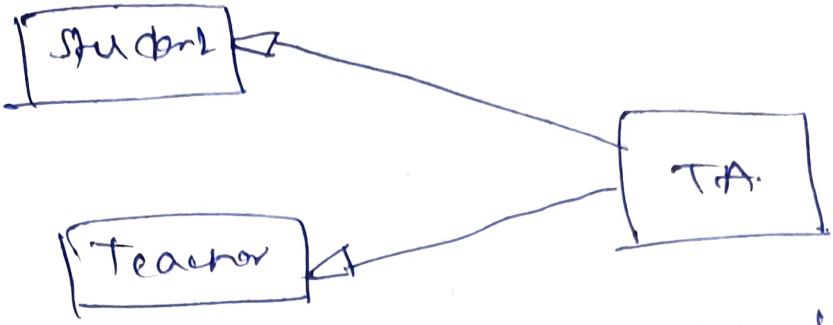
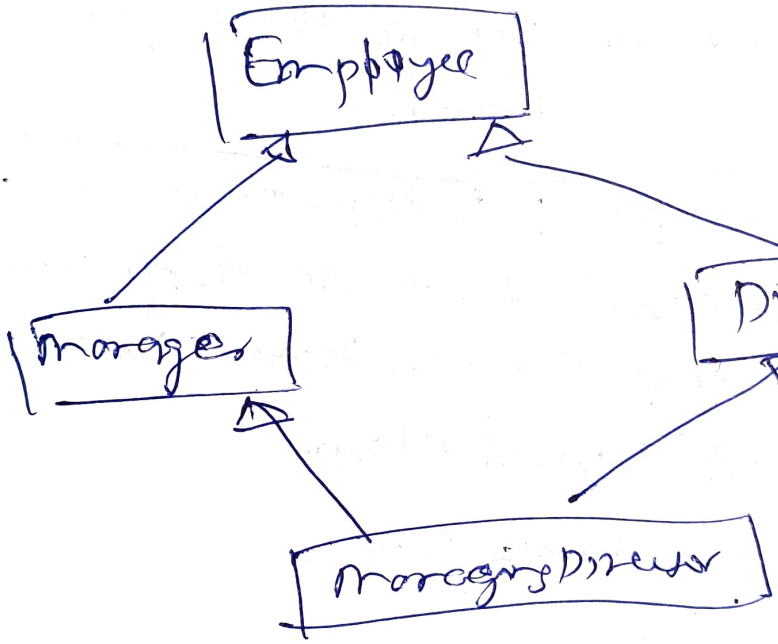
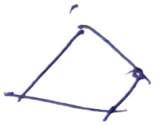
Multiple inheritance

TA is A student & TA is A Teacher.



class student;
class Teacher;
~~Employee~~
class TA : public student,
public Teacher;



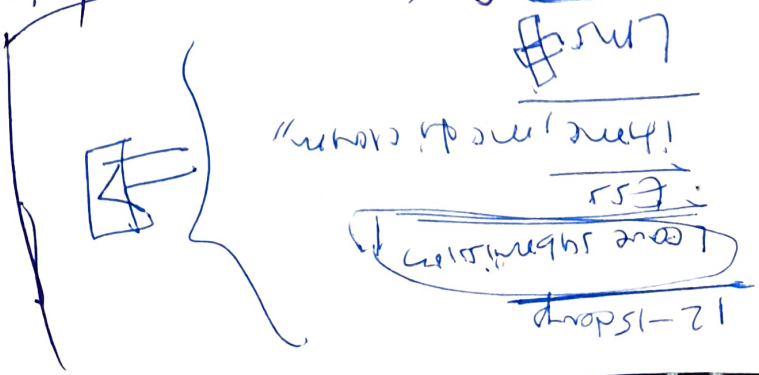


Diamond problem

class Employee;

