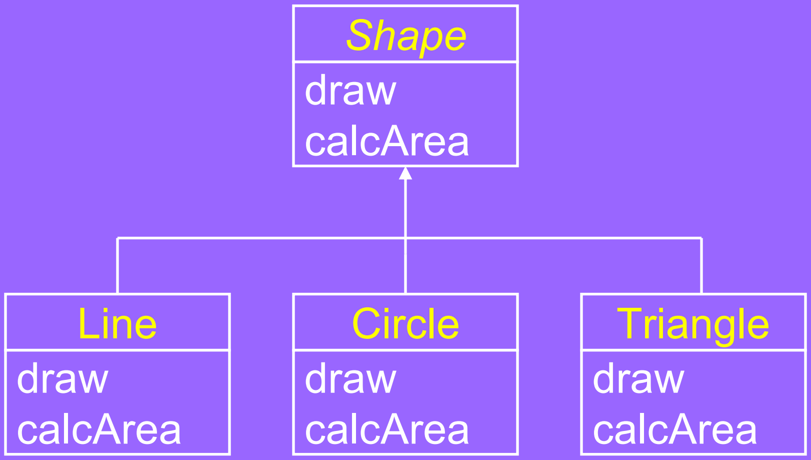
Object-Oriented Programming (OOP)
Lecture No. 29

Abstract Class

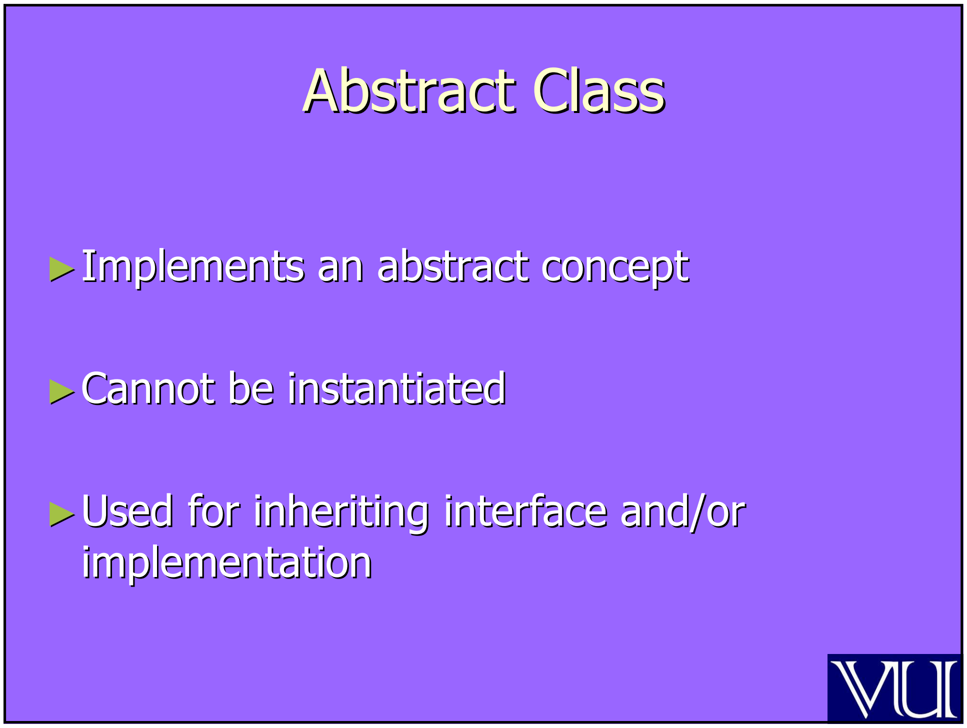


Shape
draw
calcArea

Line
draw
calcArea

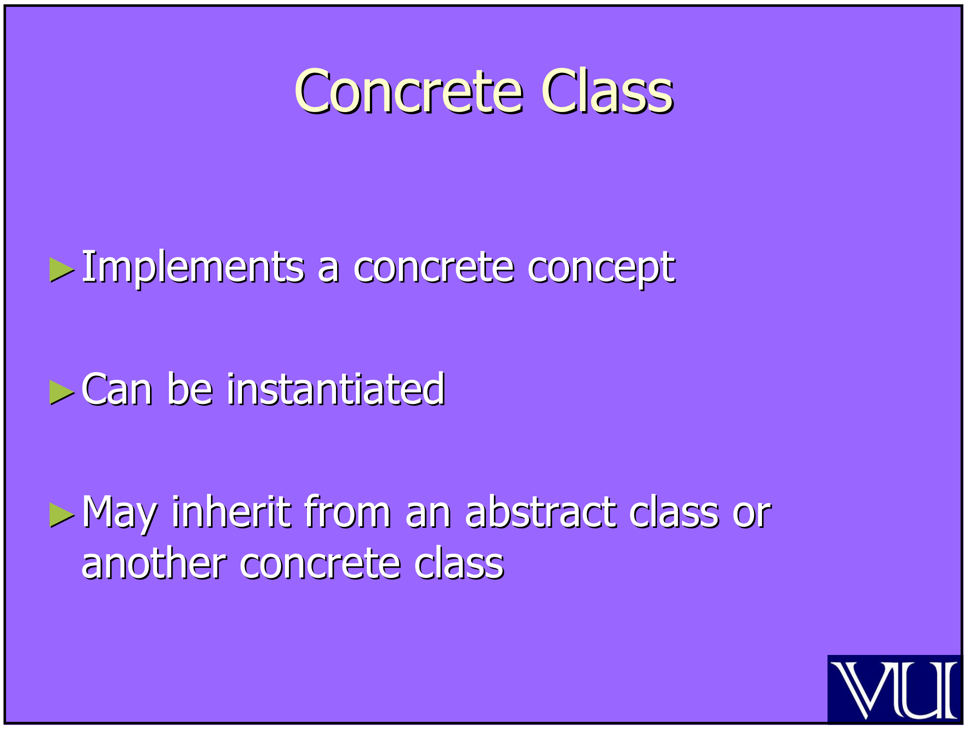
Circle
draw
calcArea

Triangle
draw
calcArea



Abstract Class

- Implements an abstract concept
- Cannot be instantiated
- Used for inheriting interface and/or implementation



Concrete Class

- Implements a concrete concept
- Can be instantiated
- May inherit from an abstract class or another concrete class

