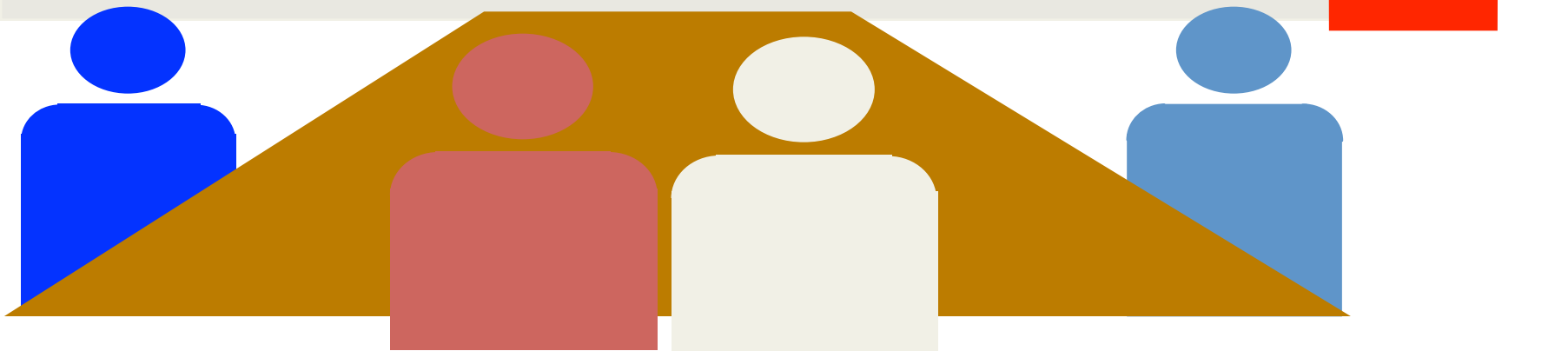


Inheritance in C++

The OOP Design Methodology

1. Decide on an appropriate set of *types*
2. Design in their *relatedness*
3. Use *inheritance* to share code among classes.



