Polymorphism
(a.k.a. Dynamic Binding or Late Binding)

Inheritance and Polymorphism

- Inheritance allows us to define a family of classes that have common data and behaviors.
- Polymorphism is the ability to manipulate objects of these classes in a type-independent way.
- In C++, polymorphism is supported only when we use pointers (or references) to objects.
- The particular method to invoke on an object is not determined until run-time and is based on the specific type of object actually addressed.

Review: Inheritance and Pointers

- A base class pointer can point to an object of a derived class type.
  - The derived class object “is-a” base class object.
  - But we can’t use the pointer to call methods only defined in the derived class.
- A derived class pointer <u>cannot</u> point to an object of a base class type.
  - The base class doesn’t have any of the extensions provided by the derived class.

Static vs. Dynamic Binding

- **Binding**

The determination of which method in the class hierarchy is to be invoked for a particular object.

- **Static (Early) Binding occurs at compile time**

When the compiler can determine which method in the class hierarchy to use for a particular object.

- **Dynamic (Late) Binding occurs at run time**

When the determination of which method in the class hierarchy to use for a particular object occurs during program execution.

Static Binding

```
class Time {
public:   Time(int h = 0, int m = 0, int s = 0);
          void setTime(int h, int m, int s);
          void printTime(  );
private: int hrs, mins, secs;
};
class ExtTime: public Time  {
public:   ExtTime(int h = 0, int m = 0, int s = 0, string z = "EST");
          void setExtTime(int h, int m, int s, string z);
          void printExtTime( );
private: string zone;
};
```

Static Binding

```
Time t1, *tPtr = &t1;
ExtTime et1, *etPtr = &et1;

t1.setTime(12, 30, 00);                    // static binding
et1.setTime(13, 45, 30);                   // static binding
et1.setExtTime(13, 45, 30, "CDT");         // static binding

t1.printTime( );        // static binding – Time’s printTime( )
et1.printTime( );       // static binding – ExtTime’s printTime( )
et1.printExtTime( );    // static binding – ExtTime’s printExtTime( )

tPtr->printTime( );         // static binding - Time’s printTime( )
etPtr->printTime( );        // static binding - ExtTime’s printTime( )
etPtr->printExtTime( );     // static binding - ExtTime’s printExtTime( )
```

Dynamic Binding

- When the compiler cannot determine the binding of an object to any method.
- Dynamic binding is determined at runtime.
- To indicate that a method is to be bound dynamically, the class must use the reserved word **virtual** in the method's prototype.
- When a method is defined as virtual, all overriding methods from that point on down the hierarchy are virtual, even if not explicitly defined to be so.
- For clarity, explicitly use the **virtual** reserved word.

Dynamic Binding

```
class Time {
public:   Time(int h = 0, int m = 0, int s = 0);
          void setTime(int h, int m, int s);
          virtual void printTime(  );
private: int hrs, mins, secs;
};
class ExtTime: public Time  {
public:   ExtTime(int h = 0, int m = 0, int s = 0, string z = "EST");
          void setExtTime(int h, int m, int s, string z);
          virtual void printTime( );
private: string zone;
};
```