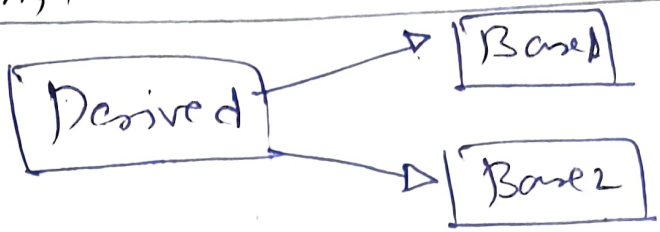
class Manager : public Employee;

class Director; ~~public Employee~~



(243) class managingDirector: public manager,
public Director;

Multiple inheritance: semantics



Have two or more Base class.

public and private inheritance may be mixed

class Base1;

class Base2;

```
class Derived : public Base1, public
                Base2.
```

private inheritance: Is implemented as i.e is a ~~Base class~~ if we are doing private inheritance then it means Base class already implemented Derived class.

* Derived is a Base1, Base2;

Data members

- Derived class inherits all Data members of all Base classes
- Derived class may add data members of its own

Member function.

* Derived class inherits all member fn of all Base classes.

* Derived class may override a member fn of any Base class by redefining it with the same signature.

* Derived class may overload a member fn of any Base class by redefining it with the same name; but diff signature.

Access specification

* Derived class cannot access private members of any Base class.
* Derived class can access protected member of any Base class.

Construction-destruction

(1) A constructor of the Derived class must first call all constructors of the Base classes to construct the Base class instances of the Derived class. — Base class constructors are called in listing order.

(2) The destructor of the derived class must call the destructor of the Base classes to destruct the Base class instances of the derived class.

(LU5) Data members and Object layout.

* Derived LSA Base1, Base2

Data members

(i) Derived class inherits all Data members of all Base classes

(ii) Derived class may add data member of its own.

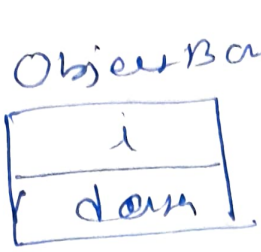
Object layout

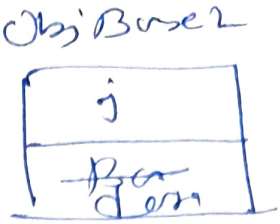
* Derived class layout contains instances of each Base class.
* Further Derived class layout will have data members of its own.
* C++ does not guarantee the relative position of the Base class instances and Derived class members.

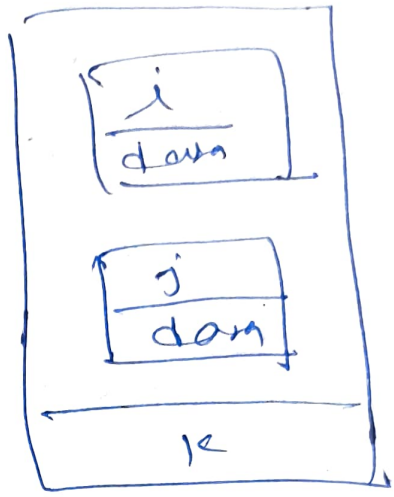
Ex:

```
class Base1 { protected:
        int i;
        int data;
    public:
};
class Base2 { protected:
        int j;
        int data
    public:
};
class Derived: public Base1, public
    Base2 {
```

int k;
public:
};







* Both Base class have data members with same name so refer them as

```
Base1:: data
Base2:: data
```

Derived d

```
d. Base1:: data;
d. Base2:: data;
```

multiple inheritance member fn overrides and overloads

* Derived class doesn't inherit the static member fn of any Base class
* Derived class doesn't inherit the friend fn of any Base class.