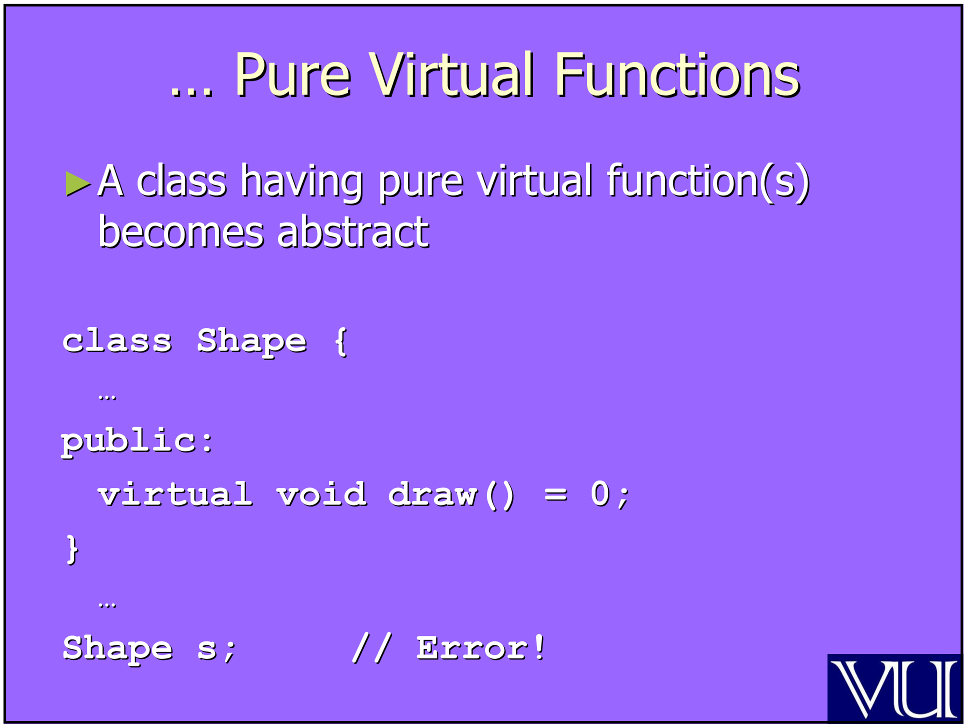
Abstract Classes in C++

- In C++, we can make a class abstract by making its function(s) pure virtual
- Conversely, a class with no pure virtual function is a concrete class

Pure Virtual Functions function

- A pure virtual represents an abstract behavior and therefore may not have its implementation (body)
- A function is declared pure virtual by following its header with "= 0"

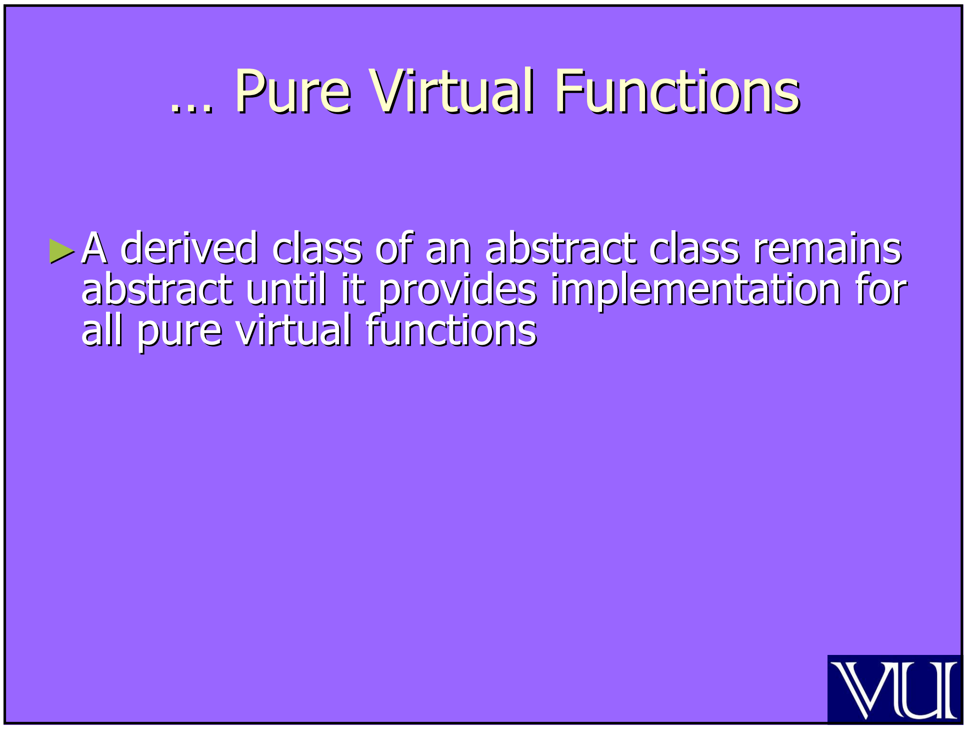
```
virtual void draw() = 0;
```



… Pure Virtual Functions

- A class having pure virtual function(s) becomes abstract

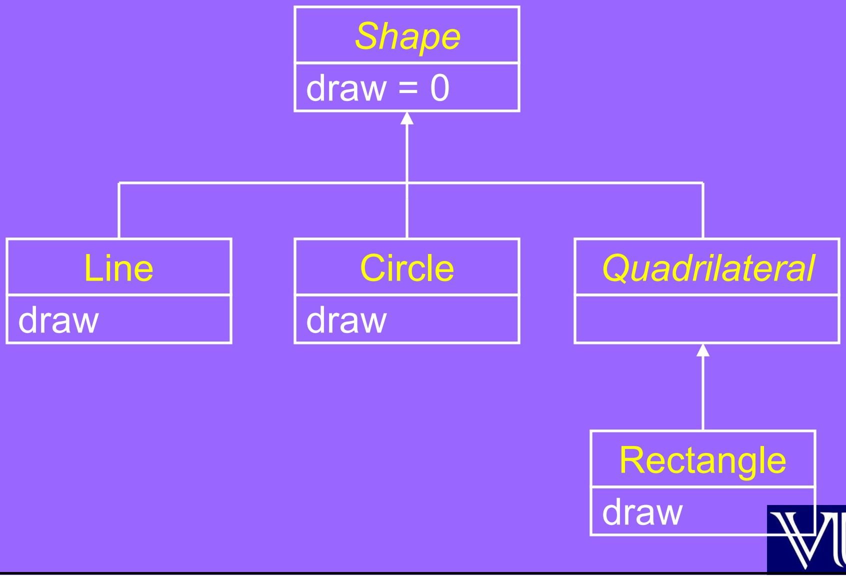
```
class Shape {
  …
public:
  virtual void draw() = 0;
}
  …
Shape s;      // Error!
```



… Pure Virtual Functions

- A derived class of an abstract class remains abstract until it provides implementation for all pure virtual functions

