Class Hierarchy

Jim Fawcett

August 2019

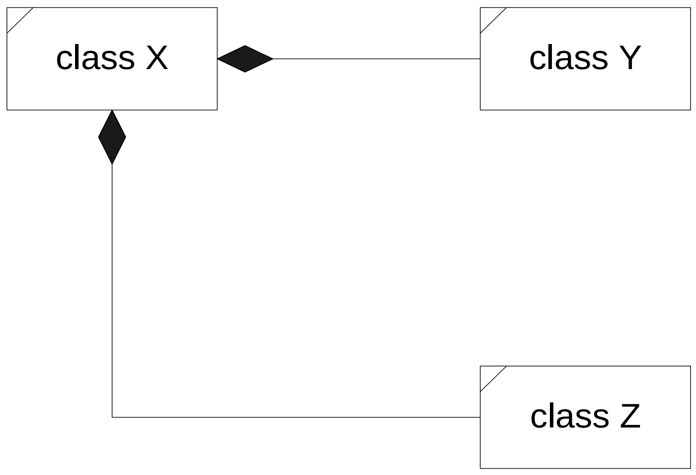
The Object Model

- **Abstraction**
  - Application analysis: class or object model extracts the essential features of a real class or object.
  - Software design: public interface supports simple logical model. Implementation complexity hidden from client view.
- **Modularity**
  - Application analysis: objects provide a more expressive and fine-grained structuring capability than decomposition by processing activity alone.
  - Software design: objects are information clusters which can be declared as often and wherever needed.
- **Encapsulation**
  - classes build “firewalls” around objects, forcing all access through public interfaces, preventing access to private implementation.
  - Objects intercept errors before they propagate outward throughout system

- **Hierarchy**
  - Form **aggregations** by class compositions, e.g., one class uses objects of another as data elements. Supports “part-of” semantic relationship between contained and containing objects.
  - Define subclasses of objects through **inheritance** from base class. Supports an “is-a” semantic relationship between derived classes and base class.

Composition

Compositions are special associations which model a "part-of" or "contained" semantic relationship.



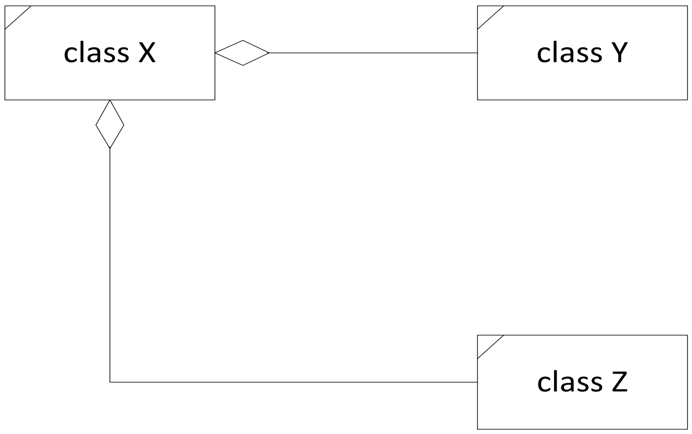
```
Class X {
   // public declarations here
     private:
       Y y;
       Z z;
};
```

In this diagram class X contains objects of classes Y and Z. Classes Y and Z are part-of class X.

Composition is transitive. That is, if class A contains B and class B contains C, then A also contains C.

Aggregation

Aggregations are special associations which model a weak "part-of" or "contained" semantic relationship.

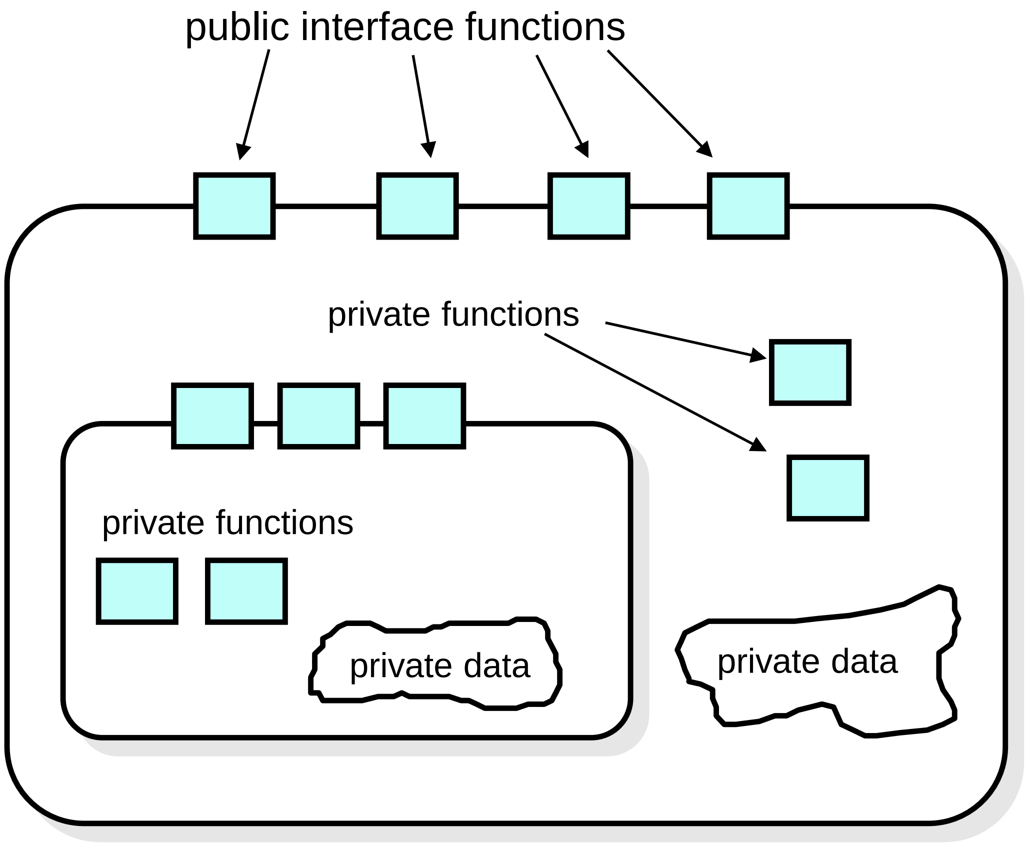


```
Class X {
   // public declarations here
     private:
       Y* pY;  // created in member function
       Z* pZ;  // with new only if needed.
};
```

In this diagram class X holds references to objects of classes Y and Z. Those instances may be part-of class X.

Hierarchy via Composition

- An object of one class may be used as a data element of another.
- This is called composition. Member objects are used to implement a **“part of”** relationship.
- The containing class has no special access to contained object’s private data unless it is made a friend. But an object of the containing class can pass initialization data to the contained object during construction.



