Type of Inheritance

Hierarchal Inheritance

$$\begin{tikzcd}
A \arrow[d, dash] & \\
B \arrow[d, dash] & \\
C \arrow[d, dash] \arrow[dr, dash] & \\
D & E
\end{tikzcd}$$

Multiple Inheritance

$$\begin{tikzcd}
A \arrow[dr, dash] & & B \arrow[dl, dash] \\
& C &
\end{tikzcd}$$

Hybrid Inheritance

$$\begin{tikzcd}
& A \arrow[d, dash] & & \\
& B \arrow[dl, dash] \arrow[dr, dash] & & E \arrow[d, dash] \\
{} & & {} & {}
\end{tikzcd}$$

$$\begin{tikzcd} D & & C & \\ & & & F \arrow[ul, dash] \arrow[u, dash] \end{tikzcd}$$

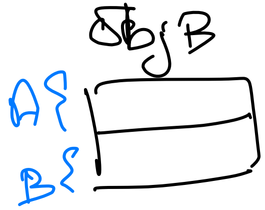
Public C : $\underbrace{\text{Public A}}_{\text{first base class}}$, $\underbrace{\text{private B}}_{\text{second base class}}$

{

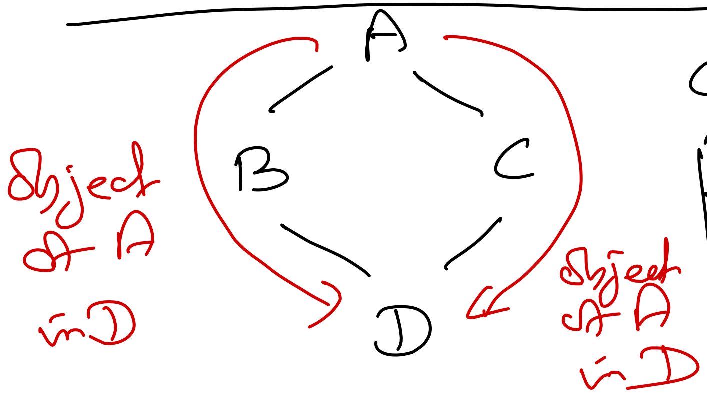
;

}

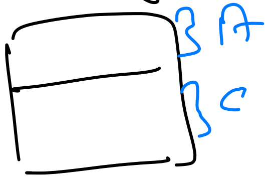
B objB;



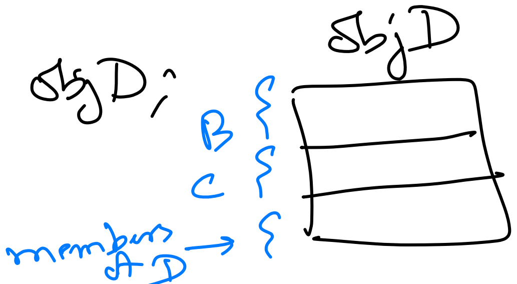
Diamond Shape Problem

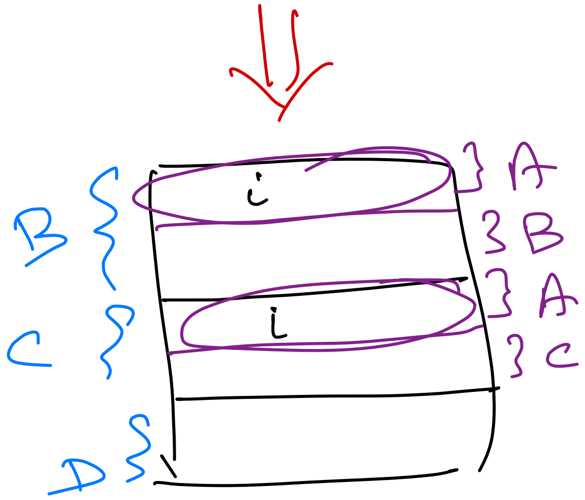


C objC;



D objD;



