OOP/Computer Programming

Instructor: Dr. Danish Shehzad

Multiple Inheritance

- Data member must be used with care when dealing with more then one level of inheritance

Example

```
class A{
        protected:
                int n;
}
Class B: public A{
        };
Class C: public A{
        };
Class D: public B, public C{
        Public();
Void set()
{
        n=10;
        Cout<<”n=”<<n<<endl;
}};
void main()
{
        D obj;
        obj.set();
}
```

```
class A{
        protected:
                int n;
}
Class B: virtual public A{
        };
Class C: virtual public A{
        };
Class D: public B, public C{
        Public();
Void set()
{
        n=10;
        Cout<<"n="<<n<<endl;
}};
void main()
{
        D obj;
        Obj.set();
}
```

Example

```
class Vehicle{
protected:
  int weight;
};
class LandVehicle : public Vehicle{
};
class WaterVehicle : public Vehicle{
};
```

Example

```
class AmphibiousVehicle:
                public LandVehicle,
                public WaterVehicle{
public:
   AmphibiousVehicle(){
       LandVehicle::weight = 10;
       WaterVehicle::weight = 10;
   }
};
```

- There are multiple copies of data member weight

Memory View

Data Members of Vehicle	Data Members of Vehicle
Data Members of LandVehicle	Data Members of WaterVehicle
Data Members of AmphibiousVehicle	

Virtual Inheritance

- In virtual inheritance there is exactly one copy of the anonymous base class object

Example

```
class Vehicle{
protected:
  int weight;
};
class LandVehicle : public virtual
  Vehicle{ };

class WaterVehicle :public virtual
  Vehicle{ };
```

Example

```
class AmphibiousVehicle:
            public LandVehicle,
            public WaterVehicle{
public:
  AmphibiousVehicle(){
     weight = 10;
  }
};
```

Memory View

<table>
<tr><td colspan="2">Data Members of Vehicle</td></tr>
<tr><td>Data Members of LandVehicle</td><td>Data Members of WaterVehicle</td></tr>
<tr><td colspan="2">Data Members of AmphibiousVehicle</td></tr>
</table>

Virtual Inheritance

- Virtual inheritance must be used when necessary
- There are situation when programmer would want to use two distinct data members inherited from base class rather then one

Today's Lecture Objectives

- Polymorphism in C++
- Pointers to derived classes
- Introduction to virtual functions

Pointers to Objects

```
class Test{
        private:
                int n;
        public:
                void in()
                {cout<<"Enter number! ";
                        cin>>n;}
                void out()
                {cout<<"The value of n ="<<n;}

};
int main()
{
        Test *ptr;
        ptr=new Test;
        ptr->in();
        ptr->out();
        return 0;
}
```