Shape Hierarchy



Shape

draw = 0

Line

draw

Circle

draw

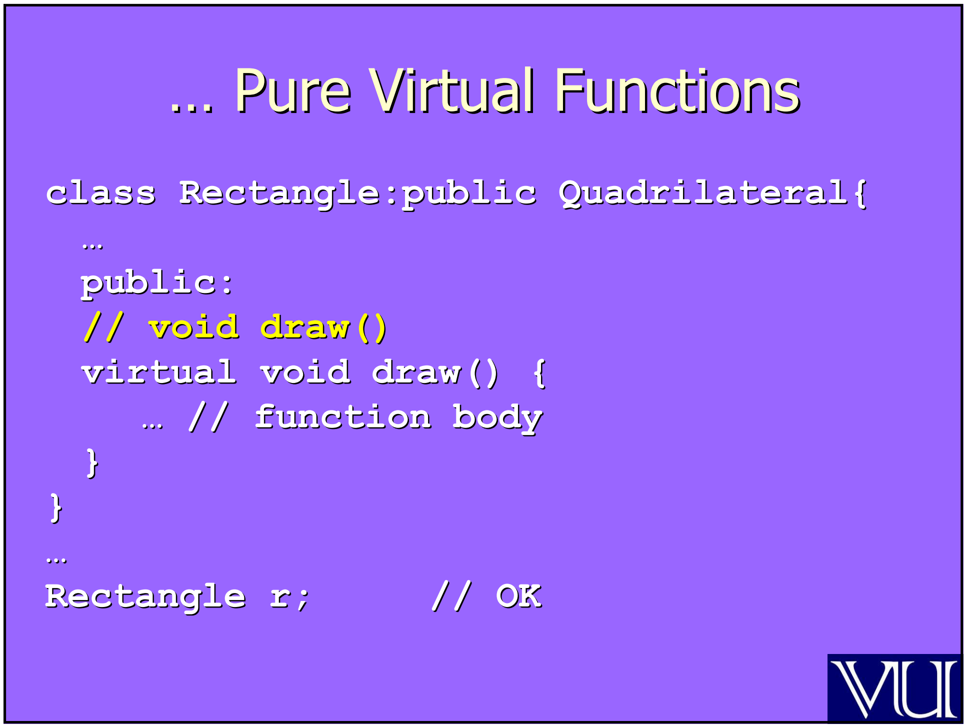
Quadrilateral

Rectangle

draw

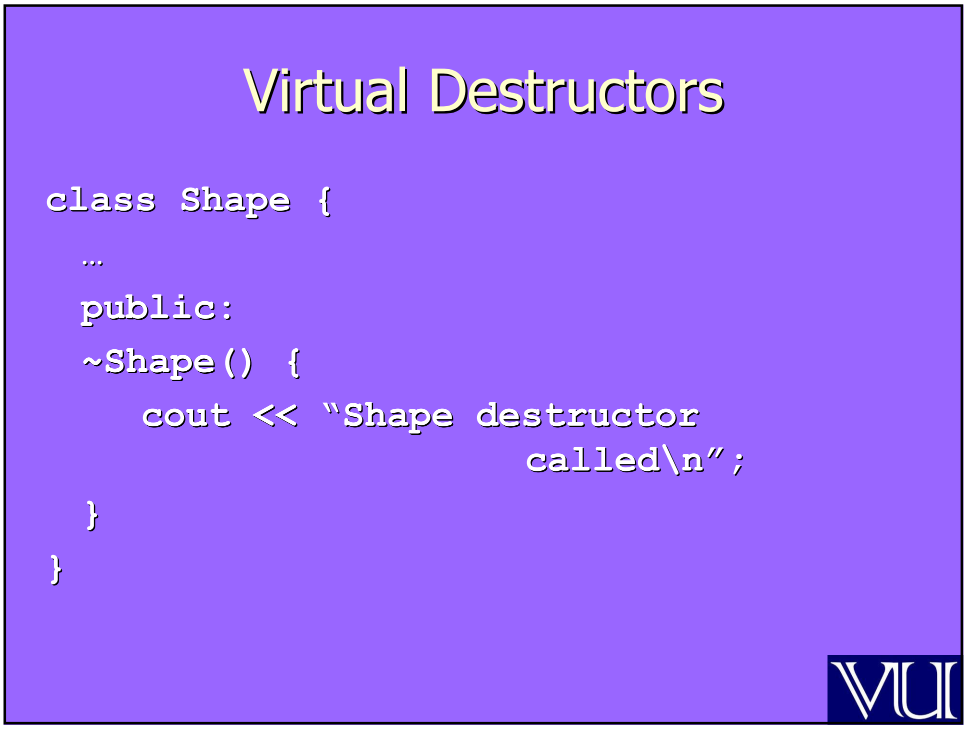
… Pure Virtual Functions

```
class Quadrilateral : public Shape {
  …
  // No overriding draw() method
}
…
Quadrilateral q; // Error!
```



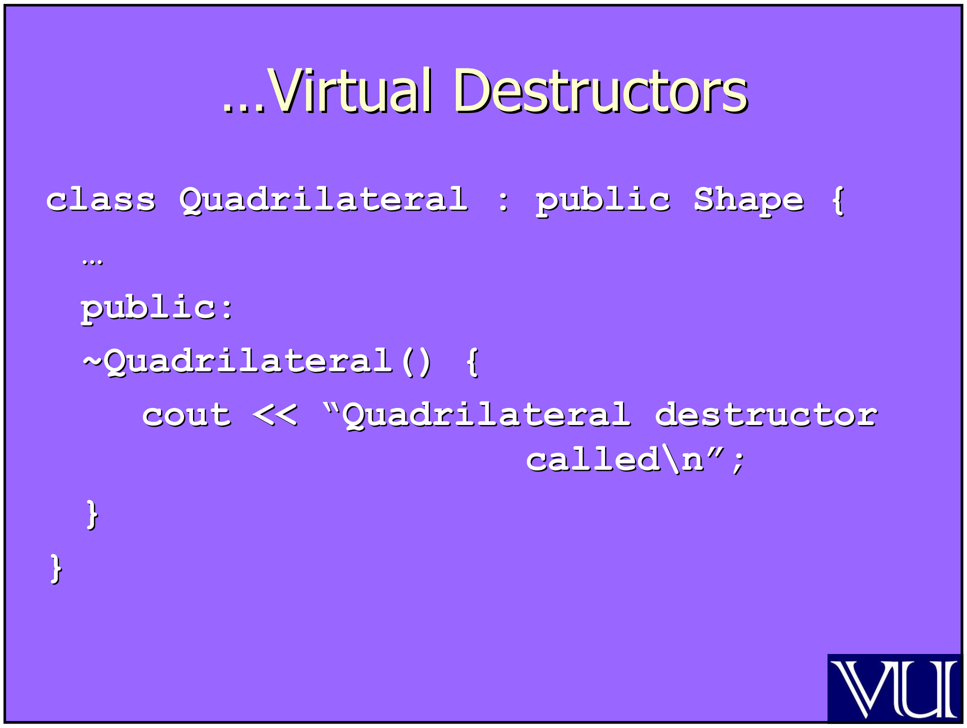
… Pure Virtual Functions

```
class Rectangle:public Quadrilateral{
  …
  public:
  // void draw()
  virtual void draw() {
     … // function body
  }
}
…
Rectangle r;      // OK
```