Dynamic Binding

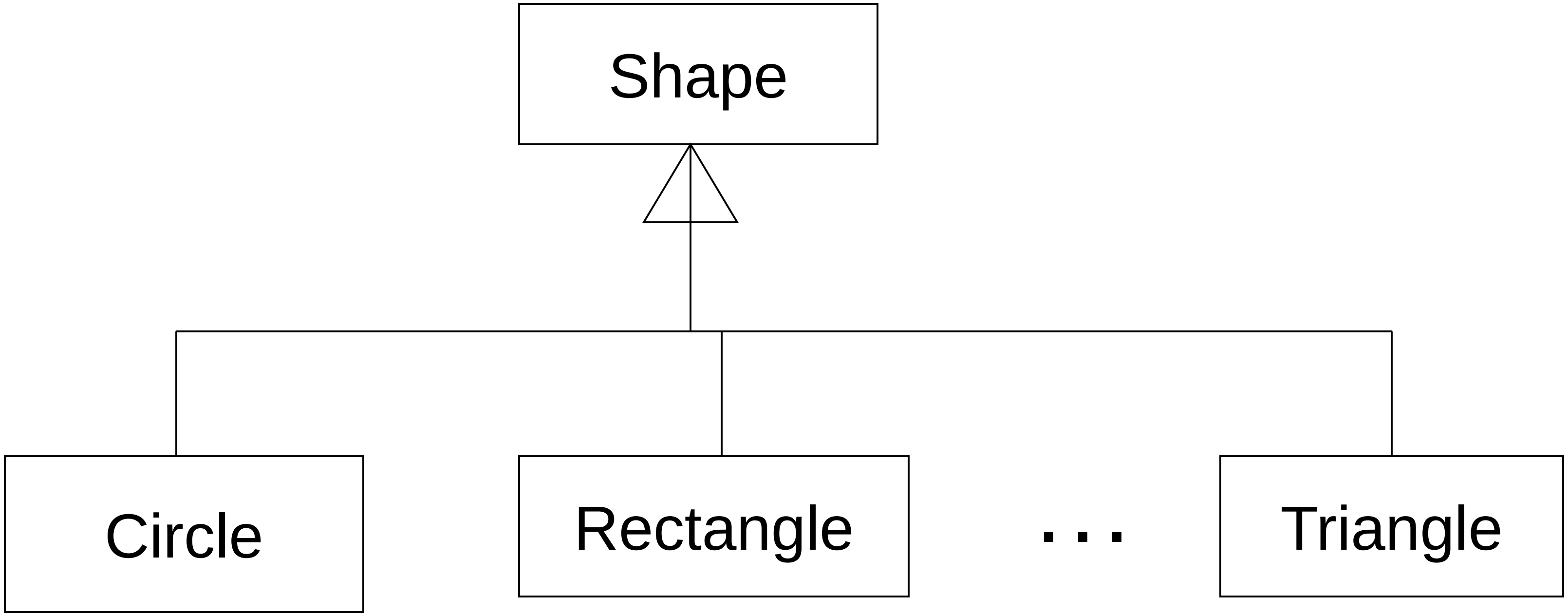
```
Time t1, *tPtr = &t1;
ExtTime et1, *etPtr = &et1;

t1.setTime(12, 30, 00);                    // static binding
et1.setTime(13, 45, 30);                   // static binding
et1.setExtTime(13, 45, 30, "CDT");         // static binding

t1.printTime( );        // static binding – Time’s printTime( )
et1.printTime( );       // static binding – ExtTime’s printTime( )

tPtr->printTime( );     // dynamic binding - Time’s printTime( )
etPtr->printTime( );    // dynamic binding - ExtTime’s printTime( )
tPtr = &et1;
tPtr->printTime( );     // dynamic binding - ExtTime’s printTime( )
```

A Common Example - Shapes



Operations: - Draw the shape
- Indicate an error has occurred
- Identify the object

Base class Shape

```
class Shape {
   public:
        virtual void draw ( ) const = 0;   // pure virtual
        virtual void error ( ) const;       // virtual
        void objectID ( ) const;            // non-virtual
};


void Shape::error ( ) const {
   cerr << “Shape error” << endl;
}

void Shape::objectID ( ) const {
    cout << “A shape” << endl;
}
```

draw( )

- draw( ) is a **pure virtual** method.
- A class with one or more pure virtual methods cannot be instantiated.
- A class that cannot be instantiated is called an **abstract class**. A class that can be instantiated is called a **concrete class**.
- Only the interface (not the implementation) of a pure virtual function is inherited by derived classes.
- Derived classes are expected to have their own implementations for the virtual method.

error( )

- error( ) is a virtual (not pure virtual) function.
- Virtual functions provide both an interface <u>and</u> an implementation to derived classes.
- The derived classes may override the inherited implementation if they wish.
- If the implementation is not overridden, the default behavior of the base class will be used.

objectID( )

- objectID( ) is a non-virtual function.
- Nonvirtual functions are provided in a base class so that derived classes inherit a function’s interface as well as a “mandatory” implementation.
- A non-virtual function specifies behavior that is not supposed to change. That is, it is not supposed to be overridden.

```
class Circle : public Shape
{
   public:
         virtual void draw( ) const;          // method for drawing a circle
         virtual void error ( ) const;        // overriding Shape::error( )
};

void Circle::draw ( ) const
{
   // code for drawing a circle
}

void Circle::error ( ) const
{
   cout << “Circle error” << endl;
}
```

```
class Rectangle : public Shape
{
   public:
         virtual void draw( ) const;      // method for drawing a rectangle
         virtual void error ( ) const;    // overriding Shape::error( )
};

void Rectangle::draw ( ) const
{
   // code for drawing a rectangle
}

void Rectangle::error ( ) const
{
   cout << “Rectangle error” << endl;
}
```

Dynamic Binding (con't)

- Now, consider these pointers:

```
Shape *pShape;
Circle *pCircle = new Circle;
Rectangle *pRectangle = new Rectangle;
```

- Each pointer has **static type** based on the way it is declared.
- A pointer's **dynamic type** is determined by the type of object to which it currently refers.