Composition – “part of”

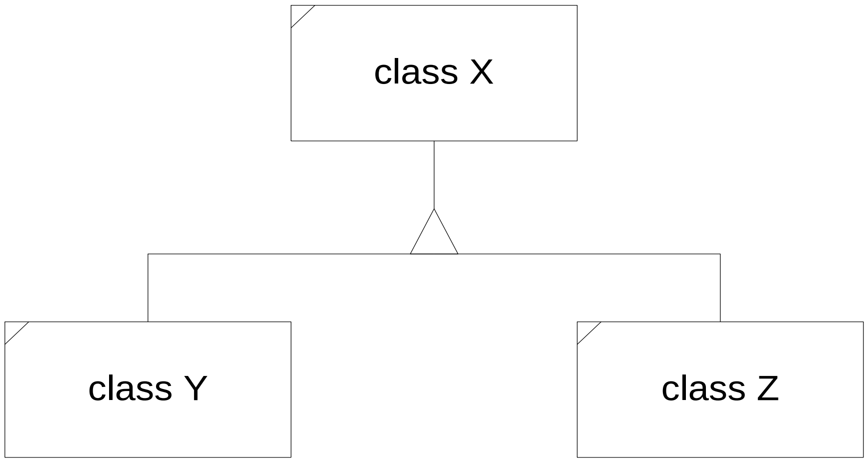
- Incorporating an object inside another by making the contained object a private or protected data member of the containing class creates a **“part-of”** relationship, called a **composition.**
- Containing class has no special privileges regarding access to contained objects internals. It has access only to public interfaces of contained objects, just like any other client.
- Since the contained object lies in the private or protected parts of the containing object, clients can not call its functions and do not see its functionality, except as manifested by the containing object’s behaviors - that is - the contained object helps to implement the outer object’s functionality.
- The “part-of” relationship can be made explicit by providing a member function which returns a reference to the contained part. In this way, clients can get direct access to the public interface of the part.
- “Part-of” relationships can also be implemented by private derivations. The semantics of a private deri-vation are the same as aggregation except that the derived class now has access to the base class’s protected (but not private) members.

Demonstration Programs

- A series of simple demonstration programs are included in Appendix I for composition, and Appendix II for inheritance.
- The demos don’t do much of anything useful, but they are all designed to illustrate points about the way C++ does its job.
- Each demo has a prologue which gives a very brief description of what the demo is about. Comments in the code are intended to draw your attention to specific details.
- Output for each demo is included at the end of each program, along with comments about how you might interpret the code and its output.
- One of the goals of these demos is to illustrate operations that occur under different design strategies. Some of the things that happen may surprise you.
- It is very important that you understand all the things that happen in a C++ implementation. A lot of things happen silently unless you specifically instrument your code to see them. That is exactly what these demo programs do.

Inheritance

Inheritance models an “is-a” semantic relationship. Here classes Y and Z inherit from class X.

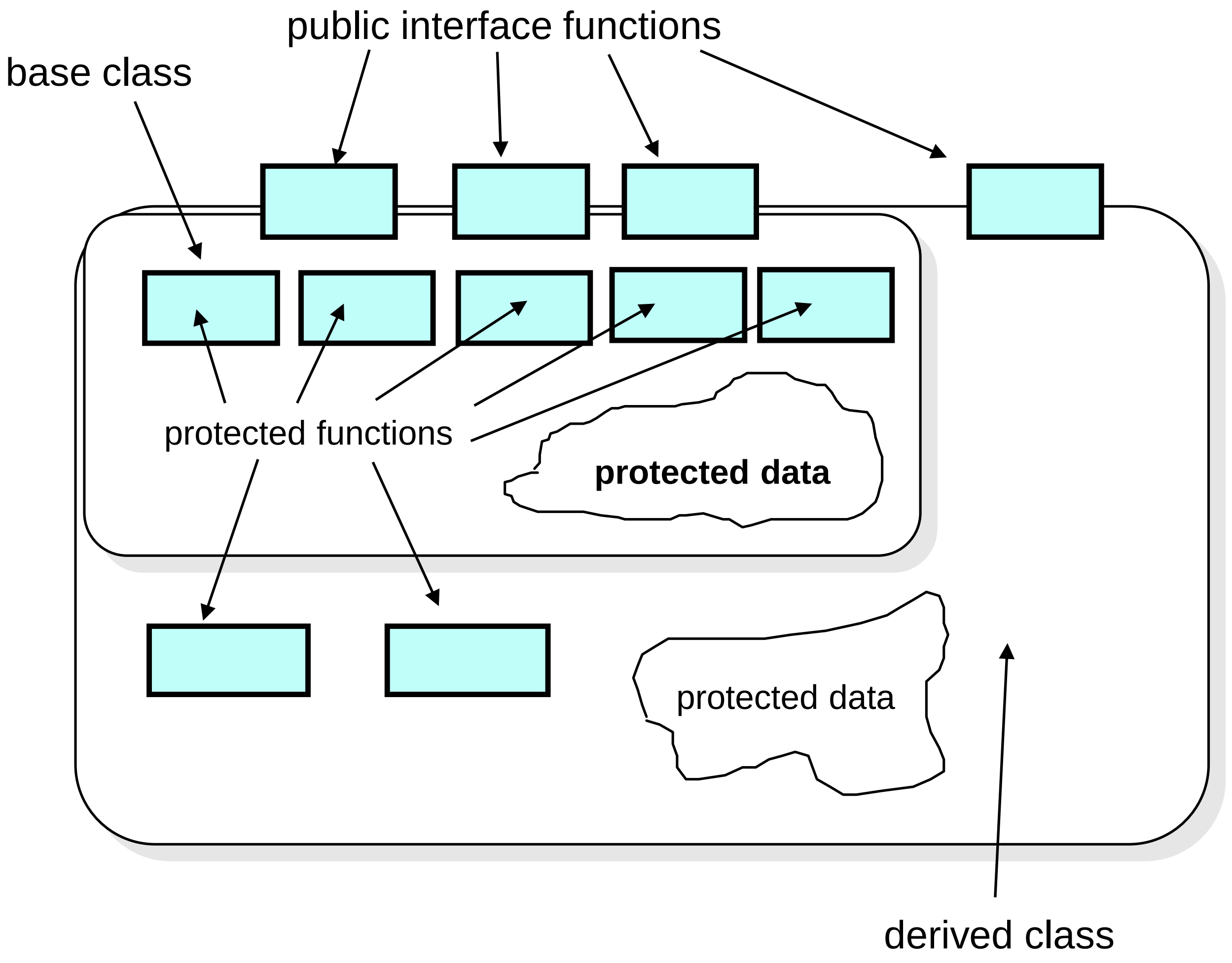


```
class Y : public X { … }
class Z : public X { … }
```

That means that class Y “is-a” class X and the same must be true for class Z. The “is-a” relationship is always a specialization. That is, both classes Y and Z must have all attributes and behaviors of class X, but may also extend the attributes and extend and modify the behaviors of class X.

Hierarchy via Inheritance

- Inheritance enables the derivation of a new class with almost all the existing methods and data members of its base class.
- Derived class functions have access to protected data and functions of the base class.
- The derived class **"is a"** base class object with additional capabilities, creating a new specialized object.