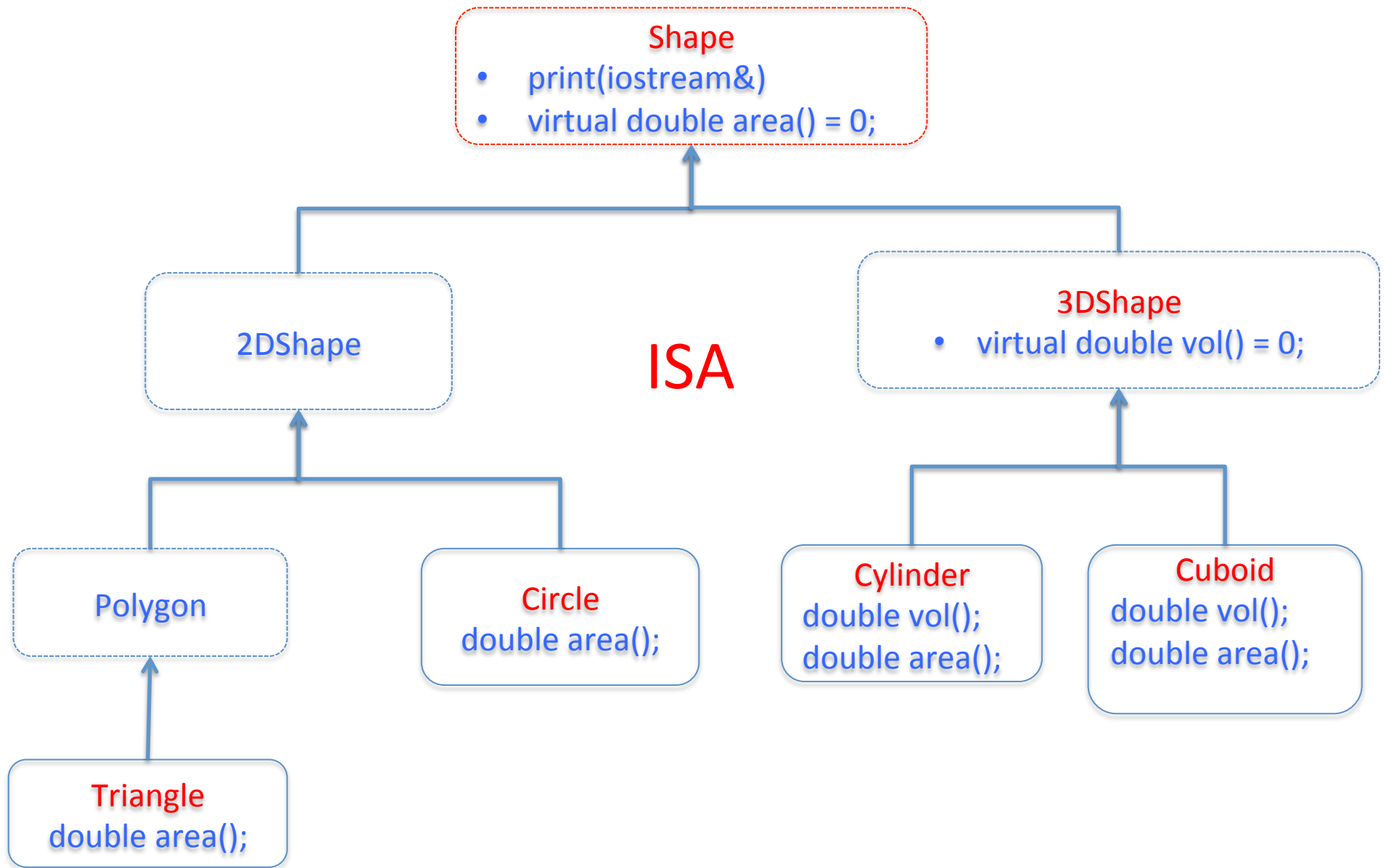
Relationships between Objects

- ISA Relationship – specialization or inheritance
- HASA relationship – *Composition*. Realized as either *aggregation* or *association*.
 - Aggregation => wholly contained or owned by.
 - Association => has a reference to. Many objects may be referencing one object.

The Inheritance Mechanism

- **WHAT?** Means of *deriving* new class from existing classes, called *base classes*
- **WHY?** - *Reuses* existing code eliminating tedious, error prone task of developing new code
- **HOW?** - Derived class developed from base by *adding* or *altering* code
- **RESULT?** *Hierarchy* of related types created that share code & interface

Inheritance as Specialization



What a derived class inherits

- Every data member defined in the parent class
- Every ordinary member function of the parent class
- *However, such members may not always be accessible or exposed through the derived class!*
 - Private members of a base class are not accessible to the derived class under any circumstances.

What a derived class doesn't inherit

- The base class's ***constructors*** and ***destructor***
- The base class's ***assignment*** operator
- The base class's ***friends***

What a derived class can add

- New data members
- New member functions
 - override existing ones
- New constructors and destructor
- New friends

When a derived-class object is *created & destroyed*

1. Space is allocated (on the stack or the heap) for the full object (that is, enough space to store the data members inherited from the base class plus the data members defined in the derived class itself)
2. The base class's constructor is called to initialize the data members inherited from the base class
3. The derived class's constructor is *then* called to initialize the data members added in the derived class
4. The derived-class object is then usable
5. When the object is destroyed (goes out of scope or is deleted) the derived class's destructor is called on the object ***first***
6. ***Then*** the base class's destructor is called on the object
7. Finally the allocated space for the full object is reclaimed

If you define your own constructors you should follow this – the compile will not do it for you.

