Objects in C++

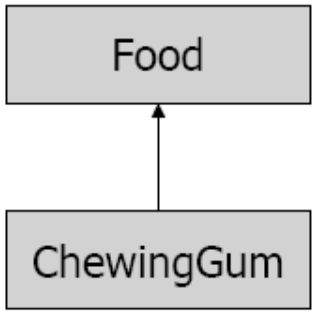
Inheritance

C++ Object System

- Object-oriented features
  1. Classes and Data Abstraction
  2. Encapsulation
  3. Inheritance
     - Single and multiple inheritance
     - Public and private base classes
  4. Objects, with dynamic lookup of virtual functions
  5. Subtyping
     - Tied to inheritance mechanism

Inheritance (1)

The ability to reuse the definition of one kind of object to define another kind of object.

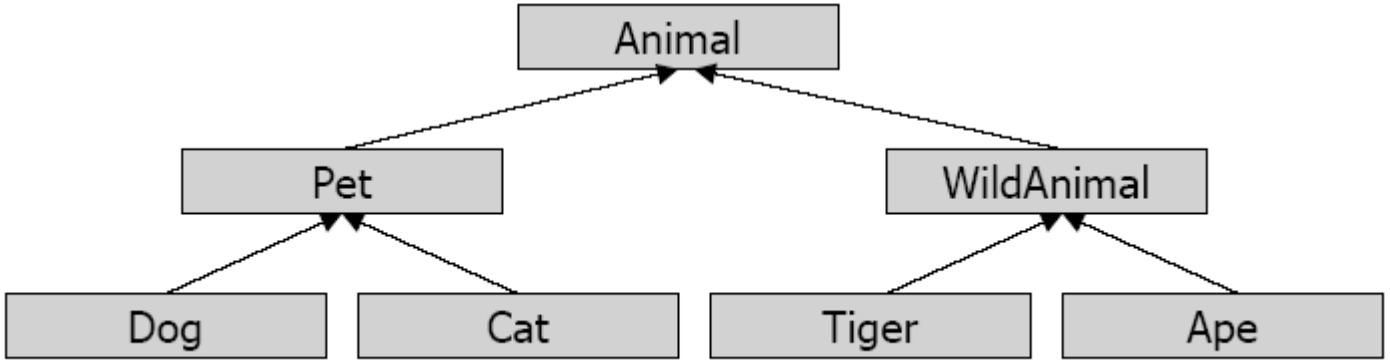



```
class ChewingGum : public Food {
  // ...
};
```

- ChewingGum inherits
  - all public class members (full access)
  - all protected class members (full access)
  - all private class members (no direct access)

Class hierarchies

- each derived class can act as a base class for further derivation



Constructors/destructors and inheritance (1)

- constructors
  - require calling the base class constructor
  - if arguments are mandatory, they have to be provided

```
class Manager : public Employee {
  public:
   Manager(const std::string& name,
                const short level)
         : Employee(name), level_(level) {
   }
  private:
   short level_;
};
```

Constructors/destructors and inheritance (2)

- destructors
  - always make destructors virtual in base classes
  - there might be cleanup work to be done in derived classes

```
class Employee {
  //...
  public:
   //...
   virtual ~Employee() {}
};
```

Public, private, protected inheritance

```
class CD: public CB{…}
class CD: private CB{…} or class CD: CB{…}
class CD: protected CB{…}
```

VISIBILITA'	TIPO di EREDITARIETA' public	TIPO di EREDITARIETA' protectet	TIPO di EREDITARIETA' private
public	public	protected	private
protected	protected	protected	private
private	private	private	private

Private inheritance –publicize members

```
class CBase {
   int x;
public:
   int y;
   voif f();
   void f(int);
};
class CDerivata: Cbase{ // private inheritance
public:
   CBase::y; // y is turned in pubblic
   CBase::x; // ERROR. Not allowed!! x is private
   CBase::f; // Both overloaded members exposed
};
```

- Thus, **private** inheritance is useful if you want to hide part of the functionality of the base class.
- In the presence of private inheritance, a subclass in not a subtype

Multiple inheritance

- simply extend the inheritance definition

```
class MobileAgentCommand :
      public Command,
      public Serializable,
      public PersistentObject {

   ...
};
```

However, multiple inheritance introduces a number of possibilities for ambiguity!