Derived Class Access Privileges

	public members	**protected members**	**private members**
public derivation models is-a relationship client sees all base and derived class behaviors	public members of base class become public members of derived class (stay the same)	protected mem-bers of base class become protected members of derived class (stay the same)	private members of base class are not accessible to derived class
protected derivation models uses relationship client sees only derived behaviors	public members of base class become protected members of derived class	protected mem-bers of base class become protected members of derived class (stay the same)	private members of base class are not accessible to derived class
private derivation models uses relationship client sees only derived behaviors	public members of base class become private members of derived class	protected members of base class become private members of derived lass	private members of base class are not accessible to derived class

Inheritance Example

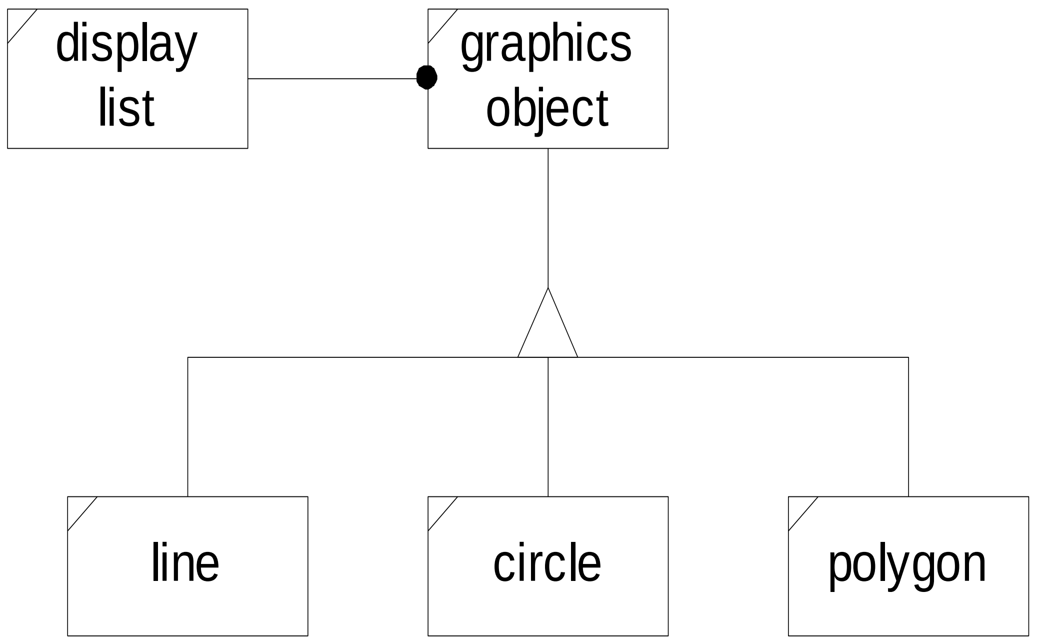
- The inheritance diagram on the next page represents an architecture for a graphics editor. The display list refers to graphics objects, which because of the “is-a” relation-ship, can be any of the derived objects. Presumably a client of the display list creates graphic objects based on user inputs and attaches them to the list. The display list and its clients do not need to know about the types of each of the objects. They simply need to know how to send messages defined by the graphics object base class.

- The base class graphicsObject provides a protocol for clients like the display list to use, e.g., draw(), erase(), move(), ... Clients do not need to know any of the details that distinguish one of the derived class objects from another.

- The absence of a diamond shape on the list class indi-cates that the list does not manage the creation or des-truction of the graphics objects it refers to.

- This example illustrates why the inheritance relationship is considered to represent an “or” relationship between objects on the same level. Display list members are either lines or circles or polygons or ...

Graphics Editor Classes



Public Inheritance – “is-a”

- **Syntax:** `class derived : public base {...};`
- **Public** derivation makes all of the base class functionality available to derived class objects. This has two very important consequences:
    - clients interpret the derived object as a base class object with either specialized, or new capabilities, or both.
    - a derived class object, since it is a base class object, can be used anywhere a base class object can be used. For example, a function typed to accept a base class pointer, or reference, will accept a derived class pointer, or reference in its place.
- New capabilities occur when the derived class adds new member functions or new state members which give the derived object richer state and functional behaviors.
- Specialized capabilities occur when the derived class modifies a base class virtual function.
    - The base class object and derived class object respond to the same message, but in somewhat different ways, determined by the implementations of the virtual function in each class.
    - Because the modified function is qualified in the base class by the keyword virtual, which function is called is determined by the type of object invoking it.

Private Inheritance – “part of”

- **Syntax:** `class derived : private base { ... };`
- **Private** derivation hides all of the base class interface from clients. By default none of the base class member functions are accessible to derived class clients.
- The base class object is “part of” the derived class. All of its attributes are attributes of the derived class.
  Public and protected member functions are available to derived class functions to support their implementation.
- Clients of the derived class see base class functionality only as manifested through derived class interface operations.
- From the client’s point of view private derivation creates an aggregation.
- The only difference is that, using private inheritance, the derived class has access to protected members of the base class.
- In private inheritance any class derived again from the derived class has no more access privileges to the base than any other client of the derived class.

Protected Inheritance – "part of"

- **Syntax:** `class derived : protected base {...};`

- Protected inheritance is just like private inheritance from client's perspective. Clients have no access to base class functionality except through the derived class interface.

- Protected inheritance is just like public inheritance from the derived class's point of view. Derived class member functions have access to all the base class public and protected members. The derived class members can use a derived class object anywhere a base class object is expected.

- Protected inheritance, unlike private inheritance, passes on to all subsequent protected derived classes access privileges to the base public and protected members.

Initialization Sequences

- C++ provides constructors specifically to initialize a newly created object so that its state will assume some correct values.

- In the body of the constructor we can allocate needed resources and assign values to data members of the class.

- However, it is preferable to use an initialization sequence:
  - Initialization sequences are more efficient than void construction and assignment.
  - Use of an initialization sequence is the ***only*** way to chose which constructor will be called for base classes and set member reference values.

    We will see in the demonstration code that this is critically important.

Initialization Sequence Syntax

- An initialization sequence is part of the syntax of a constructor definition:

```
class D : public B
{
  // public interface
  private:
    ostream &_out;
    int _size;
    double *darray;
};

// This syntax assumes that D’s base B can
// be initialized with a std::string, e.g.,
// it has a constructor that takes a string
// as its only argument.

D::D(ostream &out, int size, std::string &s)
  : B(s),  // initialize base class
    _out(out), _size(size),
    darray(new double[size]) {}
```

Polymorphism

- Consider the display list example from the next page. Objects on the list may be any of the types derived from graphicsObject. The display list is said to contain a heterogeneous collection of objects since any one of the graphicsObject types can occur on the list in any order.

- The list manager needs to be able to apply one of several specific operations, like draw() or hide(), to every member of the list. However, draw() and hide() processing will be different for each object.