Polymorphism

- “Taking many forms”

```
Shape* shp = new Circle(centerx, centery, radius);  
printShapeAreaToStdout(shp);
```

```
shp = new Cuboid(originx, originy, width, length,  
height);  
printShapeAreaToStdout(shp);
```

```
void printShapeAreaToStdout(Shape* shp){  
    cout << shp -> area() << endl;  
}
```

Dynamic Dispatch

- Runtime dispatch to the right version of the function depending on the *type of the object* on which the method is called – *NOT on the type of the pointer.*

```
void printShapeAreaToStdout(Shape* shp){  
    cout << shp -> area() << endl;  
}
```

- Which version of area is called? Can we decide this statically?

Virtual Methods

- A virtual method is a method that can be overridden by a subclass.

```
class Vehicle {  
    public:  
        ...  
        virtual int topSpeed() {return 75;}  
        ...  
};
```

```
class TwoWheeler : public Vehicle {  
    public:  
        int topSpeed() { return 60;}  
};
```

Virtual Methods and Overriding

- The subclass ***may redefine*** the virtual method.

```
class Circle : public 2Dshape {  
    public:  
        double area() {  
            //overrides any previous impl  
        }  
};
```

- In this case Circle overrides the definition of the 2Dshape class.
- Don't need to keep using the virtual keyword once used in the base class.

Co variant return types in Overriding

```
Class A {  
    public:  
    ...  
};
```

```
Class B : public A {  
    ...  
};
```

```
class C {  
    public:  
        virtual A foo();  
};
```

```
Class B : public A {  
    public:  
        virtual B foo(){};  
};
```

Overloading Vs Overriding

- Overloading => different methods with the same name and different signatures.
 - Selection is compile time.
- Overriding => same signature as in the base class but a different implementation in the derived class
 - Selection is by dynamic dispatch at run time.

Calling Virtual Functions with Dynamic Dispatch

- We could have an array of shapes on which we wish to get the area in each case.

```
void printAreas(Shape** shapes, int n) {  
    for (int i = 0; i < n; i++){  
        cout << shapes[i] -> area() << endl;  
    }  
}
```


Virtual Destructors

```
class B : public A {  
    Y * y;  
public:  
    B();  
    ~B();  
};  
  
B::B() {  
    y = new Y;  
}  
  
B::~~B() {  
    delete y;  
}
```

Virtual Destructors

```
B * b = new B();
```

```
...
```

```
delete b; // It will call ~B() and then  
~A()
```

```
A * a = new A();
```

```
..
```

```
delete a; // It will call ~A()
```

```
A* a = new B();
```

```
..
```

```
delete a; // It will call ~A() only!!!
```

Virtual Destructors

- To make sure that ~B destructor is called, you need to define the destructor in A as virtual.

A.h

```
class A {  
    X * x;  
public:  
    A();  
    virtual ~A();  
};
```

- Now

```
A* a = new B();  
..  
delete a; // It will call ~B() and ~A() that is  
          what we want.
```

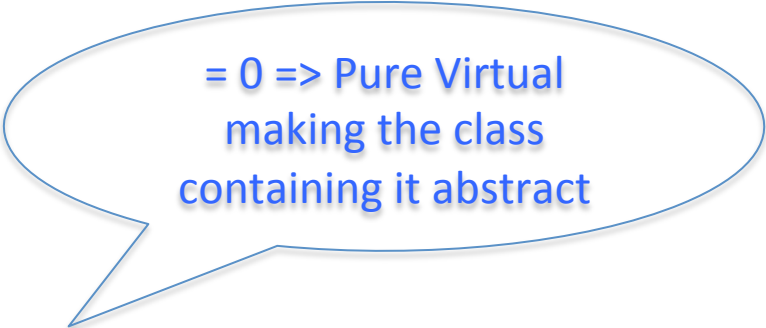
Restrictions on virtual Functions

- Only **non-static** member functions virtual
- The “Virtual” characteristic is inherited
 - Derived class function automatically virtual **virtual** keyword not needed
- ***Constructors cannot be virtual***
- Destructors can be virtual

Pure Virtual Methods – Abstract Classes

- A virtual method is a method that can be overridden by a subclass.

```
class Shape {  
    public:  
    ...  
  
    virtual double area() = 0;  
    virtual int numSides() = 0;  
    ...  
};
```



= 0 => Pure Virtual
making the class
containing it abstract

- Another way of realizing Abstract classes is by making all constructors protected. Which means the class MUST be inherited to be instantiable.