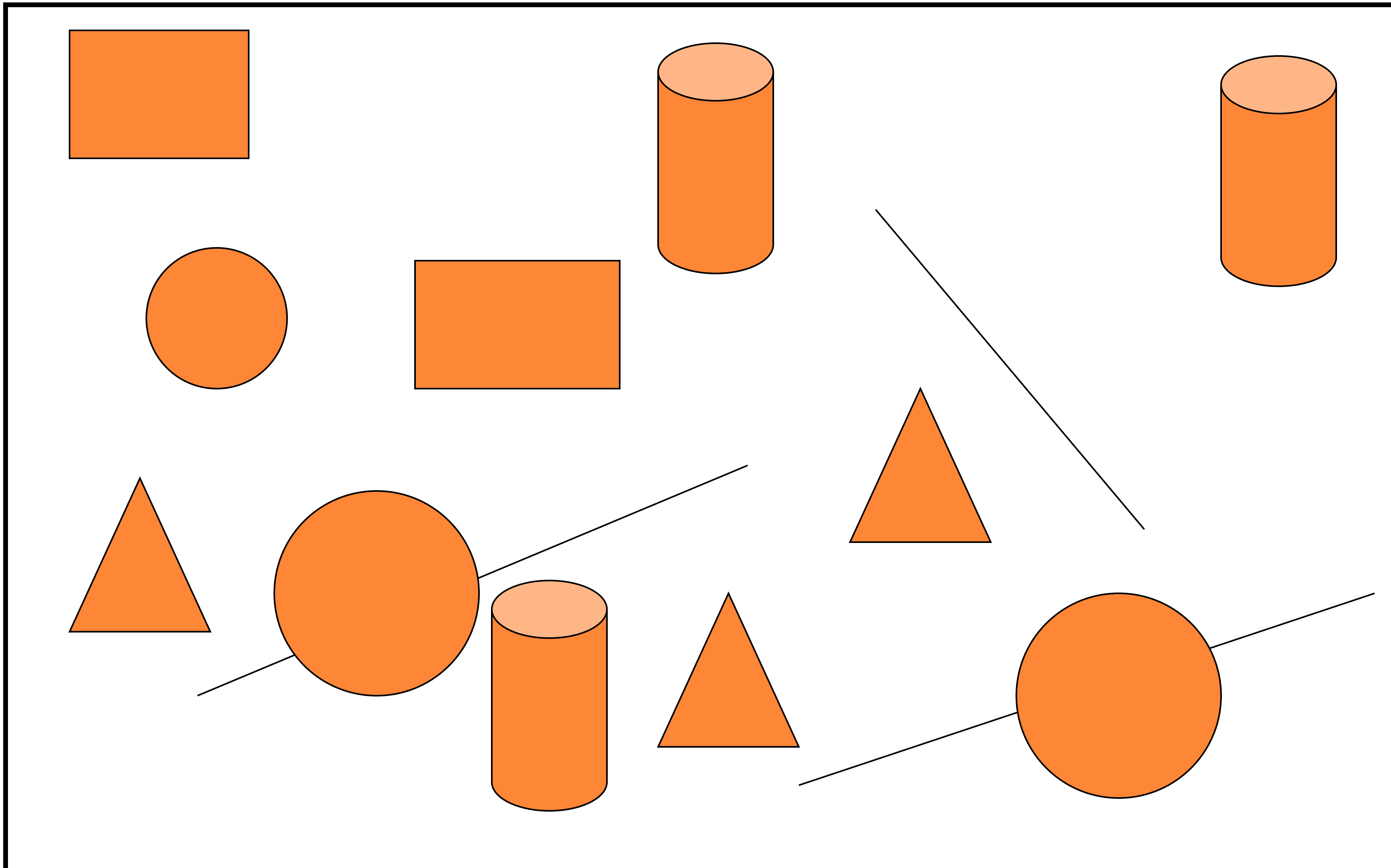
Polymorphism

- The Greek word **polymorphism** means one name, many forms.
- Typically, **polymorphism** occurs when there is a hierarchy of classes and they are related by inheritance.
- **C++ polymorphism means that a call to a member function will cause a different function to be executed depending on the type of object that invokes the function.**

- ***Static polymorphism*: It can be achieved by using overloading. It is defined at compilation time.**

- ***Dynamic polymorphism*: It can be implemented by using inheritance. It is implemented at runtime.**

Graphics Drawing Software



Graphics Drawing Software Classes

- **Line**
  - Properties:- X-Y Coordinates, Length, Color
  - Actions:- Draw Function, Change Color Function, Get Area Function.
- **Circle**
  - Properties:- X-Y Coordinates, Radius, Color
  - Actions:- Draw Function, Change Color Function, Get Area Function.
- **Rectangle**
  - Properties:- X-Y Coordinates, Width, Height, Color
  - Actions:- Draw Function, Change Color Function, Get Area Function.
- **Cylinder**
  - Properties:- X-Y Coordinates, Radius, Height, Color
  - Actions:- Draw Function, Change Color Function, Get Area Function.
- **Triangle**
  - Properties:- X-Y Coordinates, Length, Width, Color
  - Actions:- Draw Function, Change Color Function, Get Area Function.

class Line

```
{
protected:
        int x,y;
public:
        Line(int ,int );
        void draw(void);
        int GetArea (void);
};
Line::Line (int a,int b)
{
        x=a;
        y=b;
}
void Line::draw(void)
{
        cout << “\n Line Drawing code”;
}
int Line::GetArea (void)
{
        cout << “\nLine Area “;
}
```

```
class Rectangle: public Line
{
protected:
            int Width, Height;
public:
            Rectangle(int, int , int , int );
            void draw(void);
            int GetArea (void);
};
Rectangle::Rectangle(int a,int b, int c, int d) : Line (a, b )
{
            Width = c;          Height = d;
}
void Rectangle::draw(void)
{
             cout << “Rectangle drawing code”;
}
int Rectangle::GetArea (void)
{
            cout << “Rectangle area code”;
}
```

```
class Circle: public Line
{
protected:
        int radius;
public:
        Circle(int ,int, int );
        void draw(void);
        int GetArea (void);
};
Circle::Circle(int a,int b, int c) : Line(a, b)
{
        radius = c;
}
void Circle::draw(void)
{
        cout << "Circle drawing code";
}
int Circle::GetArea (void)
{
        cout << "Circle area code";
}
```

```
int main ( void )
{
    Triangle t1 (3, 4, 5, 19 );
    Circle c1 (3, 4, 5 );
    Rectangle r1 ( 3, 4, 10 , 20 );
    Cylinder c2 ( 3, 4, 5, 10 );

    t1.draw ();
    cout << “The area is “ << t1.GetArea ( );

    c1.draw ();
    cout << “The area is “ << c1.GetArea ();

    r1.draw ();
    cout << “The area is “ << r1.GetArea ();

    c2.draw ();
    cout << “The area is “ << c2.GetArea ();
    return 0;
}
```

Pointers to Derived Classes

- C++ allows base class pointers to point to derived class objects.
- Allows us to have –
  - class base { … };
  - class derived : public base { … };
- Then we can write –
  - base *p1;
  - derived d_obj;
  - p1 = &d_obj;

  - base *p2 = new derived;

Pointers to Derived Classes (contd.)

