Introduction to C++: Part 3

BOSTON UNIVERSITY

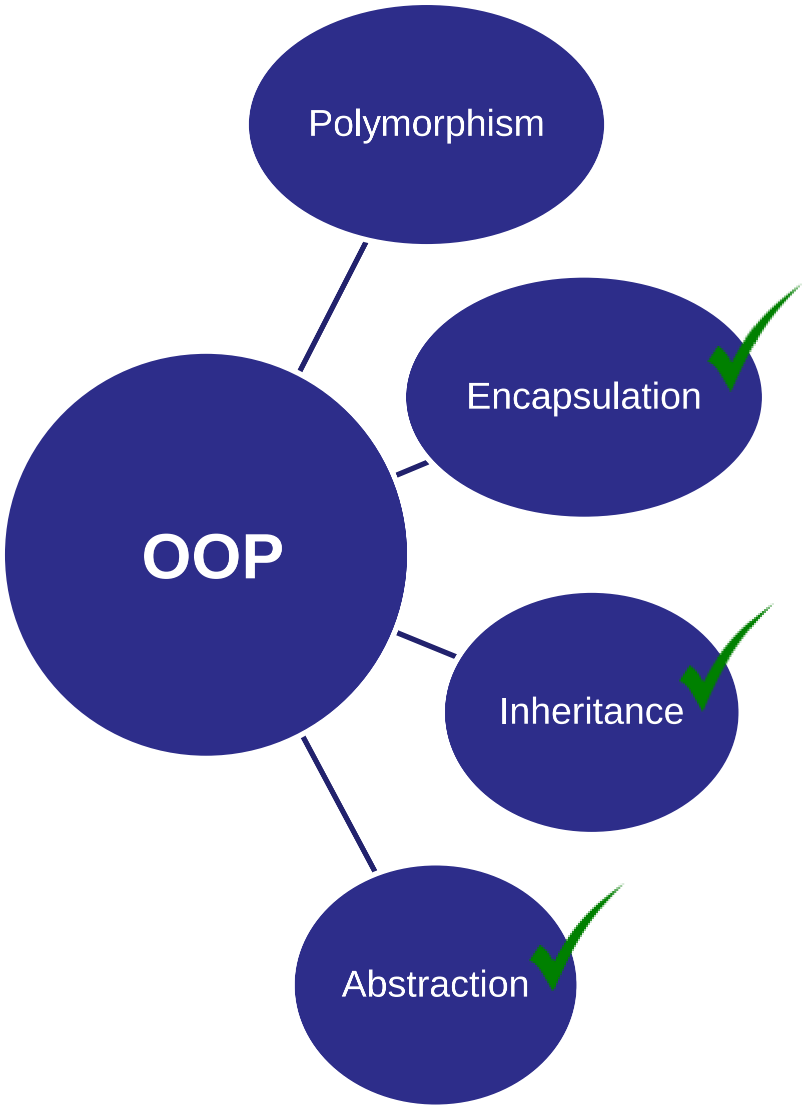
Tutorial Outline: Part 3

- Inheritance and overrides
- Virtual functions and interfaces

BOSTON UNIVERSITY

The formal concepts in OOP

- Next up: Polymorphism



BOSTON UNIVERSITY

Using subclasses

- A function that takes a superclass argument can *also* be called with a subclass as the argument.
- The reverse is **not** true – a function expecting a subclass argument cannot accept its superclass.
- Copy the code to the right and add it to your main.cpp file.

```
void PrintArea(Rectangle &rT) {
        cout << rT.Area() << endl ;
}

int main() {
        Rectangle rT(1.0,2.0) ;
        Square sQ(3.0) ;
        PrintArea(rT) ;
        PrintArea(sQ) ;
}
```

The PrintArea function can accept the Square object *sQ* because Square is a subclass of Rectangle.

BOSTON UNIVERSITY

Overriding Methods

- Sometimes a subclass needs to have the same interface to a method as a superclass with different functionality.
- This is achieved by *overriding* a method.
- Overriding a method is simple: just re-implement the method with the same name and arguments in the subclass.