Interface Vs Implementation Inheritance

- Java has two types of inheritance:
 1. Interface inheritance
 - Realized through “implements”
 2. Implementation Inheritance
 - Realized through “extends”
- C++ does this through Public Vs Private inheritance.

Public Inheritance - visibility

- It is used when you want a derived class to inherit all the public interface of the parent class *and keep it all public in the derived class.*

Figure.h

```
class Figure {  
    public:  
        enum FigureType { Line, Rectangle,  
                           Circle, Text };  
  
        Figure(FigureType figureType);  
  
        FigureType getFigureType();  
        void select(bool selected);  
        ...  
};
```

Public Inheritance

```
class Figure {  
    public:  
        void select();  
};  
  
class Line : public Figure {  
    int x0, y0, x1, y1;  
    public:  
        Line(int x0, int y0,  
              int x1, int y1);  
        int getX0();  
        ...  
}
```


Private Inheritance

- In private inheritance, *the public methods of the parent class are private in the subclass.*

```
class A {  
    public:  
        void xx();  
        void yy();  
};  
class B : private A {  
    public:  
        using A::xx(); // Makes xx() public.  
                        // Only xx() is accessible in B.  
                        // yy() is private.  
};
```

This is not realizing the ISA relationship. Why use this?

Private Inheritance OR Composition?

```
template <class T>
class MyList {
public:
    bool    Insert( const T&, size_t index );
    T       Access( size_t index ) const;
    size_t  Size() const;
private:
    T*      buf_;
    size_t  bufsize_;
};
```

```
template <class T>
class MySet1 : private MyList<T> {
public:
    bool    Add( const T& ); // calls Insert()
    ...
};
```

Another approach

```
template <class T>
class MySet2 {
public:
    bool    Add( const T& ); // calls impl_.Insert()
    T       Get( size_t index ) const;
                                // calls impl_.Access()
    size_t  Size() const;      // calls impl_.Size();
    //...
private:
    MyList<T> impl_;
};
```

Which one should you use?

Why MySet should NOT inherit from MyList

- MyList has no protected members, so we don't need to inherit to gain access to them.
- MyList has no virtual functions, so we don't need to inherit to override them.
- MySet has no other potential base classes, so the MyList object doesn't need to be constructed before, or destroyed after, another base subobject.
- MySet IS-NOT-A MyList, not even within MySet's member functions and friends.

Why have private inheritance?

- The base class provides a customizable implementation, using the template method pattern, and you have to override its virtual functions.
 - Some times you need access to protected members of a base class. If you can't change the member class itself, you have to use private inheritance to get access to them.