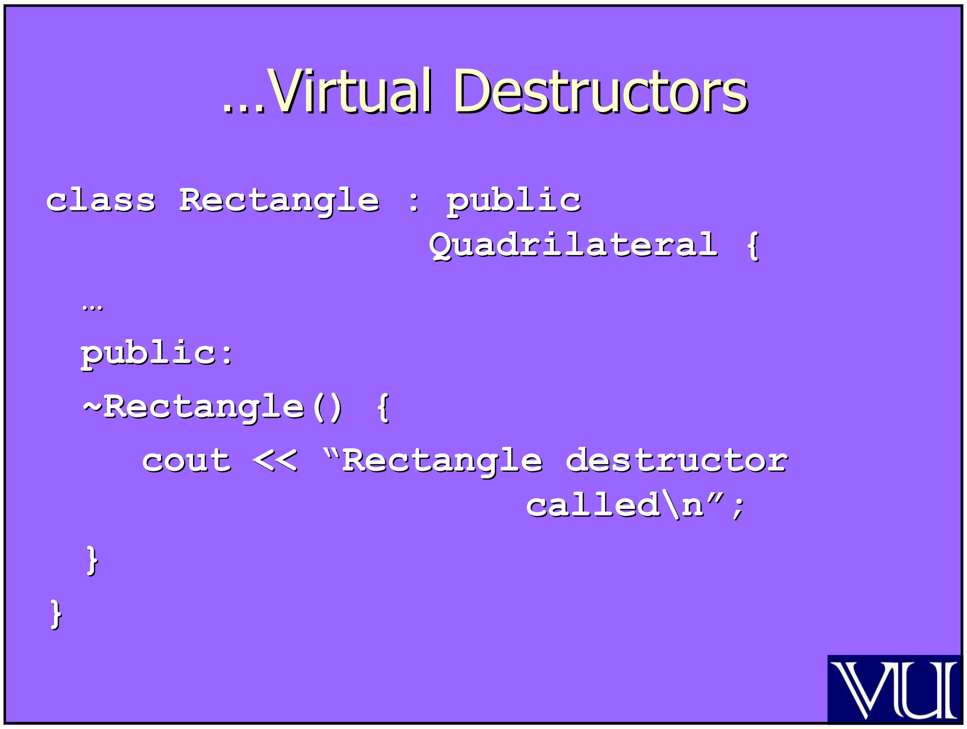
Virtual Destructors

```
class Shape {
  …
  public:
  ~Shape() {
     cout << "Shape destructor
                       called\n";
  }
}
```



…Virtual Destructors

```
class Quadrilateral : public Shape {
  …
  public:
  ~Quadrilateral() {
     cout << "Quadrilateral destructor
                      called\n";
  }
}
```



…Virtual Destructors

```
class Rectangle : public
                     Quadrilateral {
  …
  public:
  ~Rectangle() {
     cout << "Rectangle destructor
                      called\n";
  }
}
```

…Virtual Destructors

- When delete operator is applied to a base class pointer, base class destructor is called regardless of the object type

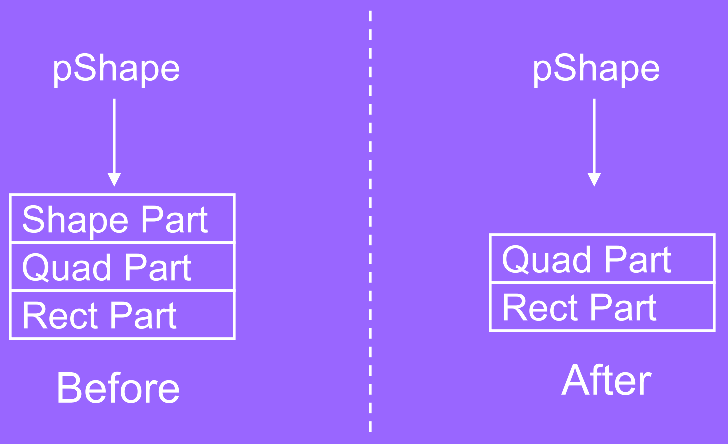
…Virtual Destructors

```
int main() {
  Shape* pShape = new Rectangle();
  delete pShape;
  return 0;
}
```

