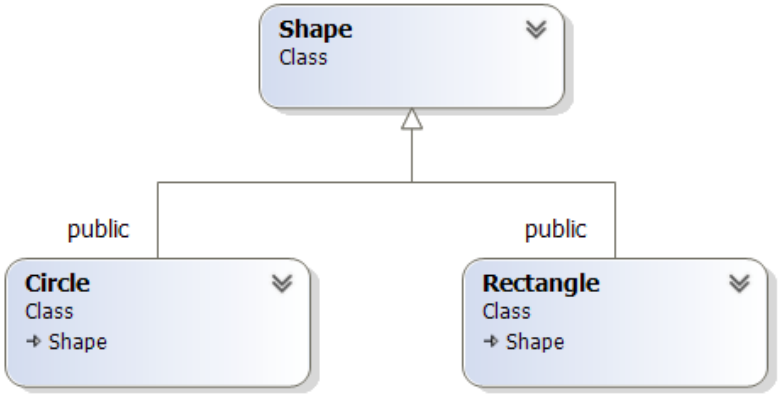
C++ Virtual and Polymorphism

- Virtual Functions
- Pure Virtual functions
- Abstract Base Classes
- Polymorphism

Virtual Functions

- Keyword virtual means find the ‘use most derived instance of’
- Example helps:
  - Have two objects; Rectangle,Circle that derive from Shape
  - Each have a whatami() function
    - Shape returns “Shape”
    - Rectangle returns “Rectangle”
    - Circle returns “Circle”



Virtual Functions

```
class Shape
{
public:
    Shape(void);
    virtual ~Shape(void);
    std::string whatami();
};
```

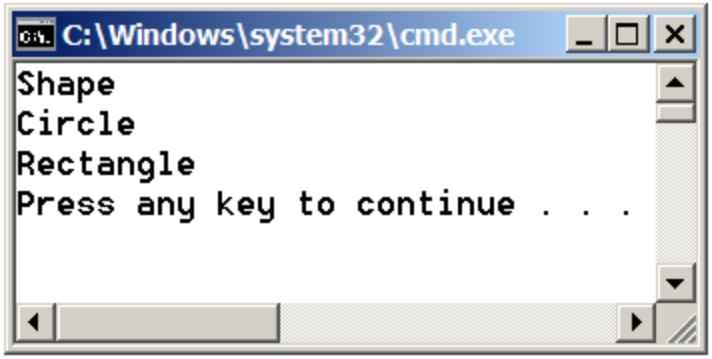
```
class Rectangle :
    public Shape
{
public:
    Rectangle(void);
    virtual ~Rectangle(void);
    std::string whatami();
};
```

```
class Circle :
    public Shape
{
public:
    Circle(void);
    virtual ~Circle(void);
    std::string whatami();
};
```

```
#include <iostream>
#include "Rectangle.h"
#include "Circle.h"

int main(){
    Shape       myShape;
    Circle      myCircle;
    Rectangle   myRectangle;

    std::cout<< myShape.whatami()<<std::endl;
    std::cout<< myCircle.whatami()<<std::endl;
    std::cout<< myRectangle.whatami()<<std::endl;
}
```



Virtual Functions- base class pointers

- Often use base class pointers to manipulate derived classes though…

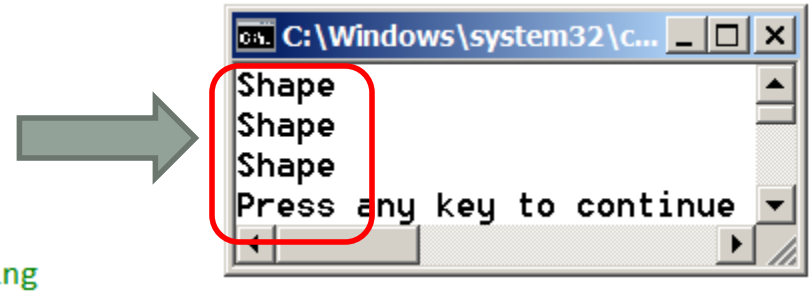
```
#include <iostream>
#include "Rectangle.h"
#include "Circle.h"
int main(){
    Shape       myShape;
    Circle      myCircle;
    Rectangle   myRectangle;

    //lets use a base class pointer, to ease printing
    Shape *pshape = 0;

    pshape = &myShape;
    std::cout<< pshape->whatami()<<std::endl;   //same line

    pshape = &myCircle;
    std::cout<< pshape->whatami()<<std::endl;   //same line

    pshape = &myShape;
    std::cout<< pshape->whatami()<<std::endl;   //same line
}
```