Dynamic Binding (con't)

```
pShape = pCircle;     // pShape's dynamic type is now
                      // Circle
pShape->draw( );      // calls Circle::draw and draws a
                      // circle
pShape = pRectangle;      // pShape's dynamic type is
                          // now Rectangle
pShape->draw ( );         // calls Rectangle::draw and
                          // draws a rectangle
```

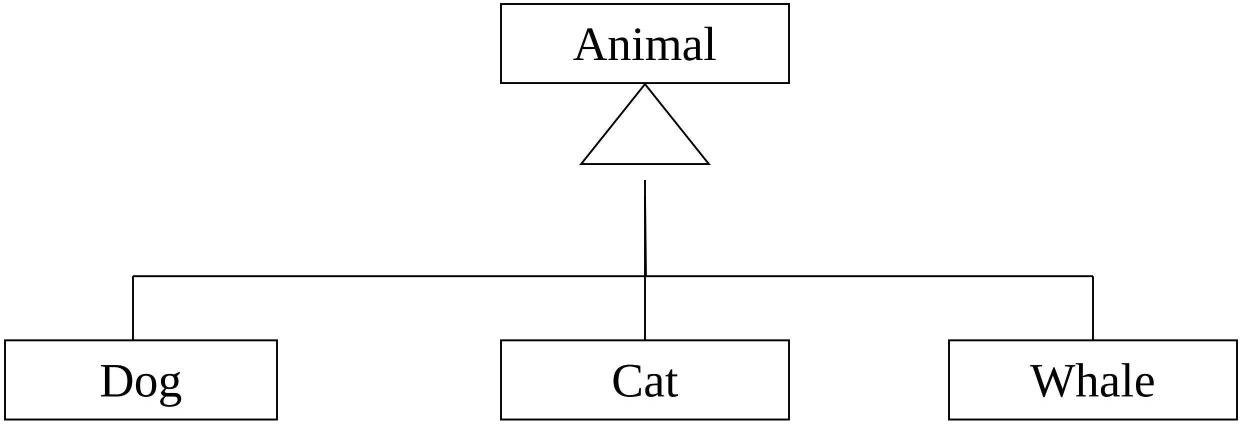
Pure Virtual Functions and Abstract Classes

- Would we ever really want to draw an object of type Shape?
- Would we ever really want to even instantiate an object of type Shape?
- Shape is really just a “place holder” in the class hierarchy.
- Therefore, Shape should be an abstract class. And draw( ) should be pure virtual.

An Array of Base Class Pointers

```
Shape *shapes[3];
shapes[0] = new Circle;
shapes[1] = new Rectangle;
shapes[2] = new Triangle;
for (int s = 0; s < 3; s++)
    shapes[s]->draw( );
```