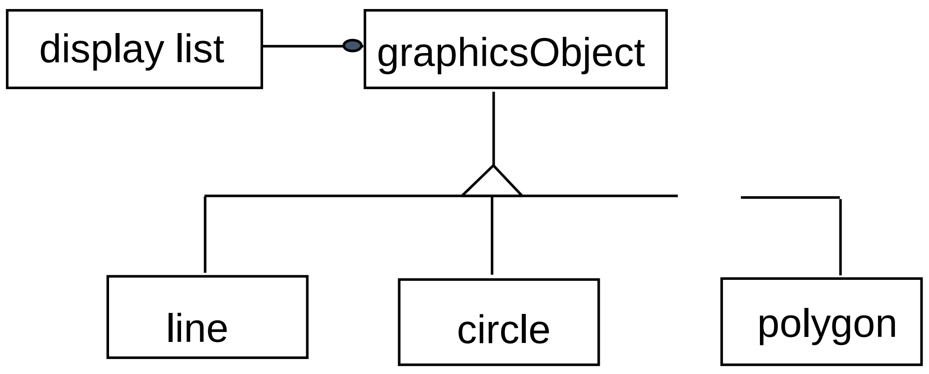
- Languages which support object oriented design provide a mechanism called <u>polymorphism</u> to handle this situation. Each object determines for itself how to process draw() or hide() messages.

- This powerful mechanism is implemented in C++ using virtual functions. Each derived class redefines the base class virtual draw() and hide() member functions in ways appropriate for its class, <u>using exactly the same signature as in the base class</u>.

- We say that the graphicsObject base class provides a protocol for its derived classes by specifying names and signatures of the polymorphic (virtual function) operations.

Polymorphism (cont)

- When a virtual function is redefined in a derived class there are multiple definitions for the same signature, one for each derived class redefinition and often one for the base class as well. Which is called?



- Suppose that a base class member function, say

  ```
  virtual void graphicsObj::draw() {...}
  ```

  is redefined by each of the derived graphics objects. If myLine is an instance of the line class, an invocation

  ```
  myLine.draw()
  ```

  will invoke the version defined by the line class.

- If, however, a display list object has a list of pointers to base class graphicsObjects, the list can point to any derived object, line, circle, ... and an invocation:

  ```
  listPtr[i] → draw();
  ```

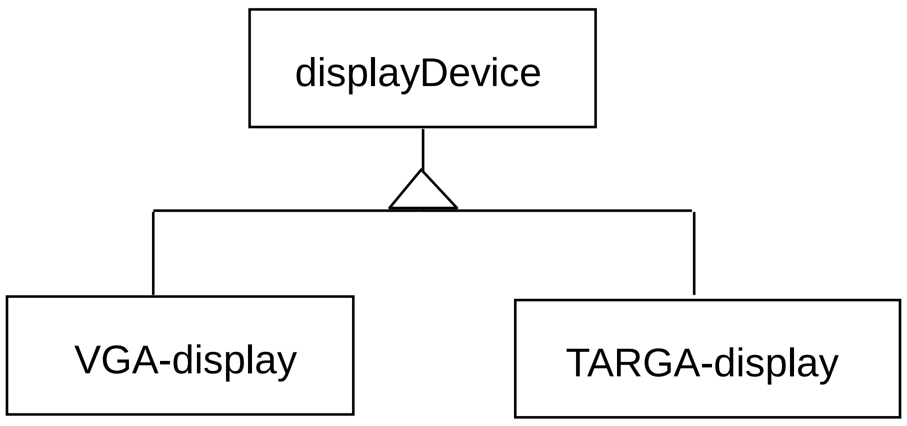
  will call the draw function of the object pointed to, e.g. line, circle, ... , polygon.

Still More Polymorphism

- An invocation of a virtual function through an object will call the function definition provided by the class of the object.
- An invocation of a virtual function through a base class pointer or reference to an object will call the function definition provided by the class of the object referred to.
- This process is called polymorphic dispatching. We say that the display list object dispatches the virtual function draw()
- Polymorphism places the responsibility for choosing the implementation to call with the object, not with the caller.
- Allowing different objects (which must be type compatible) to respond to a common message with behaviors suited to the object is a powerful design mechanism. It allows the caller to be ignorant of all the details associated with the differences between objects an simply focus on their base protocol.

Double Dispatching

- It is possible to design in more than one level of dispatching. Suppose that we need to support more than one environment with our graphics editor. We might define the class hierarchy:



- Suppose further that the draw function was designed to accept a device object which provides all the device specific details:

```
virtual void
graphicsObject::draw(displayDevice& dd);
```

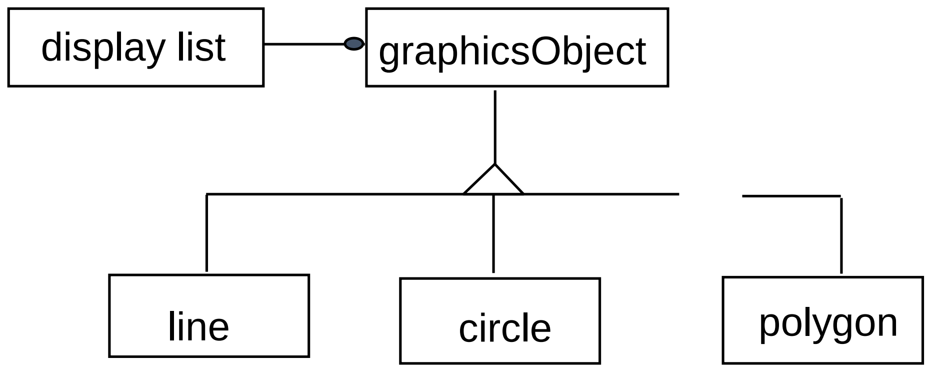
- The invocation:

```
listPtr[i] → draw(dd)
```

will dispatch the draw function for the object pointed to and will dispatch the device calls based on device reference dd.

Abstract Base Class

- A base class like graphicsObject should probably never be instantiated.



- This can be prevented by making graphicsObject an abstract base class. We do that by defining at least one pure virtual function in the class, e.g.:

```
class graphicsObject {
   public:
      virtual void draw() = 0;
     - - -
};
```

- The draw() = 0 syntax tells the compiler that draw may not be called by a client of this class. This in turn means that no instance of the class can be created. It isn't widely known that a body may be defined for a pure virtual function, although we usually don't need to do that.

Abstract Base Class (cont)

- No instance of an abstract class can be created. To attempt to do so is a compile time error.
- If a derived class does not redefine all pure virtual functions in its base class it also is an abstract class.
- If all pure virtual functions are properly redefined in the derived class, that is, with exactly the same signatures as in the base class excluding the “= 0” part, then instances of the derived class can be created.
- Abstract base classes are called protocol classes because they provide a protocol or communication standard by which all derived classes must abide.

Finite State Machine Example

This example simulates an elevator which visits only two floors. When on the first floor the elevator is stationary until the up button is pressed. It then travels toward the second floor. The arrival event brings the elevator to the second floor. It remains stationary on the second floor until the down button is pressed. It then travels toward the first floor. An arrival event brings the elevator back to the first floor.