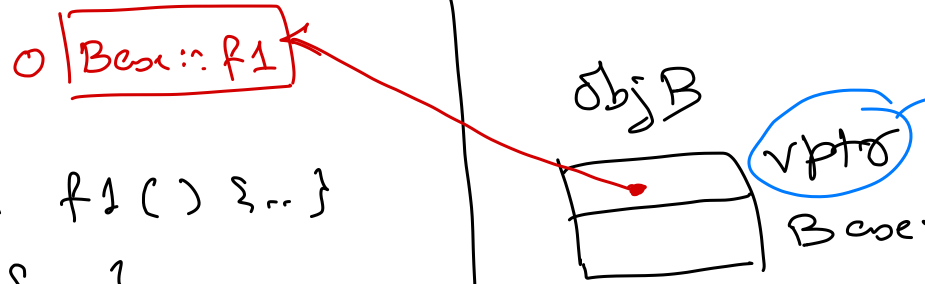


⇐ Object A class D

objD.i = 10;

```
class Base {
    int i;
public:
    virtual void f1() {...}
    void f2() {...}
};

class Derived : public Base {
    int j;
  public:
    void f1() {...}
    void f2() {...}
};
```

Base::vtable



Derived::vtable

0	Derived::f1

Base objB;

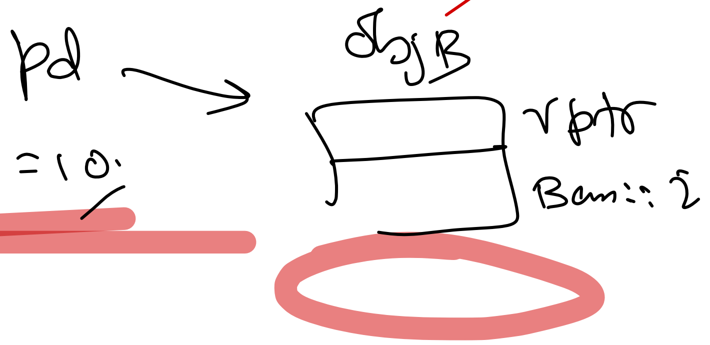
Per object of a class that has virtual function

```
Base *pb;
pb→f1();
```

Derived * pd;

~~pd = &objB;~~

pd -> f1();



pd -> j = 10;

Abstract class

```
class Base {
    int i;
public:
```

class is abstract class
⇓
object cannot be created.

```
    virtual void f1() = 0;
};

class Derived : public Base {

};
```

$\boxed{= 0;}$ → Pure virtual function

⇓

function body is not required.

Also abstract class as it do not implement inherited pure virtual function.

<u>Interface</u>

Interface

||

Animals

├ getName()

└ getType()

```
class Animals {
    public:
        virtual const char * getName() const =0;
        virtual const char * getType() const =0;

};
```

← class having only pure virtual function. No data members.

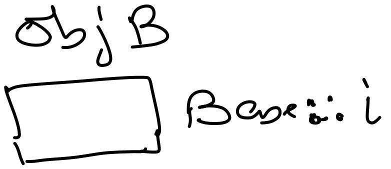
```
class Base {
      int i;
};
```

```
class Derived1 : public Base {
        int j;
};
```

```
class Derived2 : public Base {
        int k;
};
```

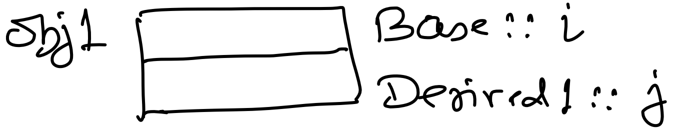
```
class Derived : public Derived1,
                public Derived2 {
      int l;
};
```