Ex-1

```
class Base1 { protected:
    int i;
    int data;
public: Base1(int a, int b) : i(a)
    data(b);
    void f(int) { cout << "Base1::f(int)
        "; }
    void g()
    { cout << "Base1::g()" << endl; }
};
class Base2 { protected:
    int j;
    int data;
public: Base2(int a, int b) : j(a);
        data(b);
    void h(int)
    { cout << "Base2::h(int)" << endl;
    }
};
class Derived: public Base1, public Base2
{   int k;
public: Derived(int x, int y, int z,
        int v, int z);
    void f(int)// override.
    { cout << "Derived::f(int)" << endl;
    }
```

```
void h(string) // Overload.
{ cout << " Derived :: h(string)" << endl;
}
void e(char)
{ cout << " Derived :: e(char)" <<
  endl; }
};
Derived c(1,2,3,4,5)
c.f(5) // Derived :: f(int)
c.g() → // Base1 :: g()
c.h("ppd"); // Derived :: h(string)
c.e('a') // Derived :: e(char)
```

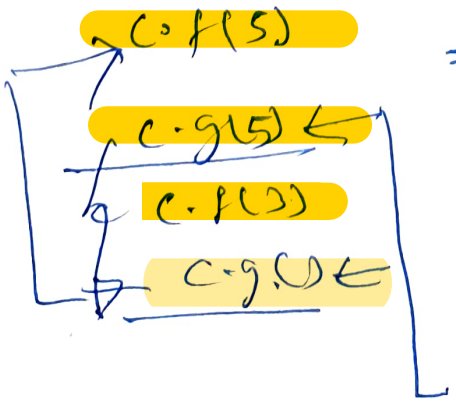
Multiple inheritance: (can be used only for non poly morphic types)

member fn using for Name resolution.

Ambiguous calls

```
class Base1
{ int i, data;
  public:
  Base1(int a, int b) : i(a), data(b);
  { }
  void f(int)
  { cout << " Base1 :: f(int) " << endl;
  }
  void g()
  { cout << " Base1 :: g()" << endl;
  }
};
```

```
class Base2
{   protected: int j;
    int data;
public: Base2()
    Base2(int a, int b) : j(a), data(b) { }
    void f(int)
    { cout << "In Base2::f(int)" << endl;
    }
    void g(int)
    { cout << "In Base2::g(int)" << endl;
    }
};
class Derived: public Base1, public Base2
{   public: Derived(int a, int x, int y,
                    int v, int z)
};
Derived c(1,2,3,4,5)
```



Question which copy of f will be included in Derived class

Here overload ambiguous resolution does not work properly

(250) Note: Overload resolution does not work bet diff namespace. It works bet same namespace.

How to resolve the previous issue

we have to restrict single copy of f or g from Base1 or Base2 in derived class like:

class Derived: public Base1, public Base2

{

public: Derived (int x, int y, int z, int u, int v, int w);

using Base1::f; // inherits f from Base1
using Base2::g;

};

Derived d(1,2,3,4,5)

d.f(5); // Base1::f(int)

d.g(5); // Base2::g(int)

d.Base2::f(3); // Base2::f(int)

d.Base1::g(); // Base1::g()

* If two base class have same function irrespective of same name irrespective of signature from the derived class must use using to access those fn.

Key

Constructor - destructor

* Derived class inherits all constructors and destructors of Base classes (but in a different scenario)
* Derived class cannot override or overload a constructor or a destructor of any Base class.

Ex.

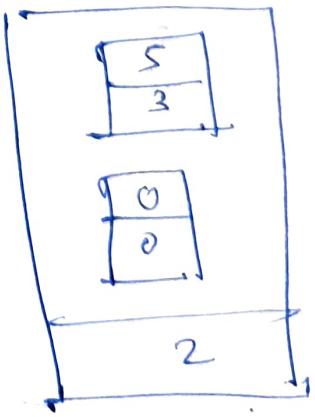
```
class Base1 { protected: int i, data;
        public: Base1(int a, int b) :
        i(a), data(b) { cout<<"Base1::
            Base1()"<<endl; }
        ~Base1()
        { cout<<"Base1::~Base1()"<<endl;
        }
};
class Base2 { protected: int j, data;
    public:
        Base2(int a=0, int b=0) : j(a),
        data(b)
        { cout<<"Base2::Base2()"<<endl;
        }
        ~Base2()
        { cout<<"Base2::~Base2()<<
                endl;
        }
};
```