- Output

```
Shape destructor called
```

Result



pShape

Shape Part
Quad Part
Rect Part

Before

pShape

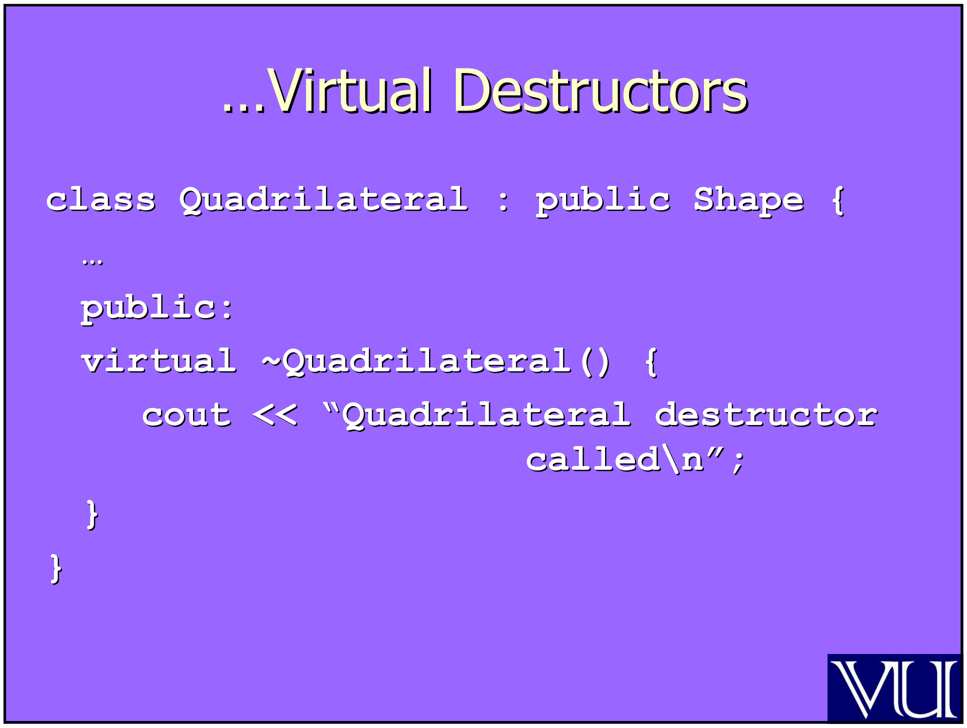
Quad Part
Rect Part

After

Virtual Destructors

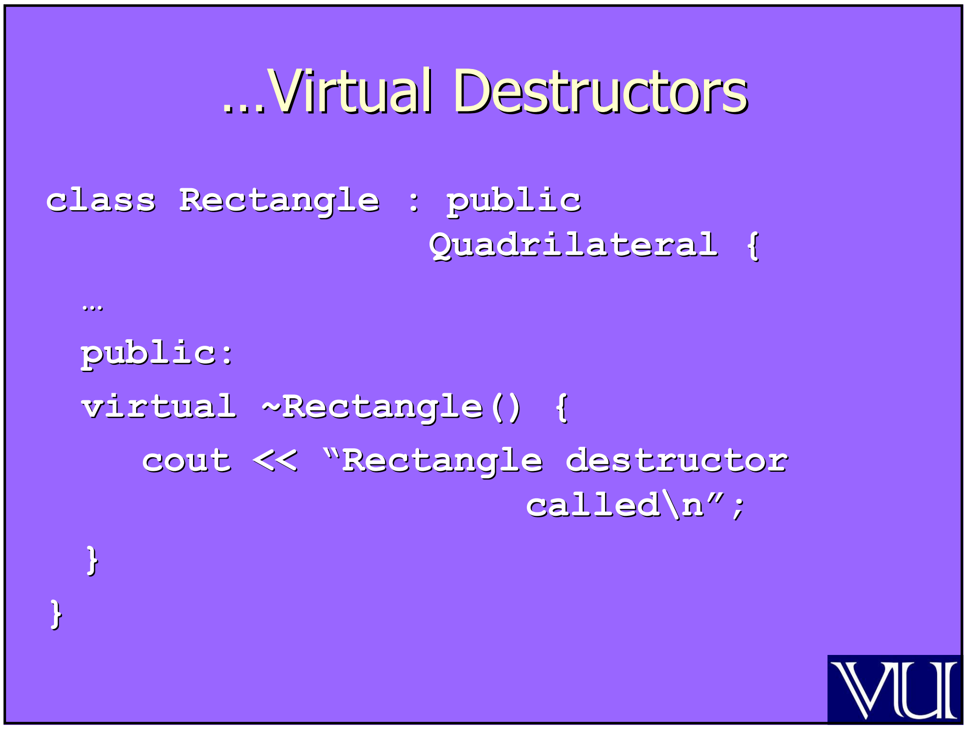
- Make the base class destructor virtual

```
class Shape {
  …
  public:
  virtual ~Shape() {
     cout << "Shape destructor
                 called\n"; }
}
```



…Virtual Destructors

```
class Quadrilateral : public Shape {
  …
  public:
  virtual ~Quadrilateral() {
     cout << “Quadrilateral destructor
                      called\n”;
  }
}
```



…Virtual Destructors

```
class Rectangle : public
                  Quadrilateral {
  …
  public:
  virtual ~Rectangle() {
     cout << “Rectangle destructor
                      called\n”;
  }
}
```

…Virtual Destructors

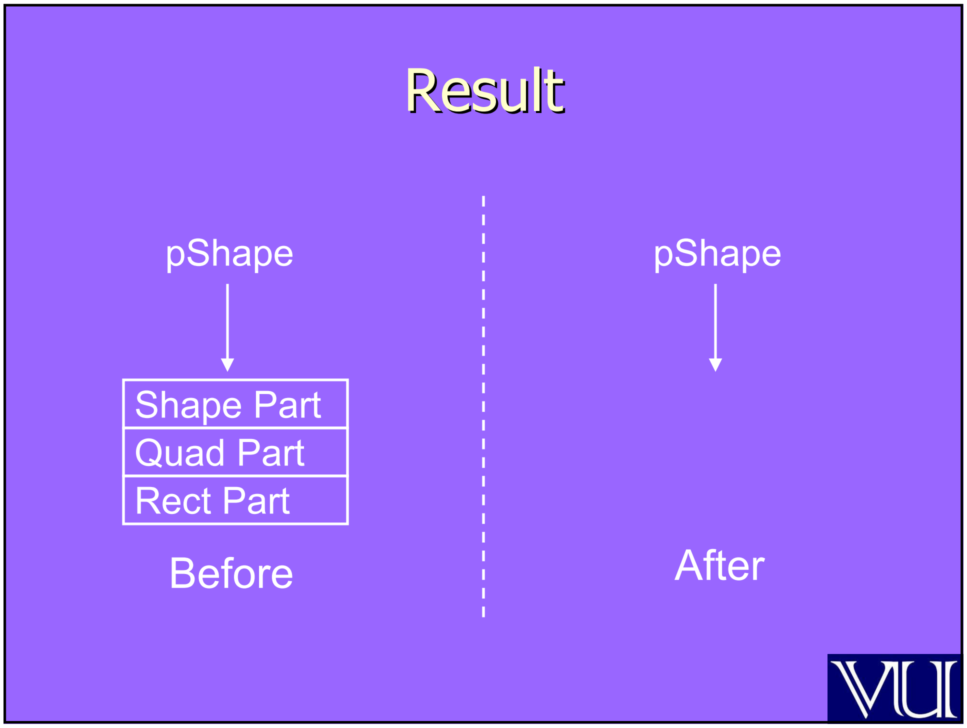
- Now base class destructor will run after the derived class destructor

…Virtual Destructors

```
int main() {
  Shape* pShape = new Recrangle();
  delete pShape;
  return 0;
}
```

- Output

```
Rectangle destructor called
Quadilateral destructor called
Shape destructor called
```