Redefining (1)

```
class X {
  int i;
  public:
  X() { i = 0; }
  void set(int ii) { i = ii; }
  int  permute() { return i = i * 47; }
};

class Y : public X {
 int i; // Different from X's i
 public:
 Y() { i = 0; }
 int change() {
   i = permute(); // Different name call
   return i;
 }
 void set(int ii) {  // redefining
   i = ii;
   X::set(ii); // Same-name function call
 }
};
```

Redefining (2)

- *Redefining* for ordinary member functions and *overriding* when the base class member function is a **virtual** function
- *Redefining* **produces an overloaded function, with code selection done at compile time** through the operator *class_name::*
- **Virtual** functions are the normal case and will be covered in detail later
- **Polymorphism** is implemented in C++ with the **dynamic lookup of virtual functions**

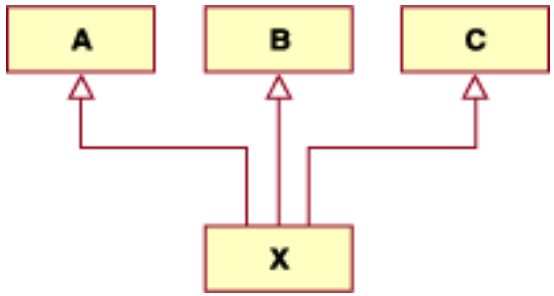
Redefining (3)

```
#include <iostream>
class A{
 int i;
 public:
 A(): i(1){};
 int f(){ return i;}
};
class B: public A{
 int i;
 public:
 B():i(2){};
 void f(int s){i = s;} //REDEFINING
 int g(){
      // return f(); ERROR
      return A::f(); //OK
 }
};
```

Multiple inhertitrance

- You can derive a class from any number of base classes. Deriving a class from more than one direct base class is called multiple inheritance.

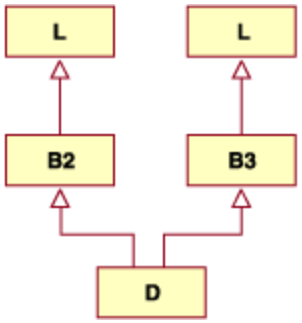
```
class A { /* ... */ };
class B { /* ... */ };
class C { /* ... */ };
class X : public A, private B, public C { /* ... */ };
```



Some problems

- a derived class can inherit an indirect base class more than once

```
class L { /* ... */ };    // indirect base class
class B2 : public L { /* ... */ };
class B3 : public L { /* ... */ };
class D : public B2, public B3 { /* ... */ }; // valid
```



Resolving the name

class D inherits the indirect base class L once through class B2 and once through class B3. ambiguities because two subobjects of class L exist, and both are accessible through class D.

- You can avoid this ambiguity by referring to class L using a qualified class name. For example:

B2::L

or

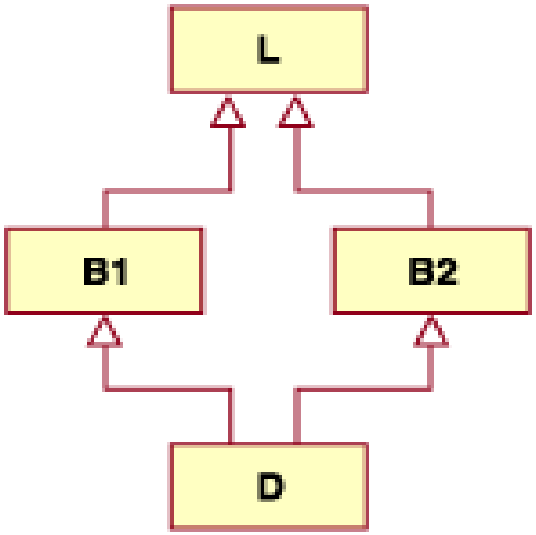
B3::L.