In C::B open project:
CodeBlocks Projects → Part 2 → Virtual Method Calls

```
class Super {
public:
    void PrintNum() {
        cout << 1 << endl ;
    }
} ;

class Sub : public Super {
public:
    // Override
    void PrintNum() {
        cout << 2 << endl ;
    }
} ;
Super sP ;
sP.PrintNum() ; // Prints 1
Sub sB ;
sB.PrintNum() ; // Prints 2
```

BOSTON UNIVERSITY

Overriding Methods

- Seems simple, right?

```
class Super {
public:
    void PrintNum() {
        cout << 1 << endl ;
    }
} ;

class Sub : public Super {
public:
    // Override
    void PrintNum() {
        cout << 2 << endl ;
    }
} ;
Super sP ;
sP.PrintNum() ; // Prints 1
Sub sB ;
sB.PrintNum() ; // Prints 2
```

How about in a function call…

- Using a single function to operate on different types is *polymorphism.*

- Given the class definitions, what is happening in this function call?

“C++ is an insult to the human brain”
– Niklaus Wirth (designer of Pascal)

```
class Super {
public:
    void PrintNum() {
        cout << 1 << endl ;
    }
} ;

class Sub : public Super {
public:
    // Override
    void PrintNum() {
        cout << 2 << endl ;
    }
} ;
```

```
void FuncRef(Super &sP) {
        sP.PrintNum() ;
}

Super sP ;
Func(sP) ;  // Prints 1
Sub sB ;
Func(sB) ;  // Hey!! Prints 1!!
```

Type casting

```
void FuncRef(Super &sP) {
        sP.PrintNum() ;
}
```

- The Func function passes the argument as a *reference* (Super &sP).
  - What’s happening here is *dynamic type casting*, the process of converting from one type to another at runtime.
  - Same mechanism as the *dynamic_cast<type>()* function

- The incoming object is treated as though it were a superclass object in the function.

- When methods are overridden and called there are two points where the proper version of the method can be identified: either at compile time or at runtime.

Virtual methods

- When a method is labeled as virtual and overridden the compiler will generate code that will check the type of an object at **runtime** when the method is called.

- The type check will then result in the expected version of the method being called.

- When overriding a virtual method in a subclass, it’s a good idea to label the method as virtual in the subclass as well.
  - …just in case this gets subclassed again!

```
class SuperVirtual
{
public:
    virtual void PrintNum()
    {
        cout << 1 << endl ;
    }
} ;

class SubVirtual : public SuperVirtual
{
public:
    // Override
    virtual void PrintNum()
    {
        cout << 2 << endl ;
    }
} ;

void Func(SuperVirtual &sP)
{
    sP.PrintNum() ;
}

SuperVirtual sP ;
Func(sP) ;  // Prints 1
SubVirtual sB ;
Func(sB) ;  // Prints 2!!
```