Result

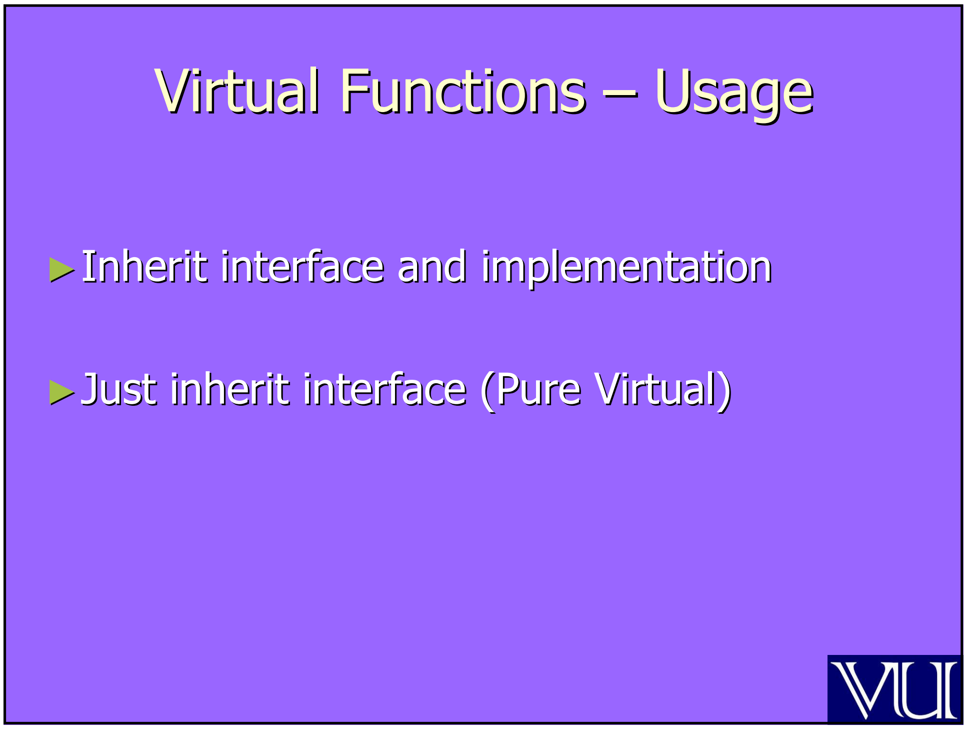
pShape

Shape Part
Quad Part
Rect Part

Before

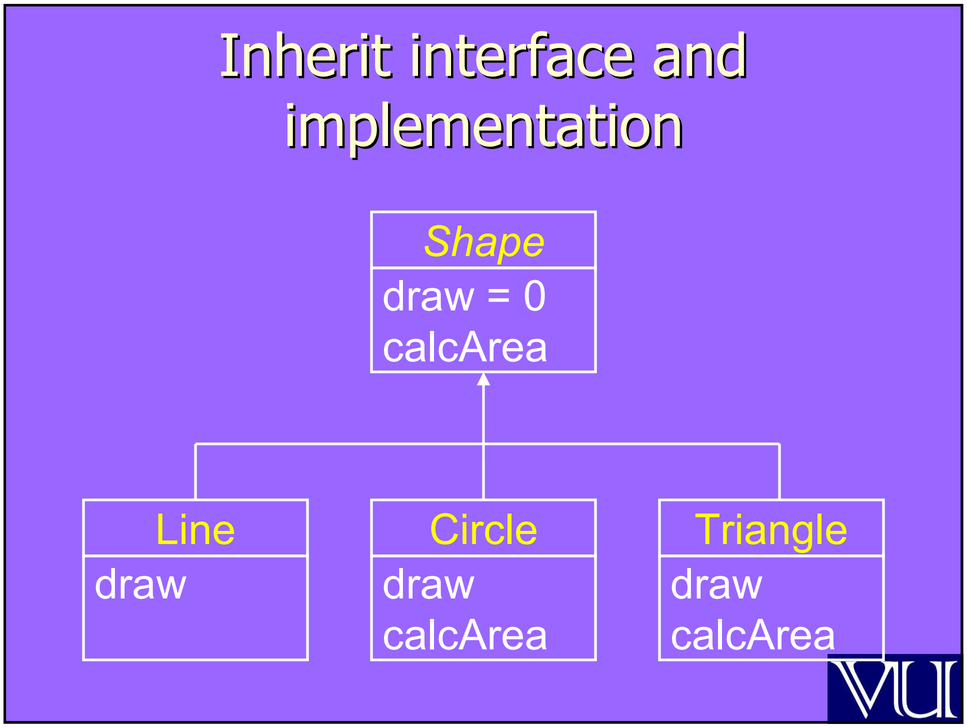
pShape

After



Virtual Functions – Usage

- Inherit interface and implementation
- Just inherit interface (Pure Virtual)