You can also avoid this ambiguity by using the base specifier **virtual** to declare a base class,

diamond

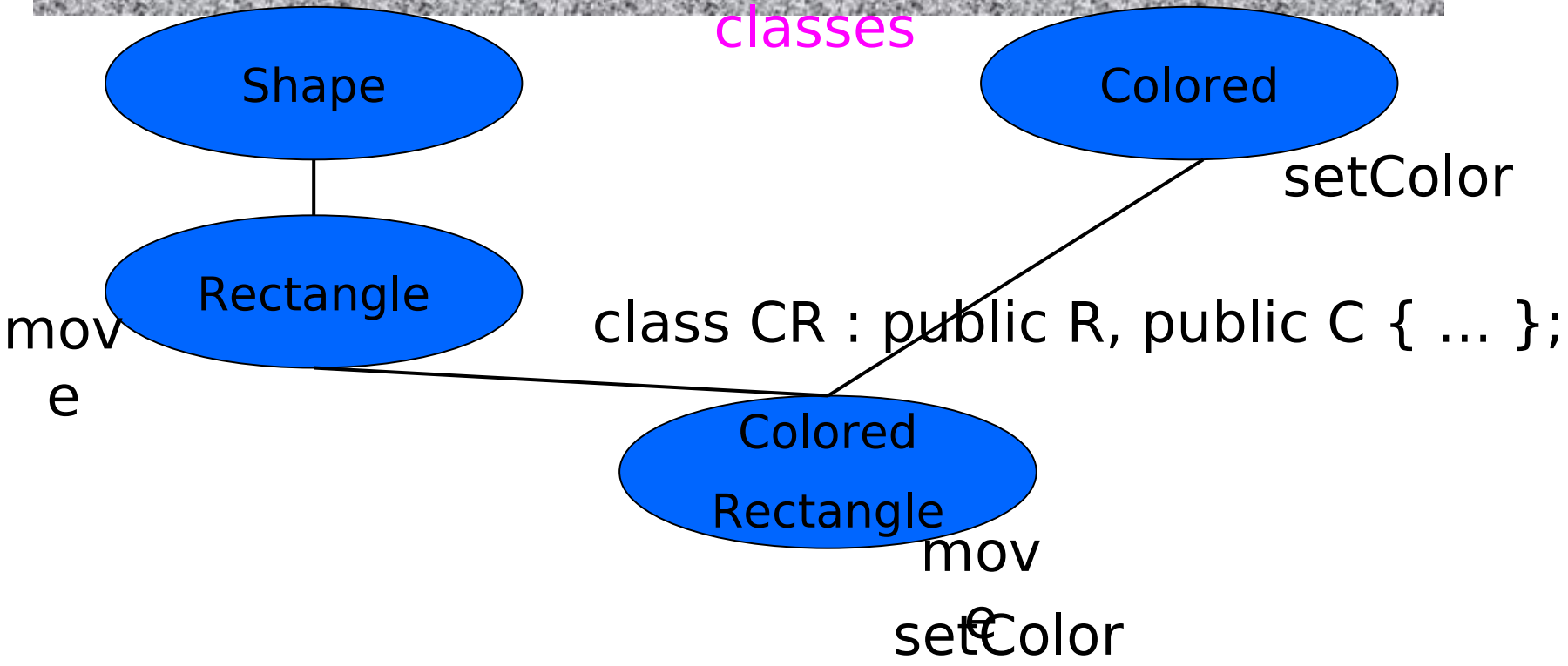
Suppose you have two derived classes B1 and B2 that have a common base class L, and you also have another class D that inherits from B1 and B2. You can declare the base class L as virtual to ensure that B1 and B2 share the same subobject of A.

```
class L { /* ... */ }; // indirect base class
class B1 : virtual public L { /* ... */ };
class B2 : virtual public L { /* ... */ };
class D : public B1, public B2 { /* ... */ }; //
valid
```



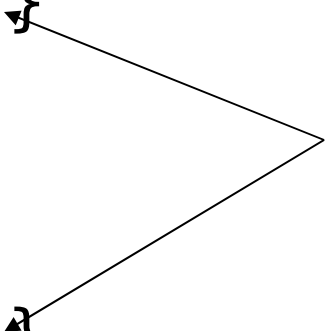
Multiple Inheritance

Inherit independent functionality from independent classes



Problem: Name Clashes

```
class A {
    public:
        void virtual f() { ... }
};
class B {
    public:
        void virtual f() { ... }
};
class C : public A, public B { ... };
...
    C* p;
    p->f();     // error
```