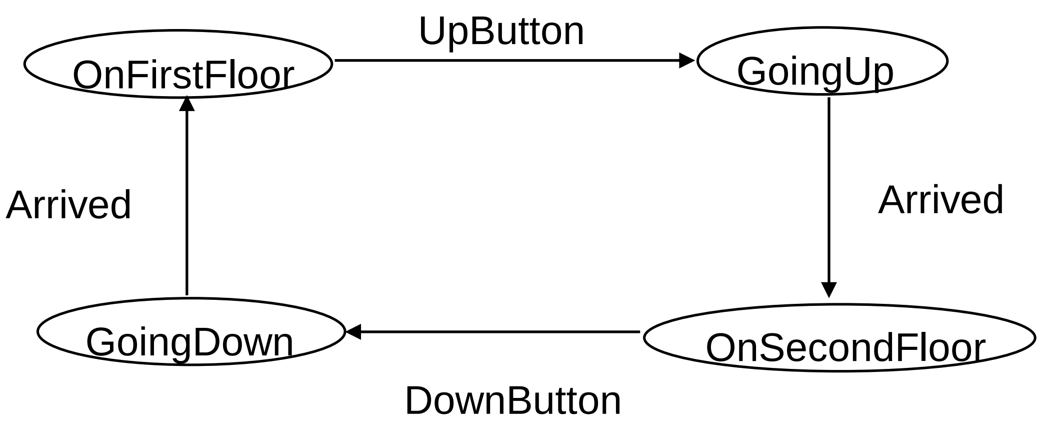
This event sequence is described by the state transition diagram shown on the next page. The elevator simulation consists of implementing the state mechanism with one derived class for each state and a global event processing loop.

The base class defines, as member functions, each of the events the system must respond. The base class members all return pointers to themselves without taking any other action. This essentially defines null events.

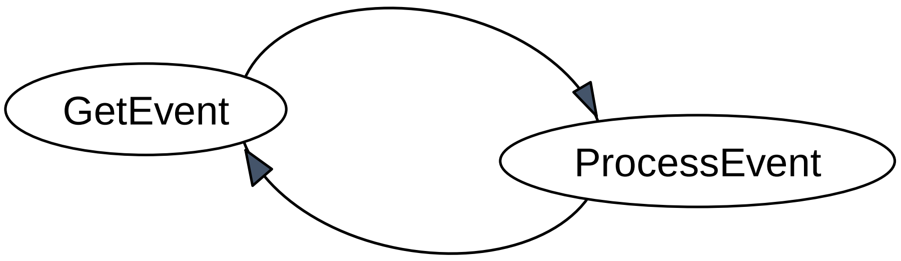
Derived classes override any event which will cause a transition out of that state by returning a pointer to the next state. So, for example, StateOnFirstFloor over-rides upButton to return a pointer to StateGoingUp. Here again, we see polymorphic operation allowing each state object to determine how it responds to the protocol established by the ElevState base class.

FSM Elevator Example

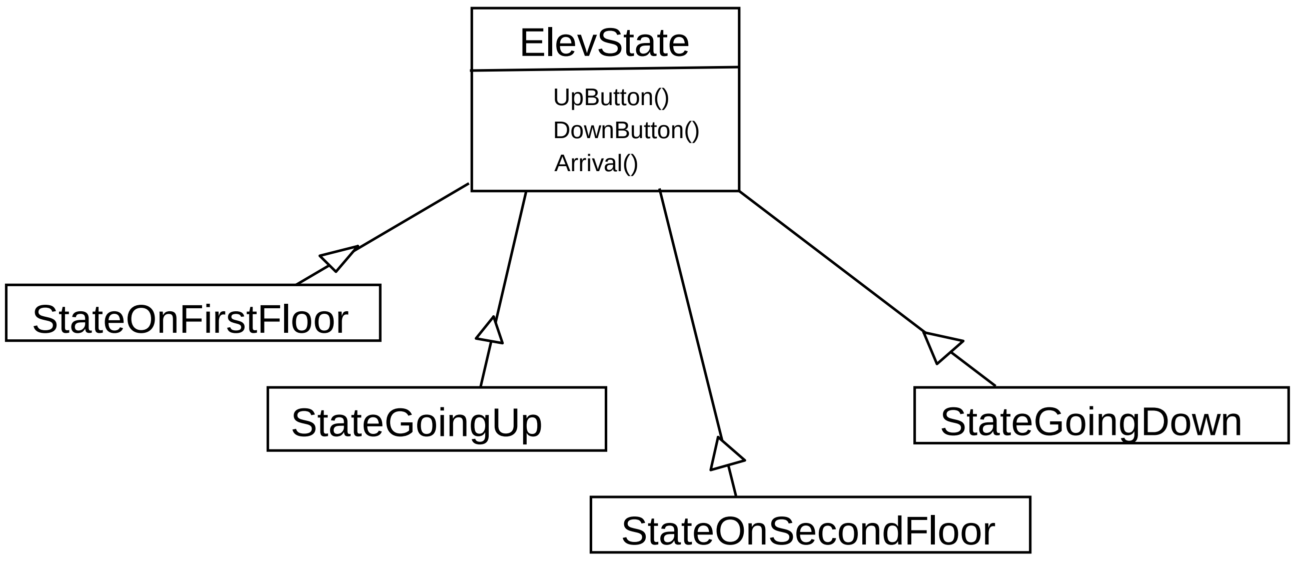
Elevator States:



Event Loop:



Class Hierarchy:



Members not Inherited

- When a class is publicly derived from a base class most of the base class member functions are inherited along with all the base class data members. However, there are a few members which are not inherited:
    - Constructors must be defined for derived class. They automatically call a base class constructor as their first operation. Derived constructors initialize the new data attributes defined by the derived class and pass initializing values to the base class constructors (see example code demInherit3.cpp).
    - Destructor must also be defined for the derived class. It should release resources allocated by the derived class. All base class resources are released by the base class destructor which is automatically called by the derived destructor as its last operation.
    - Assignment operator must be defined for the derived class. An assignment operator should explicitly invoke its base class assignment operator to assign base class data attributes and then assign any derived class data attributes.
- Under **private** inheritance none of the base class operations are accessible (to a client) by default. However, one or more member functions can be made accessible by including in the derived class declaration the expression:

  ```
  base : baseMemberFunction
  ```

  base is the name of the base class and baseMemberFunction is the name of the base class member function to be made accessible.