A Simple Animal Hierarchy



```
class Animal {
  public:
        Animal ( )   {name = “no name”, nrLegs = 0;}
        Animal ( string s, int L) : name (s), nrLegs(L) {  }
        virtual ~Animal ( ) {  }   // virtual destructor
        virtual void speak ( ) const     { cout << “Hi, Bob”; }
        string getName ( ) const        { return name; }
        int getNrLegs (  ) const         { return nrLegs;}
        void setName (string s)         { name = s; }
        void setNrLegs (int legs)       { nrLegs = legs;}
   private:
        string name;
        int nrLegs;
};
```

```
class Dog: public Animal
{
   public:
      Dog( ): Animal("dog", 4), breed("dog") {  }
      Dog(string s1, string s1): Animal(s1, 4), breed(s2) {  }
      virtual ~Dog( ) {  }
      virtual void speak( ) const   {cout << "Bow Wow";}
      void setBreed(string s1)      {breed = s1;}
      string getBreed( ) const      {return breed;}
      void speak(int n) const    // overloading speak( )
           {for (int j=0; j<n; j++)
                speak( ); }
   private:
      string breed;
};
```

```
class Whale : public Animal
{
   public:
         Whale ( ) {  }
         Whale(string n) : Animal(n, 0) {  }
         virtual ~Whale ( ) {  }
   private:
};
```

```
int main  ( ) {
   Animal anAnimal (“Homer”, 2);
   Dog aDog (“Fido”, “mixed”);
   Whale aWhale (“Orka”);
   anAnimal.speak( );         // Animal’s speak( )
   aDog.speak( );             // Dog’s speak( ) -- overriding
   aDog.speak( 4 );           // Dog’s speak( ) -- overloading
   aWhale.speak( ) ;          // Animal’s speak( ) -- inherited

   Animal *zoo[ 3 ];
   zoo[0] = new Animal;
   zoo[1] = new Dog (“Max”, “Terrier”);
   zoo[2] = new Whale;
   for (int a = 0; a < 3; a++) {
         cout << zoo[a]->getName( ) << “ says: ” ;
         zoo[a] -> speak( );     // dynamic binding -- polymorphism
   }
   return 0;
}
```

Polymorphism and Non-Member Functions

- An important use of polymorphism is the writing of non-member functions to deal with all classes in an inheritance hierarchy.
- This is accomplished by defining the function parameters as pointers (or references) to base class objects, then having the caller pass in a pointer (or reference) to a derived class object.
- Dynamic binding calls the appropriate method.
- These functions are often referred to as **polymorphic functions**.

drawShape( ) With a Pointer

```
void drawShape (Shape *sp)
{
   cout << “I am ” ;
   sp -> objectID ( );
   sp -> draw ( );
   if (something bad)
         sp->error ( );

}
```

What is the output if drawShape( ) is passed:

- a pointer to a Circle object?
- a pointer to a Rectangle object?

drawShape( ) Via Reference

```
void drawShape (Shape& shape)
{
    cout << "I am " ;
    shape.objectID ( );
    shape.draw ( );
    if (something bad)
        shape.error ( );

}
```

What is the output if drawShape is passed:

- a Circle object?
- a Rectangle object?

Don’t Pass by Value

A function that has a base class parameter passed by value should only be used with base class objects because:

- The function isn’t polymorphic. Polymorphism only occurs with parameters passed by pointer or reference.
- Even though a derived class object can be passed to such a function (because each instance of D is-a B), none of the derived class methods or data members can be used in that function.

Don’t Pass by Value (con’t)

```
void drawShape (Shape shape)    // member slicing!
{
    cout << “I am ” ;
    shape.objectID ( );
    shape.draw ( );
    if (something bad)
         shape.error ( );

}
```

What is the output if drawShape is passed:

- a Circle object?
- a Rectangle object?

Calling Virtual Methods From Within Other Methods

- Suppose virtual drawMe( ) is added to Shape and inherited by Rectangle without being overridden.

```
void Shape::drawMe ( void)
{
	cout << "drawing: " << endl;
	draw( );
}
```

Calling Virtual Methods From Within Other Methods (con't)

Which draw( ) gets called from within drawMe( )?

Rectangle r1;

r1.drawMe( );

The Rectangle version of draw( ), even though drawMe( ) was only defined in Shape.

Why? Because inside of drawMe( ), the call to draw( ) is really **this->draw( );** and since a pointer is used, we get the desired polymorphic behavior.

Polymorphism and Destructors

- A problem – If an object (with a non-virtual destructor) is explicitly destroyed by applying the **delete** operator to a base class pointer, the base class destructor is invoked.

```
Shape *sp = new Circle;
delete sp;              // calls Shape's destructor
                        // if the destructor is not virtual
```

- The Circle object is not “fully” destroyed.

Polymorphism and Destructors (cont'd)

- Solution -- Declare a virtual destructor for any base class with at least one virtual function.

```
virtual ~Shape ( );
```

- Now,

```
Shape *sp = new Circle;
delete sp;
```

will invoke the Circle destructor, which in turn invokes the Shape destructor.

Designing a Base Class With Inheritance and Polymorphism In Mind

For the base class

1. Identify the set of operations common to all the derived classes.
2. Identify which operations are type-independent (these become (pure) virtual to be overridden in derived classes).
3. Identify the access level (public, private, protected) of each operation.