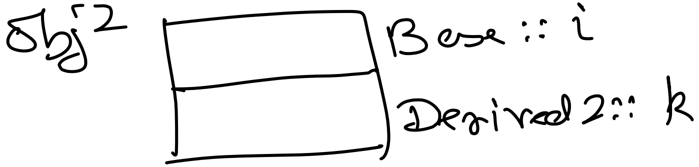
```
Base objB;
```



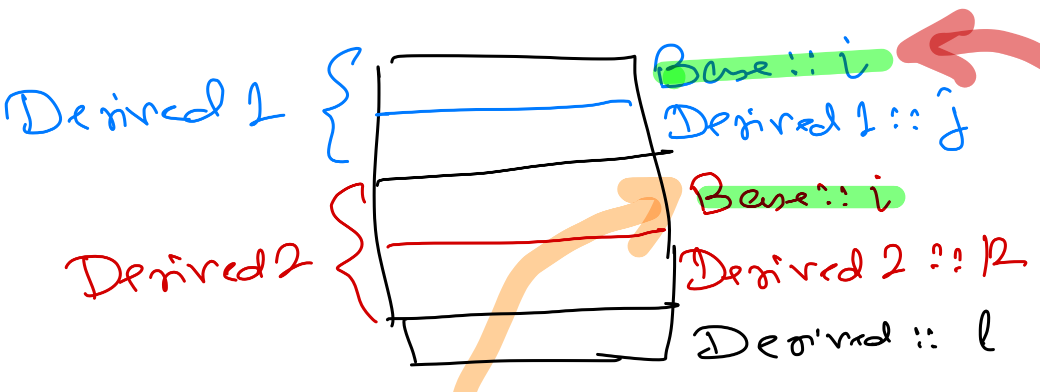
```
Derived2 obj2
```

```
Derived1 obj1;
```





Derived objD;



objD.i = 10; ✗

objD.Base::i = 10; ✗

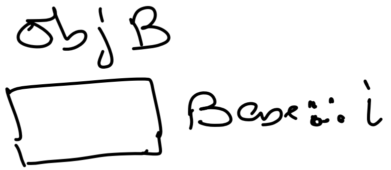
objD.Derived1::i = 10; ✓

objD.Derived2::i = 100; ✓

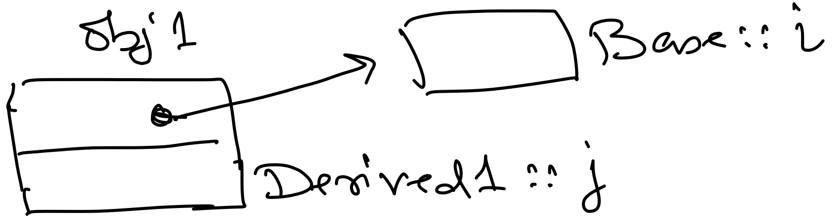
```
class Base {
      int i;
};
```

```
class Derived1 : virtual public Base {
      int j;
};

class Derived2 : virtual public Base {
      int k;
};

class Derived : public Derived1,
                public Derived2 {
      int l;
};
```