Default Members

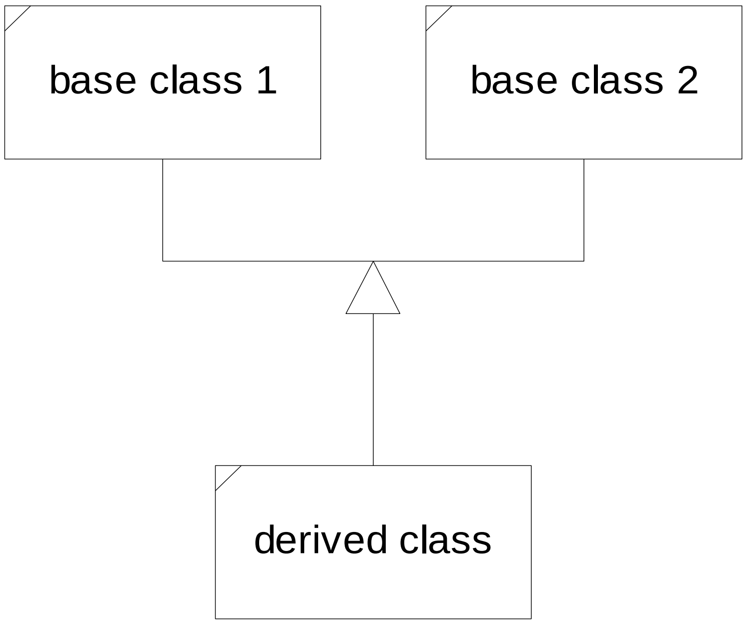
- Since derived classes do not inherit constructors, the destructor, or assignment operator, these members are created by the compiler if needed.
  - if no constructors are declared by a class, the compiler will define void constructor which is used to build arrays of objects of that class. It does member-wise void constructions. If any other constructor is declared for the class a void constructor will not be defined by the compiler. In this case, declaring an array of objects is a compile time error.
  - if no copy constructor is declared by a class the compiler will define one which does member-wise assignment of data attributes from the copied object to the constructed object. This is used for all call and return by value operations.
  - if no destructor is declared the compiler will define one which performs member-wise destruction of each of the class data attributes.
  - if no copy assignment operator is declared by a class the compiler will define one, if needed, which does member-wise assignment of the class's data attributes.
- Note that these default operations may not be what is needed by the class. For example, if a class contains a pointer data element, default copying or assignment will result in copying the pointer, not what is pointed to. This is termed a shallow copy. Usually what is wanted is a deep copy. That is, allocating new memory for the pointed to object, copying the object into new memory, and assigning the address of the new object to the pointer data element.

Default Moves

- If no copy constructor, copy assignment, and destructor are declared, then move constructor and move assignment will be implemented by the compiler.
- The defaults do move operations on the class's bases and data members.
- As on the previous slide this may not be what you want.
- The designer of every class must decide whether to accept the default members, or define those members, or disallow them (with the =delete) syntax.

Multiple Inheritance

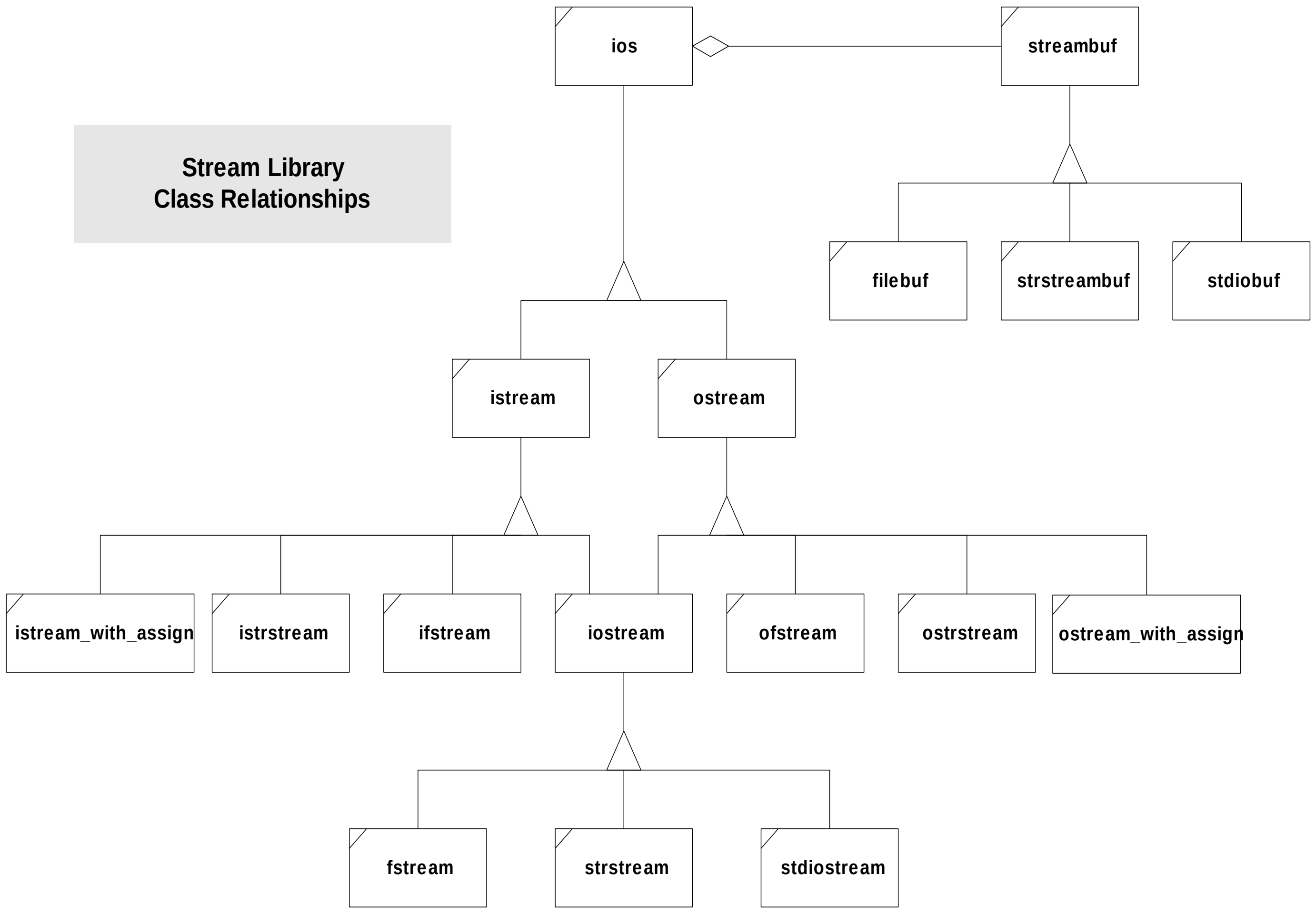
A derived class may have more than one base class. In this case we say that the design structure uses multiple inheritance.



The derived "is-a" base 1 and "is-a" base 2. Multiple inheritance is appropriate when the two base classes are orthogonal, e.g., have no common attributes or behaviors, and the derived class is logically the union of the two base classes.

The next page shows an example of multiple inheritance taken from the iostream module. The class iostream uses multiple inheritance to help provide its behaviors.

iostream Hierarchy



Multiple Inheritance (cont)

- A derived class D may inherit from more than one base class:

```
class D : public A, public B, ... { ... };
```

- A, B, ... is not an ordered list. It is a set. The class D represents the union of the members of classes A, B, and C.