(252)

```
class Derived: public Base1, public Base2
{
    int k;
public:
    Derived(int a, int y, int z);
    Base1(x,y), k(z)
    { cout << "Derived::Derived()" <<
                endl;
    }
// ~~Base1~~ ex Base1::Base1
explicit, Base2::Base2
default
    ~Derived()
    { cout << "Derived::~Derived()
                " << endl;
    }
};
```

~~Derived~~

derived d(5, 3, 2);



O/p: Base1::Base1()

Base2::Base2()

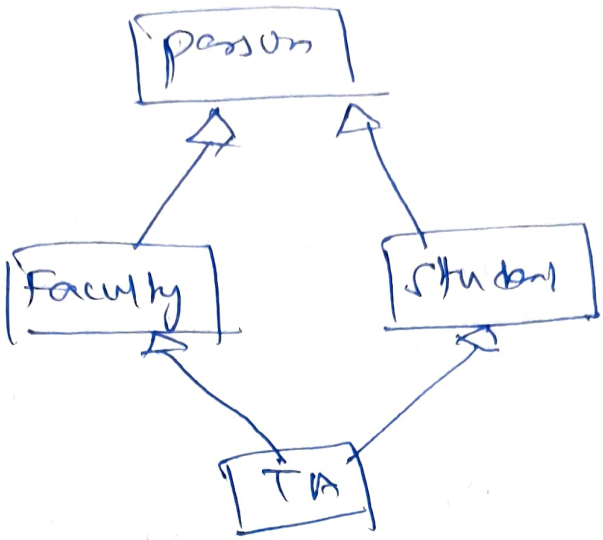
Derived::Derived()

Derived::~Derived()

Base2::~Base2()

Base1::~Base1()

Multiple inheritance - diamond problem




```
class person;
class student : public Person;
class Faculty : public person;
class TA : public student, public Faculty
```

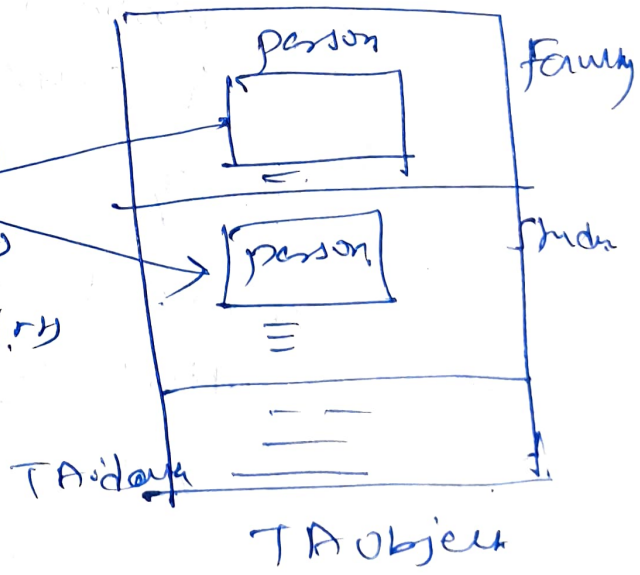
Ex

```
class Person
{   public: Person(int x)
    { cout<<"Person() : Person(int)"<<
                                endl;
    }
};
class Faculty : public Person
{   public: Faculty(int x) : Person(x)
    { cout<<"Faculty(int) : Person(x)"<<
                                endl;
    }
};
class Student : public Person
{
```

(254)

```
public: student(int a) : Person(a)
{ cout << "student:: student(int)" <<
                                endl;
}
};
class TA: public Faculty, public
        student {
public: TA(int n) : student(n),
        Faculty(n) {
    cout << "TA :: TA(int)" << endl;
}
};
TA ta(30);
```

O/p: Person :: person(int)
Faculty :: Faculty(int)
Person :: person(int)
student :: student(int)
TA :: TA(int)