- Using a base class pointer (pointing to a derived class object) we can access only those members of the derived object **that were inherited from the base**.
- This is because the **base pointer** has knowledge only of the base class.
- It knows nothing about the members added by the derived class.

```
class base {
   public:
          void show()
          {
                    cout <<
   "base\n";
          }
};

class derived : public base {
   public:
          void show()
          {
          cout << "derived\n";
          }
};
```

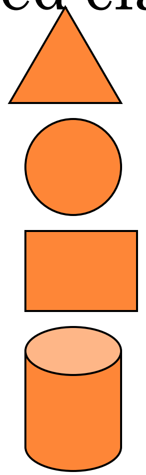
```
int main()
{
   base b1;
   b1.show(); // base
   derived d1;
   d1.show(); // derived

   base *pb = &b1;
   pb->show();          // base
   pb = &d1;
  pb->show();           // base
   return 0;
}
```

POINTERS TO DERIVED CLASSES

- With the help of pointers to derived classes, we can create an array of base class objects, and that array can hold objects of different derived classes

```
Line *p[4];
p[0] = new Triangle (3, 4, 5, 19 );
p[1] = new Circle (3, 4, 5 );
p[2] = new Rectangle ( 3, 4, 10 , 20 );
p[3] = new Cylinder ( 3, 4, 5, 10 );
for ( int loop = 0; loop < 4; loop ++ )
{       p[loop]->draw ();
        cout << "The area is " << p[loop]->GetArea ( );
}
```



Pointers to Derived Classes (contd.)

- While it allows a base class pointer to point to a derived object, the reverse is not true.
  - base b1;
  - derived *pd = &b1; // compiler error

```
class A{
        public:
                void show()
                {
                cout<<"Parent class A ";}
};
Class B: public A{
        public:
                void show()
                {
                cout<<"Parent class B ";}
};
Class C: public A{
        public:
                void show()
                {
                cout<<"Parent class C ";
                }
};
```

```
int main()
{
        A obj1;
        B obj2;
        C obj3;
        A *ptr;
        Ptr = &obj1;
        Ptr->show();
Ptr = &obj2;
        Ptr->show();
Ptr = &obj3;
        Ptr->show();
}
```

Introduction to Virtual Functions

- A virtual function is a member function that is declared within a base class and redefined (called ***overriding***) by a derived class.
- It implements the "one interface, multiple methods" philosophy **that underlies polymorphism**.
- The keyword **virtual** is used to designate a member function as virtual.
- Supports run-time polymorphism with the help of base class pointers.

Introduction to Virtual Functions (contd.)

```
class base {
public:
   virtual void show() {
      cout << "base\n";
   }
};
class derived : public
   base {
public:
   void show() {
      cout << "derived\n";
   }
};
```

```
int main() {
   base b1;
   b1.show();
   derived d1;
   d1.show();

   base *pb = &b1;
   pb->show();

   pb = &d1;
   pb->show();
}
```

Introduction to Virtual Functions (contd.)

```
class base {
public:
virtual void show() {
      cout << "base\n";
   }
};
class d1 : public base {
public:
   void show() {
      cout << "derived-1\n";
   }
};
```

```
class d2 : public base {
public:
   void show() {
      cout << "derived-2\n";
   }
};
int main() {
   base *pb;
   d1 od1;
   d2 od2;
   int n;
   cin >> n;
   if (n % 2) pb = &od1;
   else pb = &od2;
   pb->show(); // guess what ??
}
```

Dynamic Binding

Run-time polymorphism

Static vs. Dynamic Binding

There are two types of binding in C++: static (or early) binding and dynamic (or late) binding.

1. The static binding happens at the compile-time and dynamic binding happens at the runtime. Hence, they are also called early and late binding respectively.

2. In static binding, the function defination and the function call are linked during the compile-time whereas in dynamic binding the function calls are not resolved until runtime. So they are not bound until runtime.

3. Static binding happens when all information needed to call a function is available at the compile-time. Dynamic binding happens when all information needed for a function call cannot be determined at compile-time.

4. Static binding can be achieved using the normal function calls, function overloading and operator overloading while dynamic binding can be achieved using the virtual functions.

5. Since all information needed to call a function is available before runtime, static binding results in faster execution of a program. Unlike static binding, a function call is not resolved until runtime for later binding and this results in somewhat slower execution of code.

6. The major advantage of dynamic binding is that it is flexible since a single function can handle different type of objects at runtime. This significantly reduces the size of the codebase and also makes the source code more readable.