- If a member mf() of A has the same signature as a member of B, then there is an ambiguity which the compiler will not resolve. Sending an mf() message to a derived object will result in compile time failure unless it is explicitly made unambiguous:

```
d.A::mf();
```

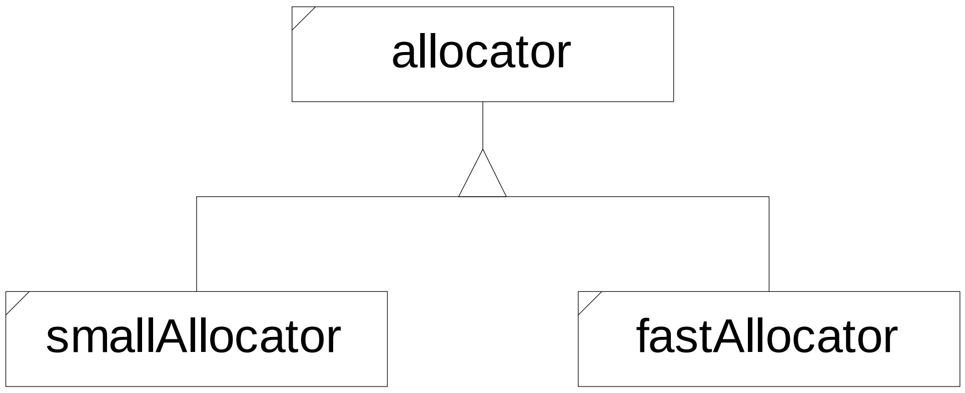
- A constructor for D will automatically call constructors for base objects, in the order cited in D's declaration.

- D's constructor may explicitly initialize data members of each of the base classes by naming parameterized base constructors in an initialization list:

```
D(Ta a, Tb b, Tc C) : A(a), B(b), C(c) {...}
```

Multiple Inheritance Mixins

- Suppose that we wish to design a string class with two specific types of applications in mind.
  - Most applications require minimal character space for each string, If we assign a short string to an existing long string the assignment operator would return left hand's character space and allocate a new, smaller, space.
  - For tokenizing it is critically important that string operations like assignment and copying be as fast as possible. In this case we might decide to reallocate space only if new string length exceeded existing allocation. If smaller simply use front part of existing space. This may eliminate many calls to a slow memory manager.
- We can accomplish these opposing objectives using multiple inheritance in a "mixin" strategy.



- We derive a string class from a base string representation class and one of the allocators. The chosen allocator replaces pointer to string's character space.

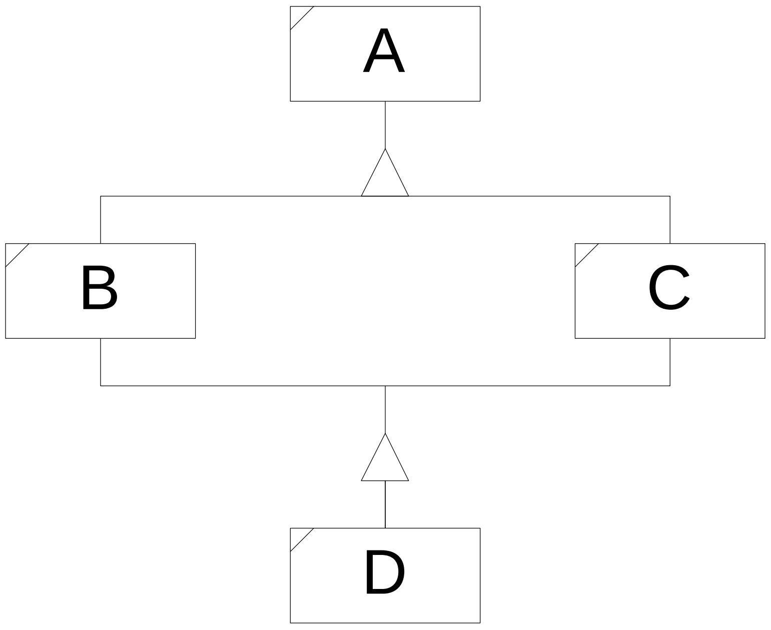
```
class str : public str_rep, private
fastAlloc { ... };
```

Here we mixin the string representation and allocation capabilities using multiple inheritance.

Dreaded Diamonds

- Suppose we have the situation:

```
class B : public A { ... };
class C : public A { ... };
class D : public B, public C { ... };
```

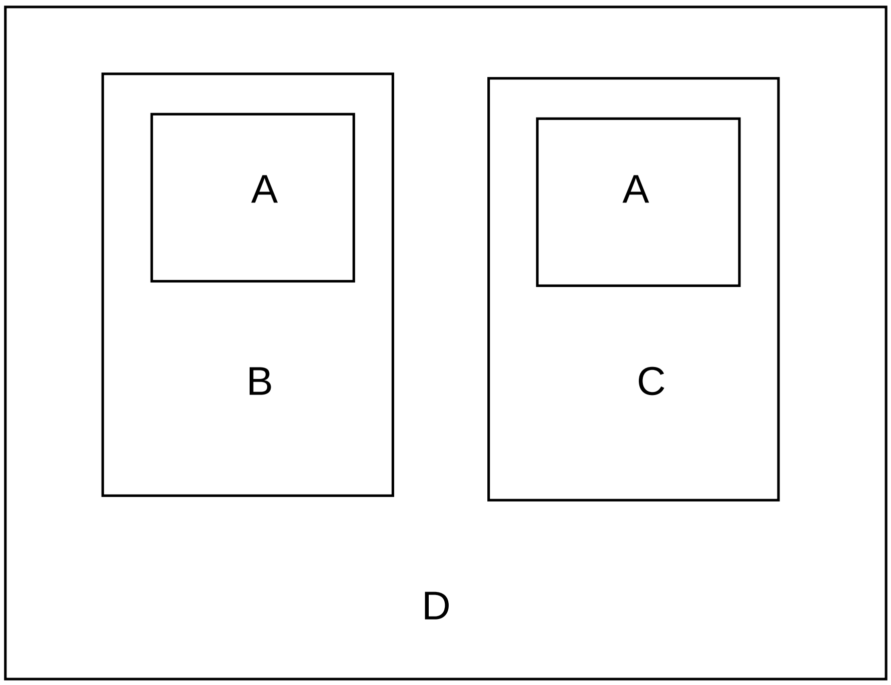


- Since D contains the attributes of all its base classes, all of the attributes of A are repeated twice in D.