Inherit interface and implementation

Shape

draw = 0

calcArea

Line

draw

Circle

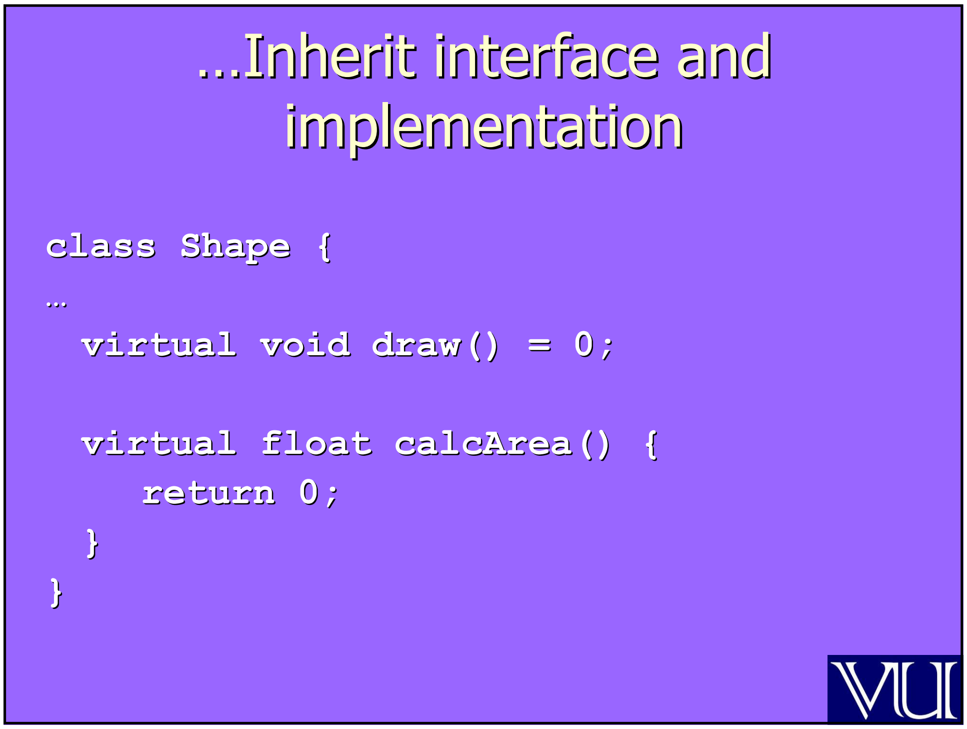
draw

calcArea

Triangle

draw

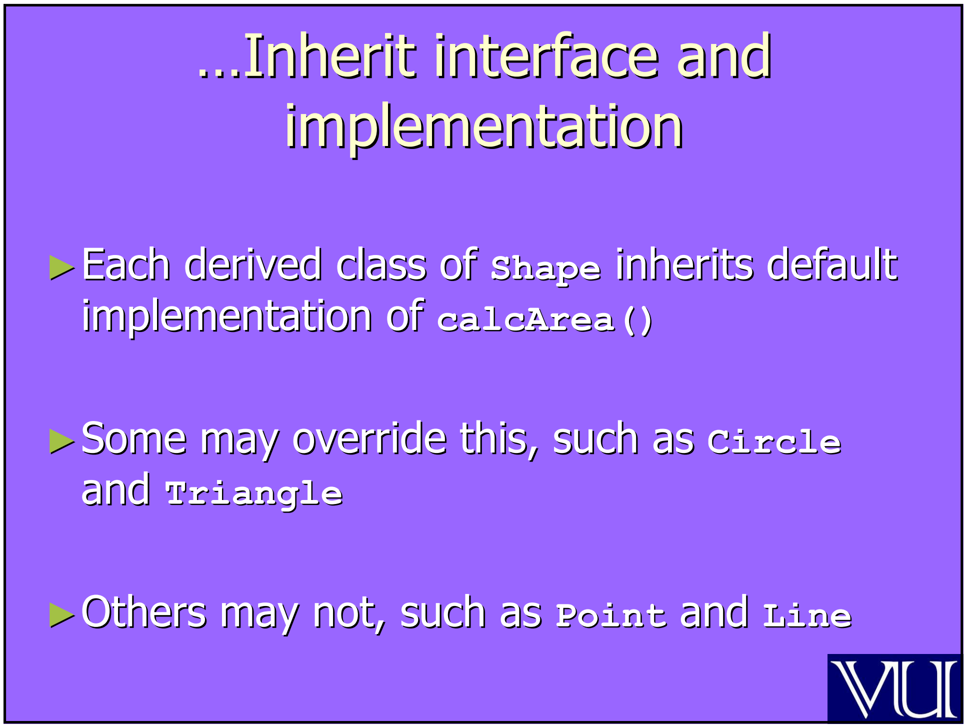
calcArea



…Inherit interface and implementation

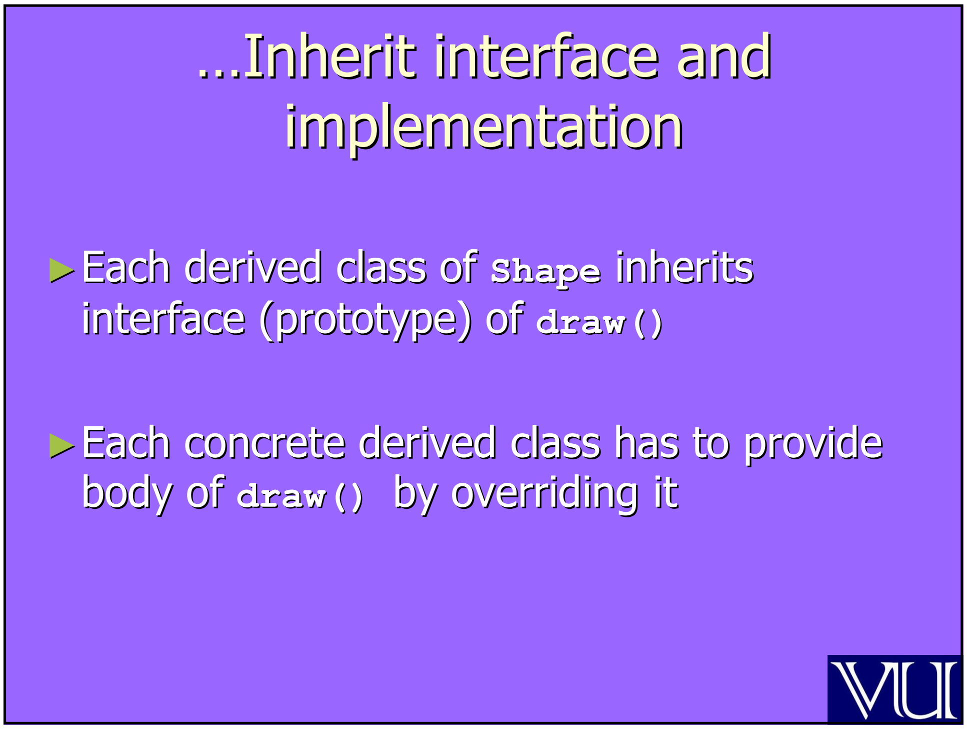
```
class Shape {
…
  virtual void draw() = 0;

  virtual float calcArea() {
     return 0;
  }
}
```



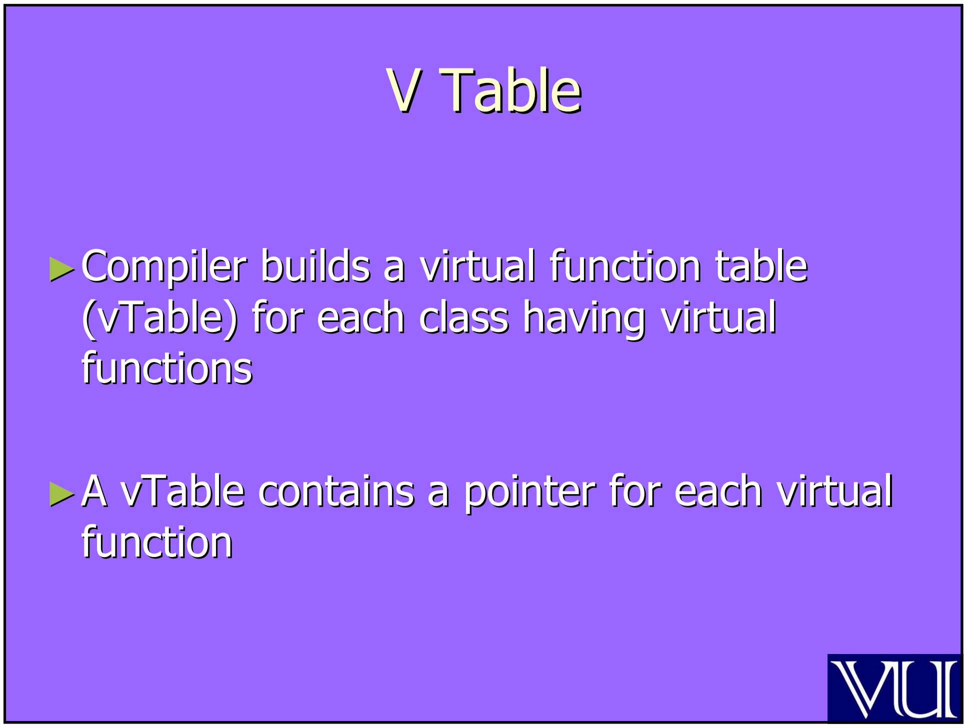
…Inherit interface and implementation

- Each derived class of `Shape` inherits default implementation of `calcArea()`
- Some may override this, such as `Circle` and `Triangle`
- Others may not, such as `Point` and `Line`