Two instances of base class Object

Multiple instances of the root class (or) diamond class into the derived class Object. This is known as diamond problem

How to resolve this

↓

Using virtual inheritance

```
class Person {
  public: Person(int x)
          { cout<<" Person:: Person(int)"<<
                  endl;
          }
          Person() // Default constr for virtual
                   inheritance
          { cout<<"Person:: Person()"<<
                  endl;
          }
};
```

→ virtual Base class

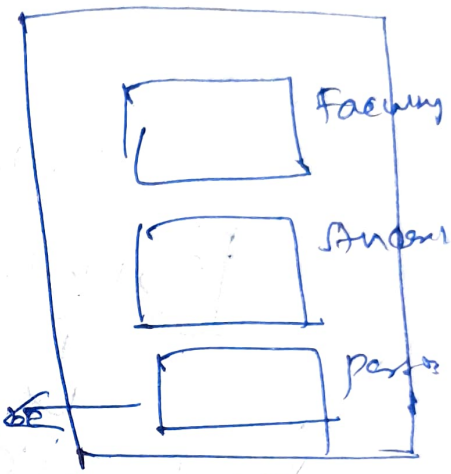
```
class Faculty: virtual public Person
{
  public: Faculty(int x) : Person(x)
          { cout<<" Faculty:: Faculty(int)"
            <<endl;
          }
};
class Student: virtual public Person
{
  public: Student(int x) : Person(x)
          { cout<<" Student:: Student(int)"<<endl;}
};
```

(256)

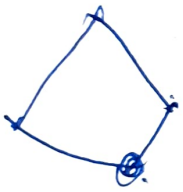
```
class TA: public Faculty, public Student
{ public: TA(int x): Student(x),
    Faculty(x) { cout<<"TA::TA("
    <<x<<")"<<endl;
  }
};
TA ta(30)
```

* one common person instance for both Fac and Stud

O/P: Person::Person
Faculty::Faculty(int)
Student::Student(int)
TA::TA(int)



this will be constructed through default constructor of Base class



~~*~~ if instance is virtual then it knows that some other class might be constructed so ~~derived class~~ there could be multiple Base class so Faculty will not construct its Base class.

~~*~~ Virtual Base class: they will not directly construct Base class portion Base class it will be common in derived class.

(25)

what if we want to pass the parameter to root class.

Then we have to explicitly call the constructor of root class in leaf class of diamond(TP) with parameter like.

```
class TA : public Faculty, public
    Student {
public:
    TA(int x) : Student(x), Faculty(x
    {
        cout << "TA::TA(int)"
        << endl;
    }
};
```

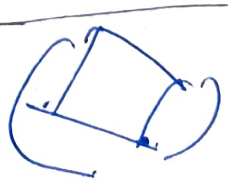
Person(x) → calling parameterised constructor of person

```
TA ta(30
```

O/P: person::person(int)
Faculty::Faculty(int)
Student::Student(int)
TA::TA(int)

Ambiguity in multiple inheritance with

dyn diamond structure



(25)

```
class Person
{ public: Person(int a)
  { cout<<"Person::Person("<<a<<")"<<
    endl;
  }
  Person()
  { cout<<"Person::Person()"<<
    endl;
  }
  virtual ~Person();
  virtual void teach()=0;
};
class Faculty: virtual public Person
{ public: Faculty(int a): Person(a)
  { cout<<"Faculty::Faculty("<<a<<")"<<
    endl;
  }
  virtual void teach();
};
class Student: public virtual public
    Person
{ public: Student(int a): Person(a)
  { cout<<"Student::Student("<<a<<")"<<
    endl;
  }
};
```

(259)

```
class TA: public Faculty, public student
{ public:
    TA(int x): student(x), Faculty(x)
    { cout << "TA :: TA(int)" << endl;
    }
};
// TA ta(30) // No allowed. since
in the absence of TA :: teach
which of teach should be inherited.
```