Dreaded Diamonds

- Suppose we have the situation:

```
class B : public A { ... };
class C : public A { ... };
class D : public B, public C { ... };
```

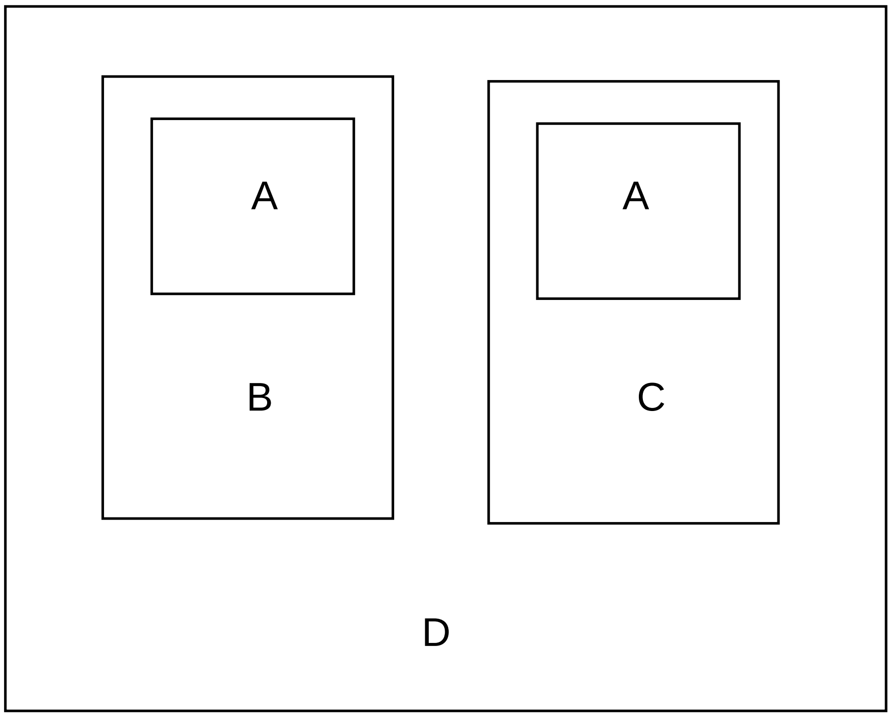


- Since D contains the attributes of all its base classes, all of the attributes of A are repeated twice in D.

Construction Sequence

- Base class constructors are called implicitly by derived class constructors. The B and C constructors are called by D's constructor.

- Who calls A's constructor?

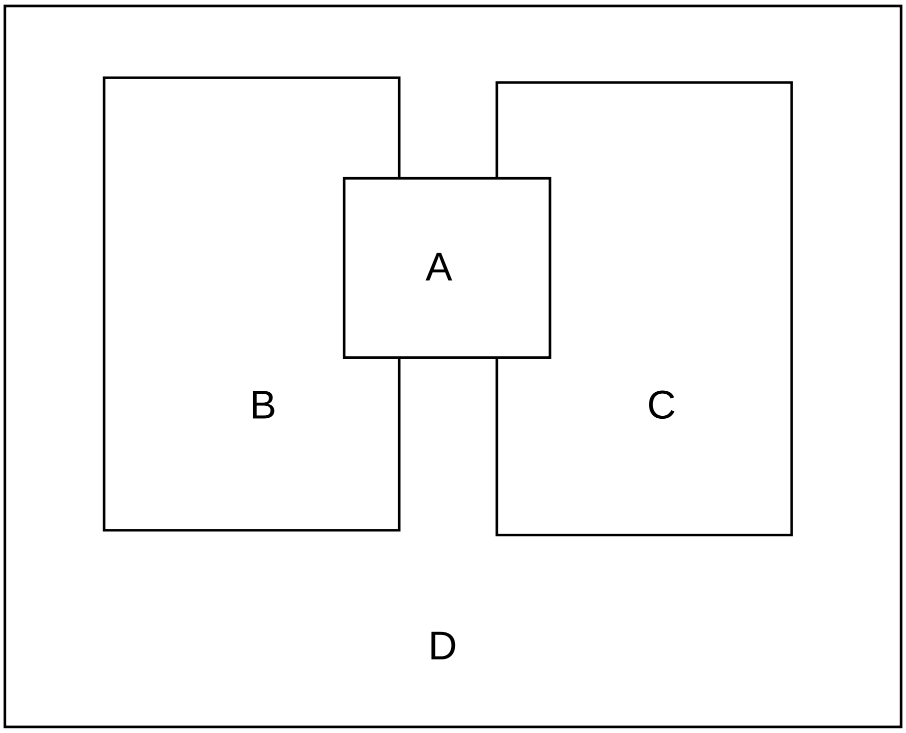


- B will call its A constructor. C will call its A constructor.

Virtual Base Classes

- We can avoid duplication of A’s members by making it a virtual base class:

```
class B : virtual public A { ... };
class C : virtual public A { ... };
class D : public B, public C { ... };
```

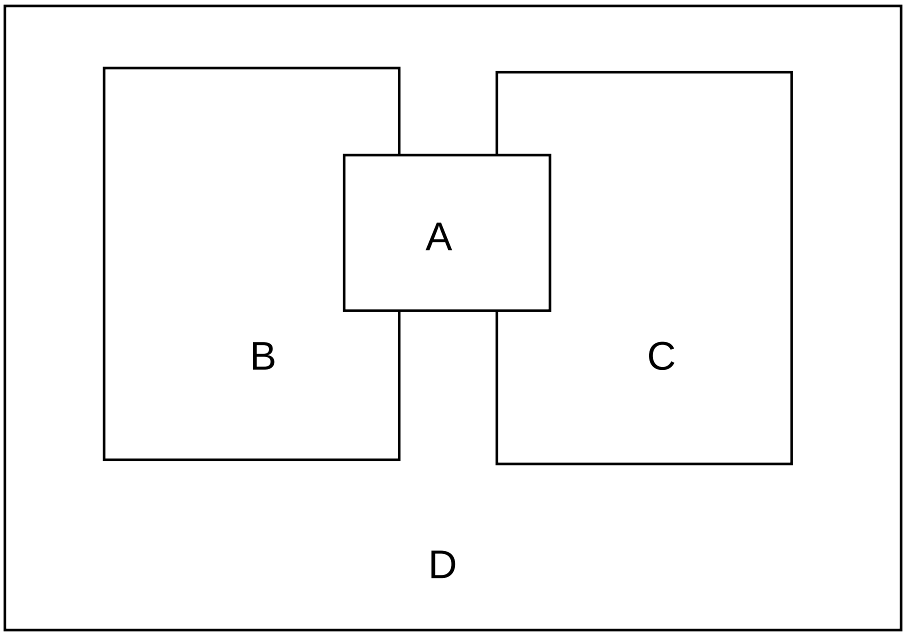


- Now an object of the D class contains only one set of base class A’s attributes.

Construction Sequence

- We can avoid duplication of A's members by making it a virtual base class:

```
class B : virtual public A { ... };
class C : virtual public A { ... };
class D : public B, public C { ... };
```



- Who calls A's constructor? The constructor for B? The constructor for C? C++ resolves the ambiguity by requiring the <u>most derived class to invoke a virtual base class's constructor</u>. So D's constructor will construct B and C and A. Note that that sequence is different than for any derivation chain with non-virtual base.

Initializing Virtual Base Classes

- A constructor of a derived class may explicitly initialize its base(s) with the syntax:

```
B(Ta a) : A(a) { ... }
```

- A virtual base class must be initialized by the most derived constructor, so, for example:

```
class B : virtual public A { ... };
class C : virtual public A { ... };
class D : public B, public C { ... };
```

  A will be initialized by:

```
D(Ta a, Tb b, Tc c)
    : A(a), B(b), C(c) { ... }
```

  If A were not virtual B’s copy would be initialized by B and C’s copy would be initialized by C.

- Note that changing a base class from non-virtual to virtual can break correct code.

End of Presentation