Possible solutions to name clash

- Three general approaches
  - Implicit resolution
    - Language resolves name conflicts with arbitrary rule
  - Explicit resolution
    - Programmer must explicitly resolve name conflicts
  - Disallow name clashes
    - Programs are not allowed to contain name clashes
- No solution is always best
- C++ uses explicit resolution by using fully qualified names

Repair to previous example

- Rewrite class C to call A::f explicitly

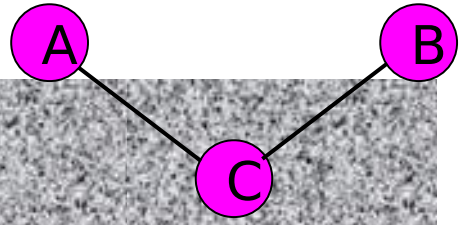
```
class C : public A, public B {
    public:
        void virtual f( ) {
            A::f( );    // Call A::f(), not B::f();
        }
```

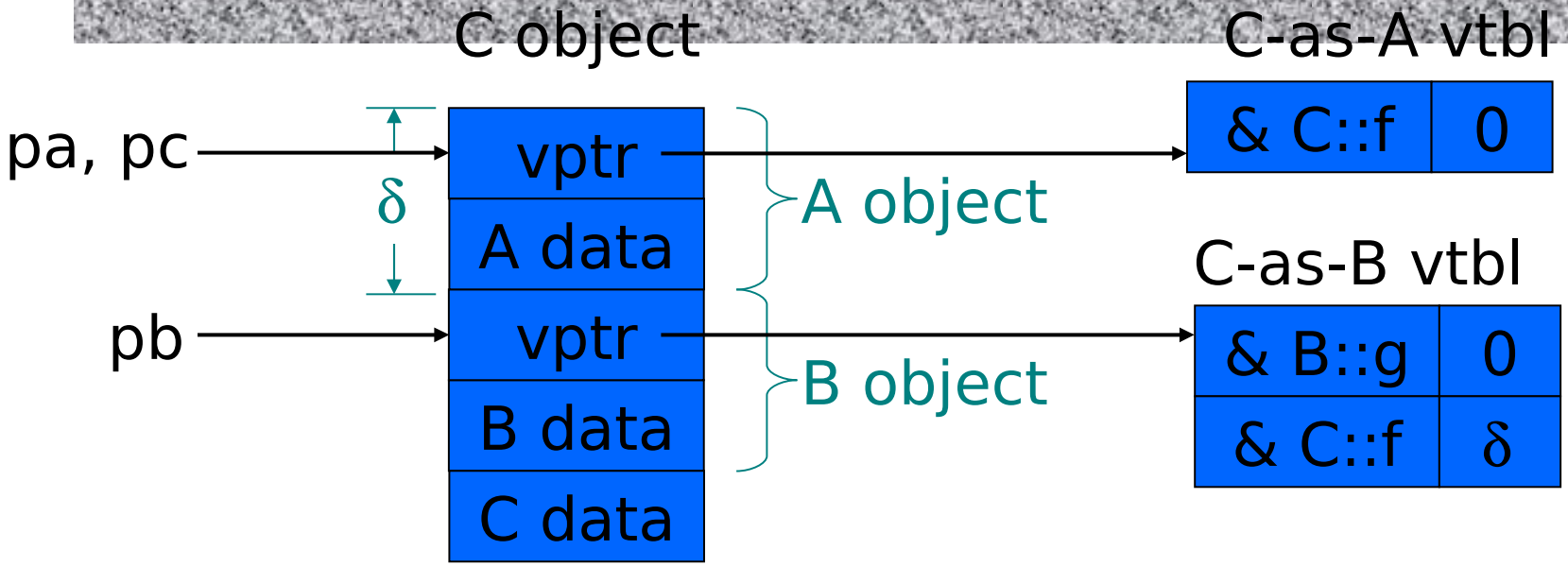
- Reasonable solution
  - This eliminates ambiguity
  - Preserves dependence on A
    - Changes to A::f will change C::f

vtable for Multiple Inheritance

```
class A {
    public:
        int x;
        virtual void f();
};
class B {
    public:
        int y;
        virtual void g();
        virtual void f();
};
```