This cannot be resolved by using like non polymorphic type. It can be resolved by overriding teach() teach() in TA like

```
class TA: public Faculty, public student
{ public:
    TA(int x): student(x), Faculty(x)
    { cout << "TA :: TA(int)" << endl;
    }
    virtual void teach()
};
```

Now TA ta(30) // ✓ OK

* exercise from slides (260)

Exception Handling

Wherefore exceptions?

conditions that arise

* infrequently and unexpectedly
* generally betray a program error
* Require a considered programmatic Response
* Runtime Anomalies — Yes, but not necessary

leading to

* crippling the program
* may pull the entire system down.
* Defensive technique
  * crashing exception versus Tangled Design and Code

Exception causes

* unexpected system state
  * Exhaustion of Resources
    * Low free store mem.
    * Low disk space
  * putting to full stop.
* external events
  * control + C
  * Socket error
* Logical errors

* pop from empty stack.
* Resume error like from I/O Read/ write.

* Run time error
  * Arithmetic overflow/underflow
  * Out of Range.
* Undefined opn
  * Devi division by zero

Exception Handling:

* Exception handling is a mechanism that separates the detection and handling of circumstantial Exception flow from Normal Flow
* Current state saved in a predefined loc
* Excep execution switched to a pre-defined handler

Types of exception

* Asynchronous exception
  * Exception that come unexpectedly
  * Example - or interrupt in a pgm
  * Take control away from the executing Thread context to a context that is diff from the. that which caused the exception

Synchronous exception

* planned exception
* Handled in an organized manner.
* The most common type of synch exception is implemented as a throw

Exception stages

```
int f()
{
    int error                          low mem
    if(error) /* stage 1: error occurred   create
        return -1 // stage 2: generate exception Object
                                          Report
}

int main(void) {
    int(f() != 0) /* stage 3: detect      Detect
                     exception
    {
        /* stage 4: Handle exception      Handle
    }
    /* stage 5: recover */                Recover
}
```

Stage 1: error Evidence

* Synchronous (s/w) logical error
* Asynchronous (h/w) Interrupt (s/w Interrupt)

Stage 2: Create Object and Raise exception.

(26) * An exception Object can be of any complete type

* An int to a full blown C++ class object

(3) Detect exception

* polling slow test
* Notification control (stack) Adjustment.

(4) Handle exception

* Ignore; hope someone else handles it that is do not catch
* Act; but allows others to handle it afterwards, that is catch, handle and Re throw
* Own; take complete ownership that is, catch and handle.

(5) Recover from exception

* Continue execution; If handled inside the pgm.
* Abort execution; If Handled outside the pgm

Support for exception in C

* C lang does not support anything keeping error handling in mind but we can use some of lang features to handle errors in more structured way

* Language feature
  * Return value and parameters
  * Local goto
* Standard lib support
  * Global variable
  * Abnormal termination
  * Conditional ''
  * Non local goto
  * Signals

Global variable mechanism

* Use a designated global error variable
  * set it on error
  * poll/check it for detection
* Standard lib CW mechanism

<errno.h> <cerrno>

Ex: #include <errno.h>

```
main()
{
    double x, y, result;
    errno = 0;
    result = pow(x, y);
    if (errno == EDOM)
        printf("domain error only
        par \n");
    else if (errno == ERANGE)
        printf("range error in
        result \n");
    else
        printf("x to the y = %d\n",
                (int)(result));
    return 0;
}
```

Exception Handling in C++

Exceptions:

(i) Separate error handling code from ordinary code

(ii) Log. mechanism rather than OS the library

(iii) Compiler for tracking automatic variables

(iv) Schemes for destruction of dynamic mem

(v) less overhead for systems

(vi) exceptions propagation from the deepest of levels

(vii) various exceptions handle by style handler

try-throw-catch

```
class UserExcp: public exception { };

void fcn
{
    ...
    UserExcp ex("From fcn");   — create
    throw ex;   — (It will throw then the control give to catch clause)
    return;
}
```