BOSTON UNIVERSITY

Early (static) vs. Late (dynamic) binding

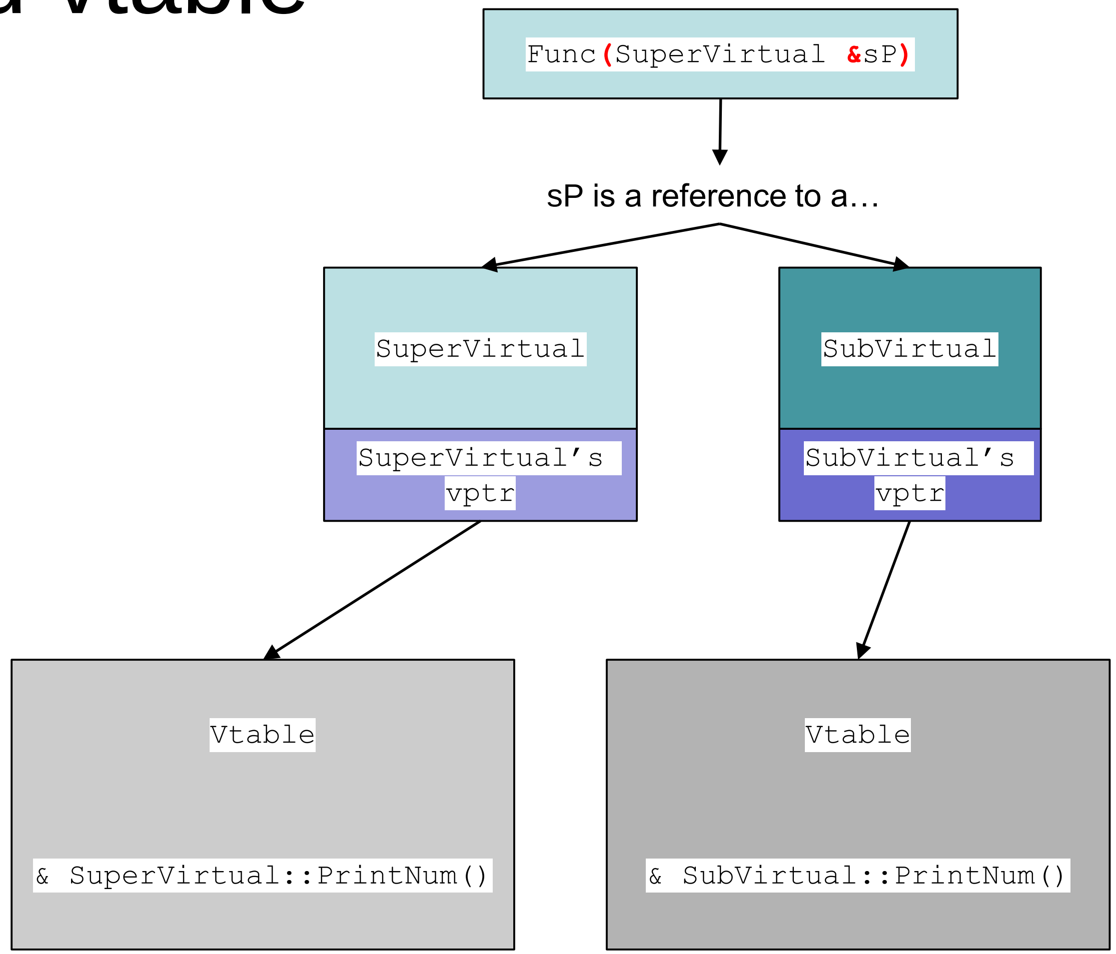
- Leaving out the virtual keyword on a method that is overridden results in the compiler deciding *at compile time* which version (subclass or superclass) of the method to call.
- This is called early or static *binding*.
- At compile time, a function that takes a superclass argument will only call the **non-virtual** superclass method under early binding.

- Making a method virtual adds code behind the scenes (that you, the programmer, never interact with directly)
  - Lookups in a hidden table, called the *vtable*, are done to figure out what version of the virtual method should be run.
- This is called late or dynamic binding.
- There is a small performance penalty for late binding due to the vtable lookup.
- **This only applies when an object is referred to by a reference or pointer.**

BOSTON UNIVERSITY

Behind the scenes – vptr and vtable

- C++ classes have a hidden pointer (vptr) generated that points to a table of virtual methods associated with a class (vtable).
- When a virtual class method (base class or its subclasses) is called by reference (or pointer) *when the program is running* the following happens:
  - The object's **class** vptr is followed to its **class** vtable
  - The virtual method is looked up in the vtable and is then called.
  - One vptr and one vtable per class so minimal memory overhead
  - If a method override is **non**-virtual it won't be in the vtable and it is selected at **compile time**.



BOSTON UNIVERSITY

Let’s run this through the debugger

- Open the project: Parts 2-3/Virtual Method Calls.
- Everything here is implemented in one big main.cpp
- Place a breakpoint at the first line in main() and in the two implementations of Func()
- Make sure the “Watches” debugging window is open.

When to make methods virtual

- If a method will be (or might be) overridden in a subclass, make it virtual
  - There is a *minor* performance penalty. Will that even matter to you?
    - i.e. Have you profiled and tested your code to show that virtual method calls are a performance issue?
  - When is this true?
    - Almost always! Who knows how your code will be used in the future?