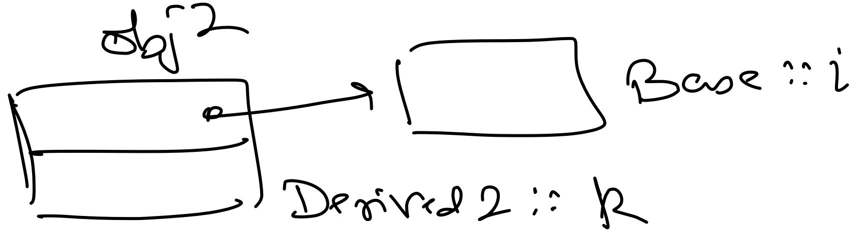
```
Base objB;
```



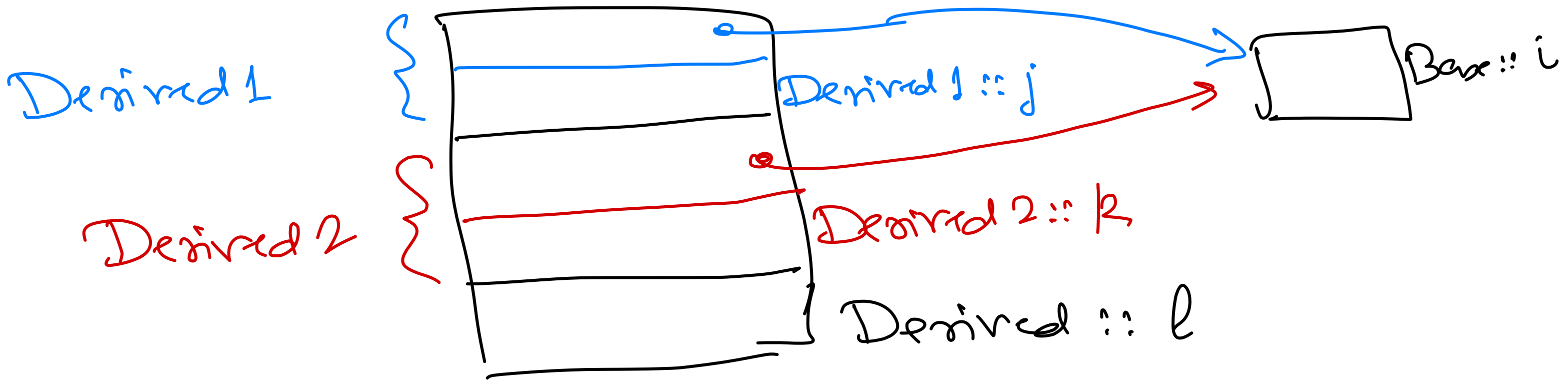
```
Derived1 obj1;
```



Derived2 obj2



Derived objD



TEMPLATES

```
void swap(int& a, int &b) {
    int t;

    t = a; a = b; b = t;
}
void swap(float& a, float &b) {
    float t;

    t = a; a = b; b = t;
}

int main() {
    int x = 10, y = 20;
    std::cout << "Before swap x = " << x << ", y = " << y << "\n";
    swap(x, y);
    std::cout << "After swap  x = " << x << ", y = " << y << "\n";

    float f1 = 1.5, f2 = 2.5;
    std::cout << "Before swap f1 = " << f1 << ", f2 = " << f2 << "\n";
    swap(f1, f2);
    std::cout << "After swap  f1 = " << f1 << ", f2 = " << f2 << "\n";

    return 0;
}
```

```
template<class T>
void swap(T& a, T& b) {
    std::cout << "Calling template function\n";

    T t;

    t = a; a = b; b = t;
}

int main() {
    int x = 10, y = 20;
    std::cout << "\nBefore swap x = " << x << ", y = " << y << "\n";
    swap(x, y);
    std::cout << "After swap  x = " << x << ", y = " << y << "\n";

    float f1 = 1.5, f2 = 2.5;
    std::cout << "\nBefore swap f1 = " << f1 << ", f2 = " << f2 << "\n";
    swap(f1, f2);
    std::cout << "After swap  f1 = " << f1 << ", f2 = " << f2 << "\n";

    return 0;
}
```

← generic type T ⇒ each generic type must be used at least once in function argument list.

```
template<class T>
void swap(T& a, T& b) {
    std::cout << "Calling template function\n";

    T t;

    t = a; a = b; b = t;
}
```

Template specialization

```
template<>
void swap(float& a, float& b) {
    std::cout << "Calling template specialization for float\n";

    float t;

    t = a; a = b; b = t;
}

int main() {
    int x = 10, y = 20;
    std::cout << "\nBefore swap x = " << x << ", y = " << y << "\n";
    swap(x, y);
    std::cout << "After swap  x = " << x << ", y = " << y << "\n";

    float f1 = 1.5, f2 = 2.5;
    std::cout << "\nBefore swap f1 = " << f1 << ", f2 = " << f2 << "\n";
    swap(f1, f2);
    std::cout << "After swap  f1 = " << f1 << ", f2 = " << f2 << "\n";

    return 0;
}
```