Finer grained Control: Protected Inheritance

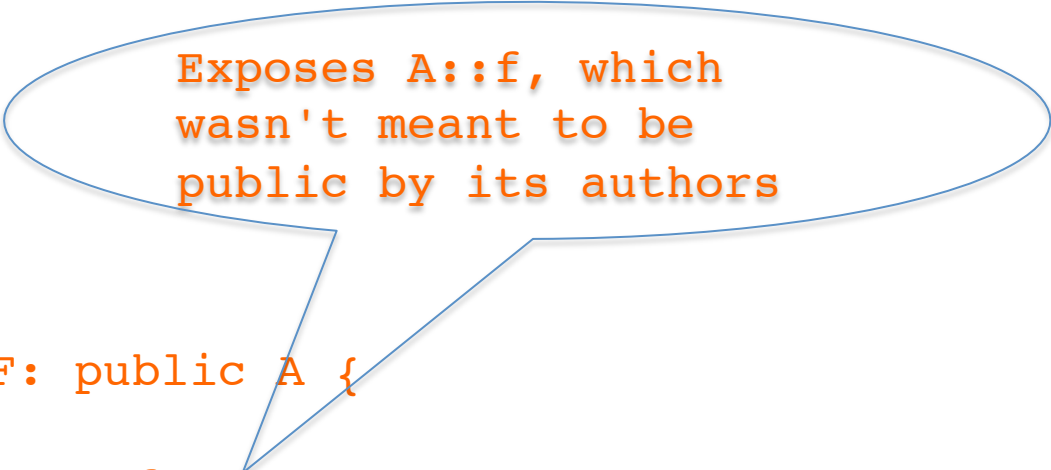
- In protected inheritance, the public methods of the parent class are protected in the subclass.

```
class A {  
    public:  
        void xx();  
        void yy();  
};  
class B : protected A {  
    public:  
        using A::xx(); // Makes xx() made public.  
                        // Only xx() is public in B.  
                        // yy() is protected so it  
                        //can be inherited in a subclass.  
};
```

Replacing protected inheritance with private inheritance

```
class A {  
protected:  
    void f(); // <- I want to use this from B!  
};
```

```
class B : protected A {  
private:  
    void g() { f(); }  
};
```



Exposes A::f, which
wasn't meant to be
public by its authors

```
class AWithF: public A {  
public:  
    void f() { A::f(); }  
};
```

```
class B {  
private:  
    AWithF m_a;  
    void g() { m_a.f(); }  
};
```

Multiple Inheritance

```
class A {  
    public:  
        void xx();  
};  
class B {  
    public:  
        void yy();  
};  
class C: public A, public B {  
};  
// C inherits from both A and B so xx() and yy() are public.
```

- ***Multiple inheritance is discouraged since adds extra complexity that is not needed.***
 - *Java uses single inheritance but a class may implement multiple interfaces.*

The Real Issue with Multiple Inheritance

```
Class student {
```

```
    . . .
```

```
};
```

```
class under_grad: public student {
```

```
    . . .
```

```
};
```

```
class grad: public student {
```

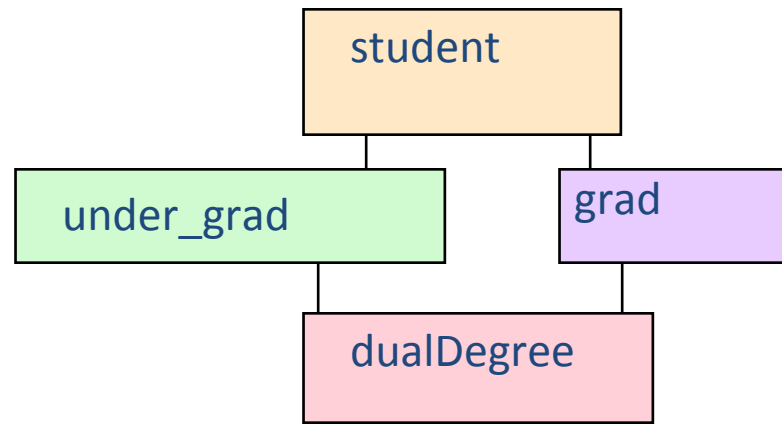
```
    . . .
```

```
};
```

```
class dualDegree : public under_grad, public grad {
```

```
    . . .
```

```
};
```



Without use of virtual, class dualDegree would have objects of class::under_grad::student and class::grad::student

Using virtual inheritance

```
Class student {
```

```
    . . .
```

```
};
```

```
class under_grad: public virtual student {
```

```
    . . .
```

```
};
```

```
class grad: public virtual student {
```

```
    . . .
```

```
};
```

```
class dualDegree : public virtual under_grad, public  
virtual grad {
```

```
    . . .
```

```
};
```