- Constructors are **never** virtual in C++.
- Destructors in a base class should always be virtual.
  - Also – if any method in a class is virtual, make the destructor virtual
  - These are important when dealing with objects via reference and it avoids some subtleties when manually allocating memory.

BOSTON UNIVERSITY

Why all this complexity?

```
void FuncEarly(SuperVirtual &sP)
{
    sP.PrintNum() ;
}
```

- Called by **reference** – late binding to PrintNum()

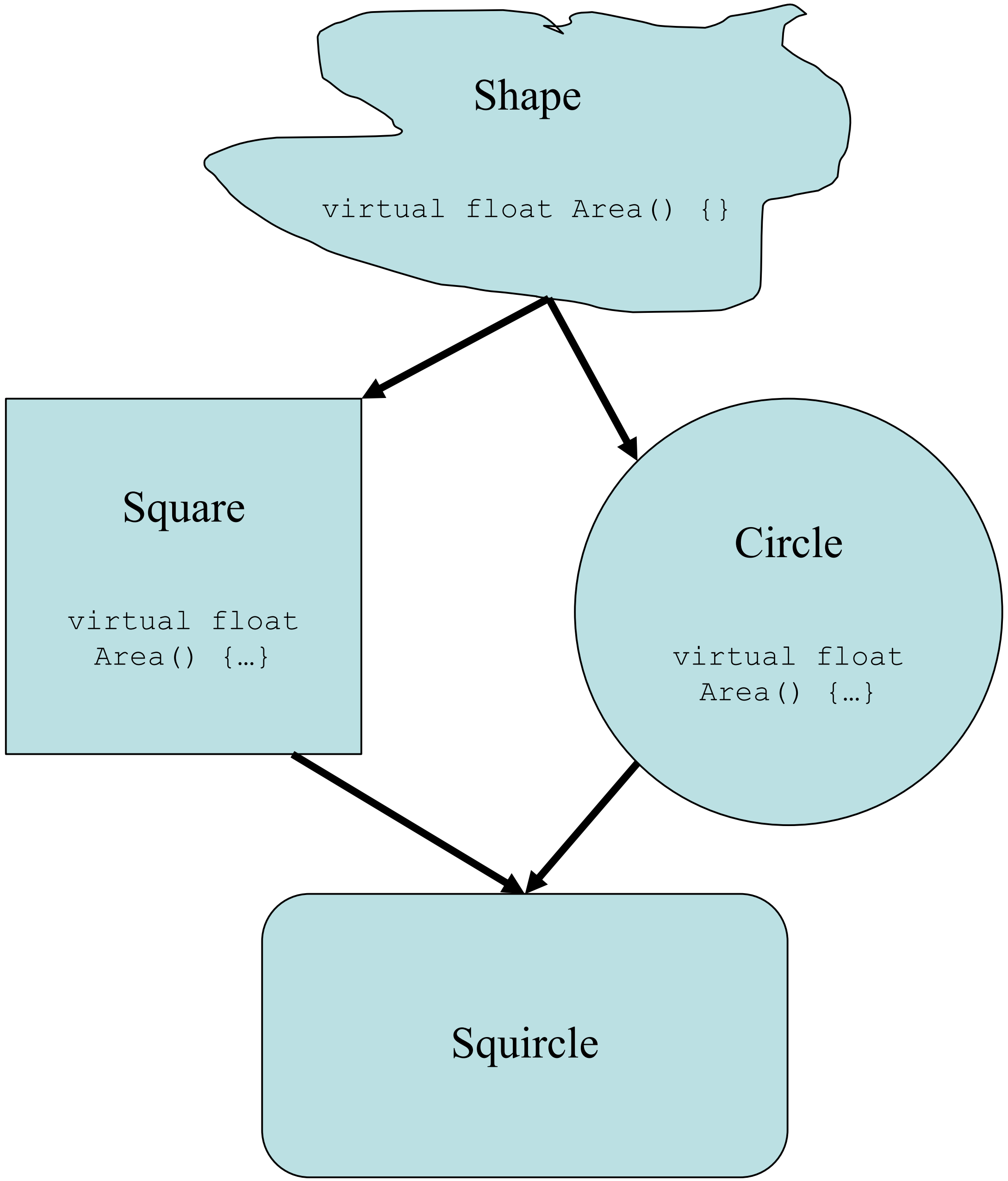
```
void FuncLate(SuperVirtual sP)
{
    sP.PrintNum() ;
}
```

- Called by **value** – early binding to PrintNum even though it's virtual!