Object and classes



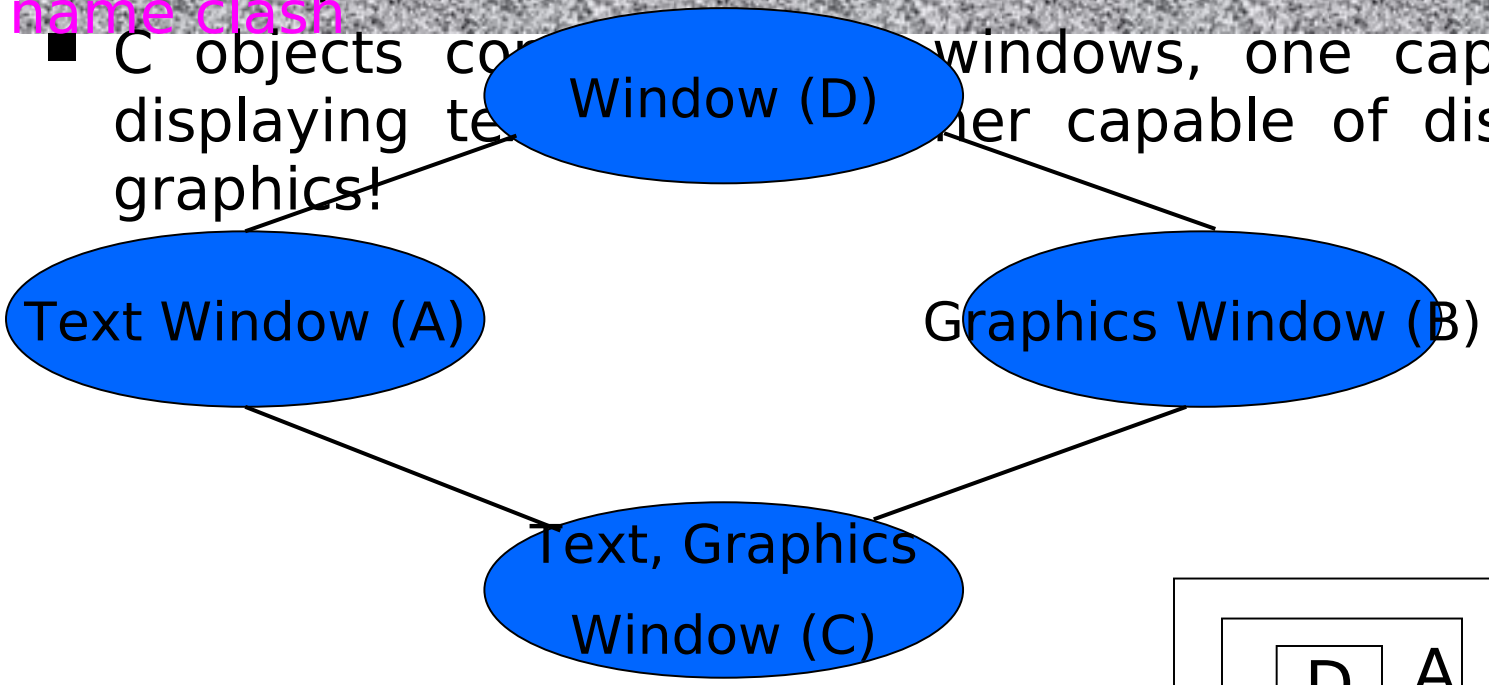


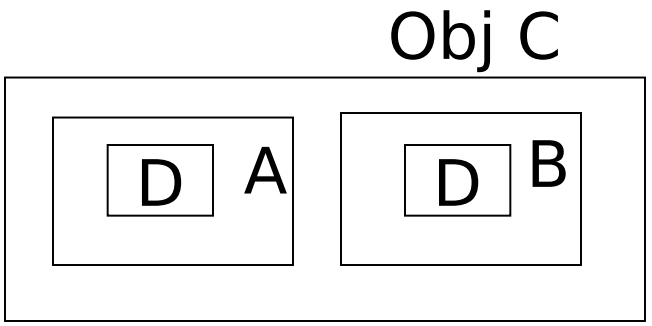
- Offset δ in vtbl is used in call to pb->f, since C::f may refer to A data that is above the pointer pb
- Call to pc->g can proceed through C-as-B vtbl