Slicing problem: using a base class pointer on a non virtual function, slices All derived content

Virtual Functions- base class pointers

- Fix for slicing problem is to make the functions virtual
- Adds a little overhead but at runtime you call most derived instance of whatami()

```
class Shape
{
public:
    Shape(void);
    virtual ~Shape(void);
    virtual std::string whatami();
};
```

```
class Rectangle :
    public Shape
{
public:
    Rectangle(void);
    virtual ~Rectangle(void);
    virtual std::string whatami();
};
```

```
class Circle :
    public Shape
{
public:
    Circle(void);
    virtual ~Circle(void);
    virtual std::string whatami();
};
```

Virtual Functions

- Now run it with newly virtualized functions

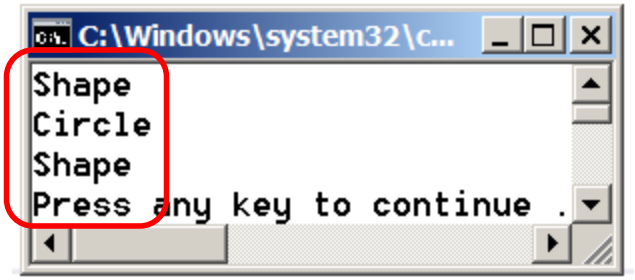
```
#include <iostream>
#include "Rectangle.h"
#include "Circle.h"
int main(){
    Shape       myShape;
    Circle      myCircle;
    Rectangle   myRectangle;

    //lets use a base class pointer, to ease printing
    Shape *pshape = 0;

    pshape = &myShape;
    std::cout<< pshape->whatami()<<std::endl;   //same line

    pshape = &myCircle;
    std::cout<< pshape->whatami()<<std::endl;   //same line

    pshape = &myShape;
    std::cout<< pshape->whatami()<<std::endl;   //same line
}
```



Slicing problem gone
All are correct

What should be virtual

- Only comes into play when using inheritance
- Any function that overrides a parent class function
- The destructor should always be virtual

Destructors in Derived Classes

- Virtual or not compiler calls them for you
  - Base *pBase = new Derived;
  - delete pBase; //calls~Derived() then ~Base()
- If destructor virtual and using base pointer compiler starts with most derived and works back to base.
- If destructor not virtual and using base pointer compiler calls Base destructor (slices of Derived portion of the object).

Sidenote: calling base class functions from derived classes

- You can explicitly call a base class implementation of a function you override
- Just give it the proper scoping
- Below the scope is ‘Shape’

```
class Shape
{
public:
    Shape(void);
     ~Shape(void);
    virtual std::string whatami();
    virtual void afunction();
};
```

```
void Rectangle::afunction(){
    Shape::afunction();
}
```

Pure Virtual Functions

- Sometimes virtual isn't quite right
- What if you had a method area() that returned the area of the object?
  - Rectangle::area() { return height*width};
  - Circle::area(){return pi*r**2}
- What should Shape's area function return?
- Answer: You should not be able to call it because it cannot be defined

Pure Virtual Functions

- Make the method Abstract (=0 in .h file)

```
class Shape
{
public:
    Shape(void);
     ~Shape(void);
    virtual std::string whatami();
    virtual void area()=0;
};
```

- Makes it an Abstract Base Class (ABC)

Abstract Base Class (ABC)

```
#pragma once
class Liquid
{
public:
    //this pure virtual function makes this
    //Base Class an abstract class
    //you cannot instantiate it in any way
    virtual void whoAmI()=0;
    Liquid(void);
    virtual ~Liquid(void);
};
```

- In header at least 1 virtual function =0 (pure virtual)
- You cannot invoke an abstract method
- Cannot instantiate ABC
- Derived classes MUST implement the pure virtual function
- See AbstractBaseClass project

ABC – What good is it?

- Defines required behavior that child classes must implement
- Used for manipulation of derived classes from base class pointers (polymorphism)
- Go to ABC_and_Virtual project

Summary

- Inheritance and Composition
- Hide Data and minimal public interface
- Virtual functions – ensure most derived version of function called
- Virtual functions – allow list of base class pointers that point to variety of derived objects
- Abstract base classes – force implementation of virtual functions
- Some OO design practice (Forrest Life ABC Demo)