- Late binding allows for code libraries to be updated for new functionality. As methods are identified at runtime the executable does not need to be updated.
- This is done all the time! Your C++ code may be, for example, a plugin to an existing simulation code.
- Greater flexibility when dealing with multiple subclasses of a superclass.
- Most of the time this is the behavior you are looking for when building class hierarchies.

- Remember the Deadly Diamond of Death? Let's explain.
- Look at the class hierarchy on the right.
  - Square and Circle inherit from Shape
  - Squircle inherits from both Square and Circle
  - Syntax:

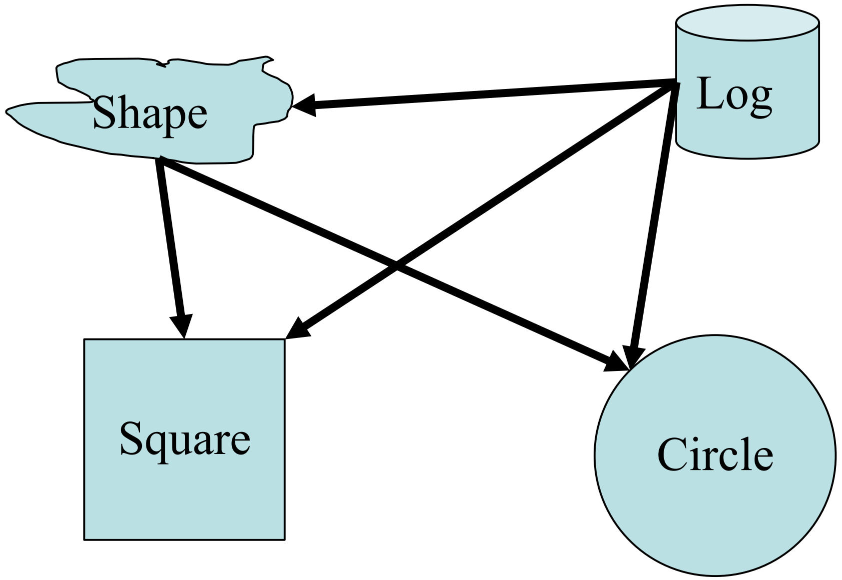
    class Squircle : public Square, public Circle
- The Shape class implements an empty Area() method. The Square and Circle classes override it. Squircle does not.
- Under late binding, which version of Area is accessed from Squircle? Square.Area() or Circle.Area()?



BOSTON UNIVERSITY

Interfaces

- Another pitfall of multiple inheritance: the *fragile base class* problem.
  - If many classes inherit from a single base (super) class then changes to methods in the base class can have unexpected consequences in the program.
  - This can happen with single inheritance but it's much easier to run into with multiple inheritance.
- Interfaces are a way to have your classes share behavior without them sharing actual code.
- Gives much of the benefit of multiple inheritance without the complexity and pitfalls



- Example: for debugging you'd like each class to have a Log() method that would write some info to a file.
  - Implement with an interface.

Interfaces

- An interface class in C++ is called a pure virtual class.
- It contains virtual methods only with a special syntax. Instead of {} the function is set to 0.
  - Any subclass needs to implement the methods!
- Modified square.h shown.
- What happens when this is compiled?

```
(…error…)
include/square.h:10:7: note:   because the following virtual
functions are pure within 'Square':
 class Square : public Rectangle, Log
       ^
include/square.h:7:18: note:  virtual void Log::LogInfo()
     virtual void LogInfo()=0 ;
```

- Once the LogInfo() is uncommented it will compile.