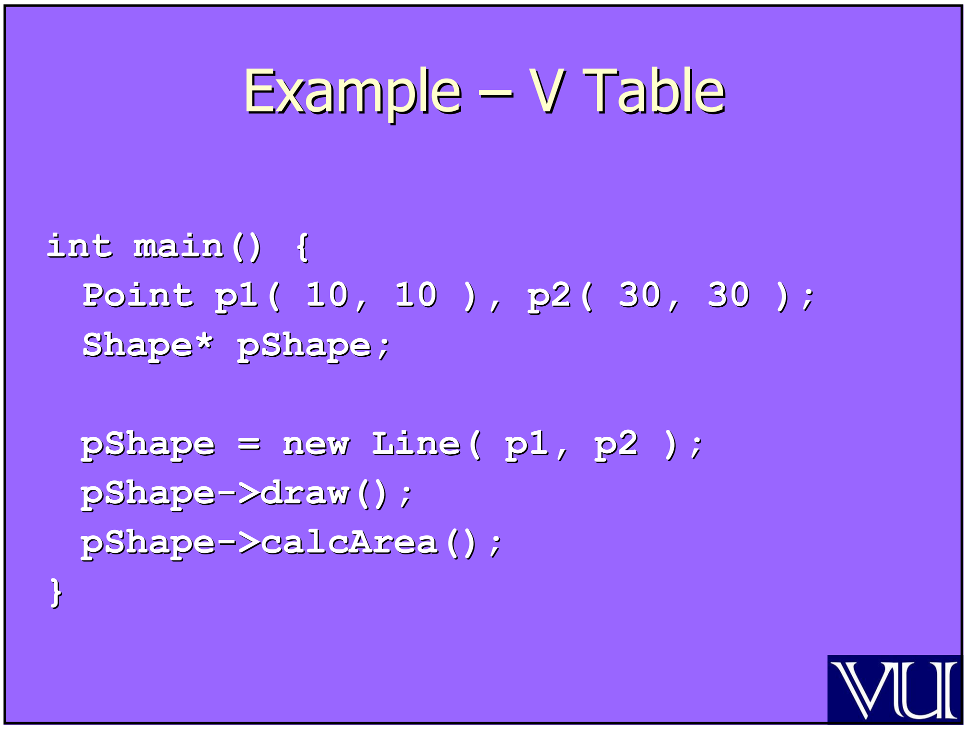
…Inherit interface and implementation

- Each derived class of `Shape` inherits interface (prototype) of `draw()`
- Each concrete derived class has to provide body of `draw()` by overriding it



V Table

- Compiler builds a virtual function table (vTable) for each class having virtual functions
- A vTable contains a pointer for each virtual function

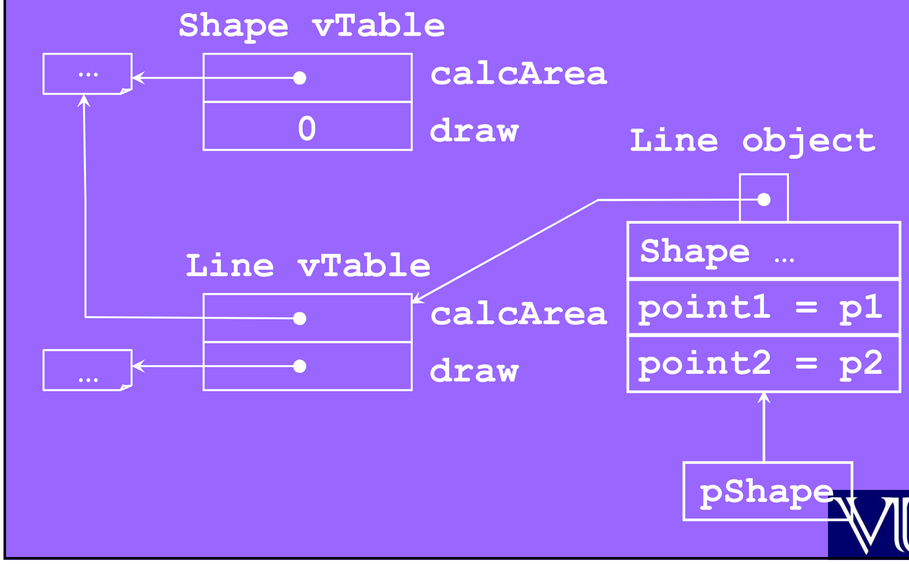


Example – V Table

```
int main() {
  Point p1( 10, 10 ), p2( 30, 30 );
  Shape* pShape;

  pShape = new Line( p1, p2 );
  pShape->draw();
  pShape->calcArea();
}
```

Example – V Table



Shape vTable

…

calcArea

0

draw

Line object

Shape …

point1 = p1

point2 = p2

Line vTable

calcArea

draw

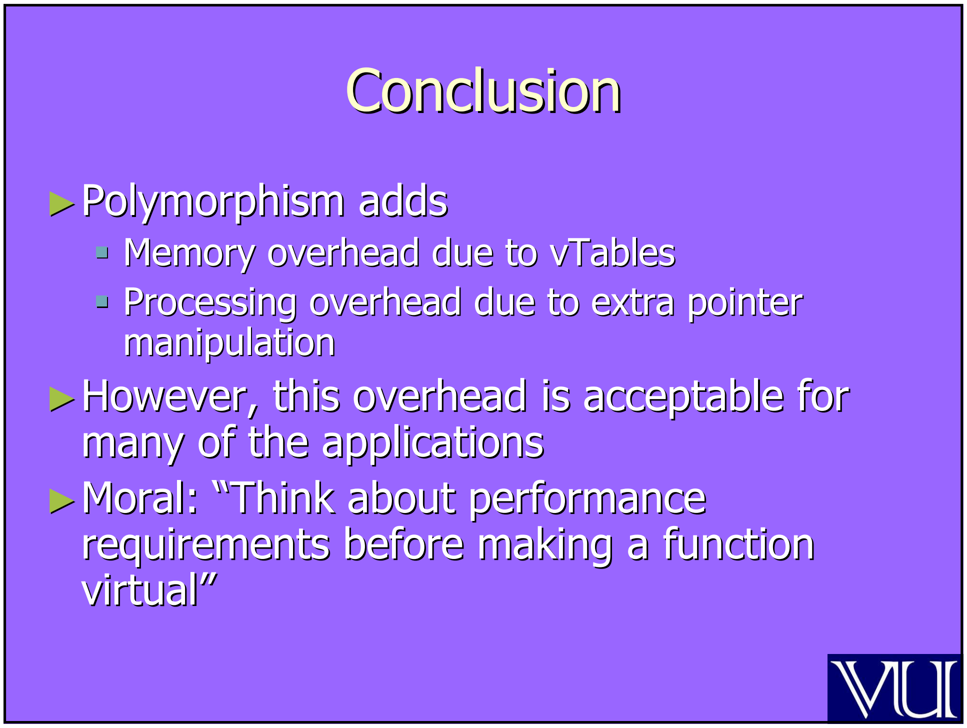
…

pShape

Dynamic Dispatch

- For non-virtual functions, compiler just generates code to call the function

- In case of virtual functions, compiler generates code to
  - access the object
  - access the associated vTable
  - call the appropriate function



Conclusion

- Polymorphism adds
  - Memory overhead due to vTables
  - Processing overhead due to extra pointer manipulation
- However, this overhead is acceptable for many of the applications
- Moral: “Think about performance requirements before making a function virtual”