Multiple Inheritance “Diamond”

The *diamond inheritance Problem*: an interesting kind of name clash

- The implementation is inherited twice
- C objects co… windows, one capable of displaying te… …er capable of displaying graphics!



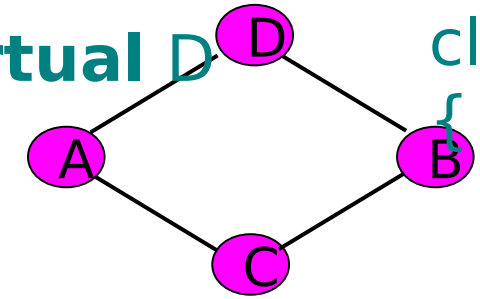


A solution: virtual base classes

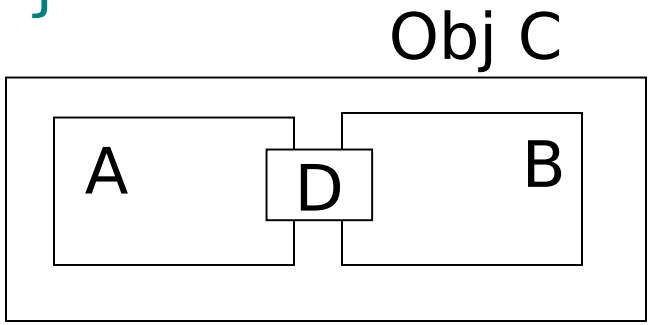
- C++ has a mechanism for eliminating multiple copies of duplicated base-class members,

- called ***virtual base classes*** and consists in declaring D as virtual base class of A and B

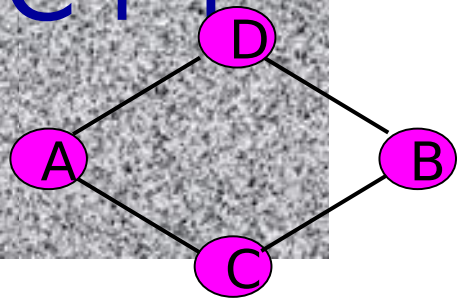
class A : public **virtual** D
{ ... }



class B : public **virtual** D
{ ... }

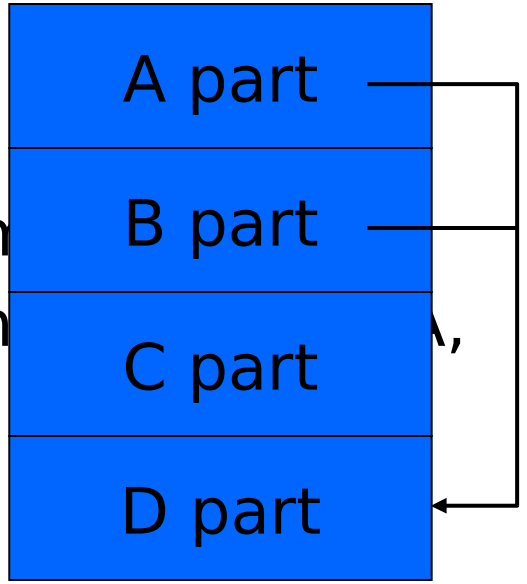


Diamond inheritance in C++



- Standard base classes
  - D members appear twice in C
- Virtual base classes

  class A : public virtual D { ... }
  - Avoid duplication of base class n
  - Require additional pointers so th
    B parts of object can be shared



- C++ multiple inheritance is complicated in part because of desire to maintain efficient lookup
- Virtual base classes give rise to other type conversion problems