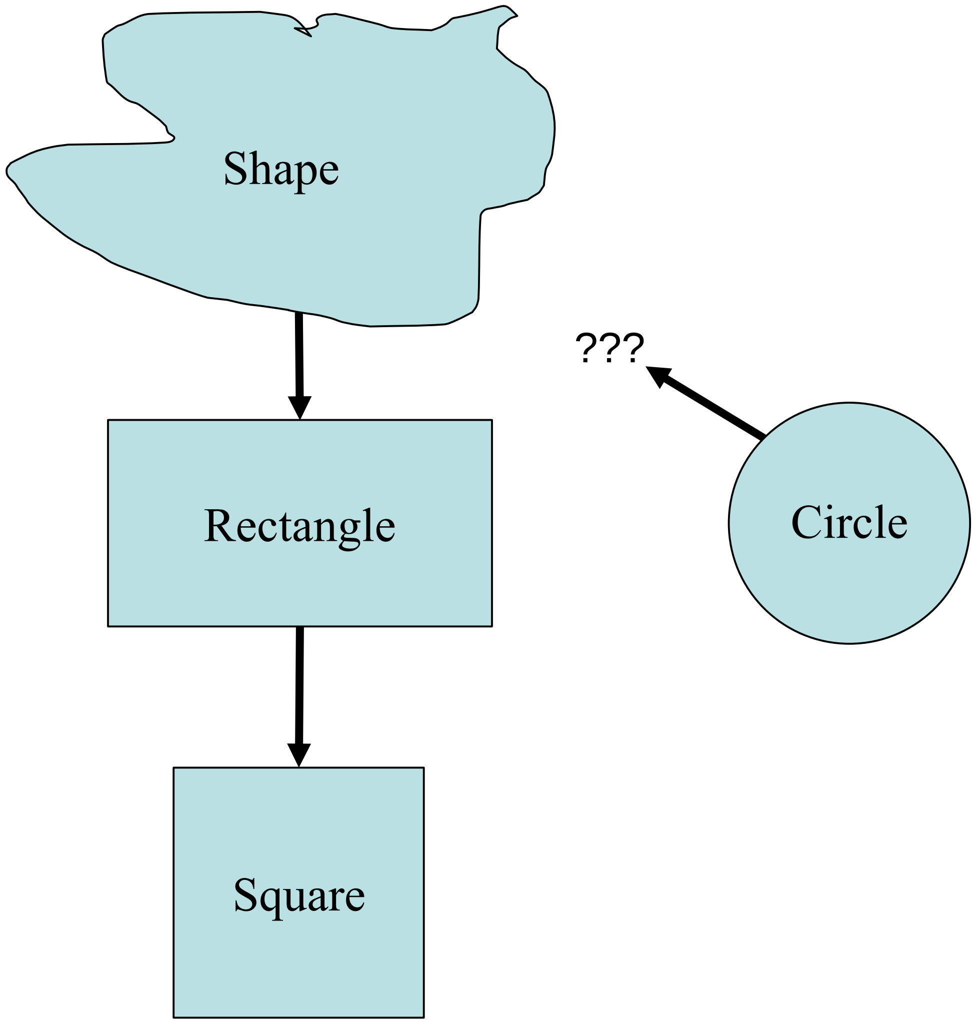
```
#ifndef SQUARE_H
#define SQUARE_H

#include "rectangle.h"

class Log {
    virtual void LogInfo()=0 ;
};

class Square : public Rectangle, Log
{
    public:
        Square(float length);
        virtual ~Square();
        // virtual void LogInfo() {}
protected:

    private:
};

#endif // SQUARE_H
```

BOSTON UNIVERSITY

- C++ offers another fix for the diamond problem, Virtual inheritance. See: https://en.wikipedia.org/wiki/Virtual_inheritance

Putting it all together

- Now let's revisit our Shapes project.
- In the directory of C::B Part 2-3 projects, open the **"Shapes with Circle"** project.
  - This has a Shape base class with a Rectangle and a Square
- Add a Circle class to the class hierarchy in a sensible fashion.



- Hint: Think first, code second.

New pure virtual Shape class

- Slight bit of trickery:
  - An empty constructor is defined in shape.h
  - No need to have an extra shape.cpp file if these functions do nothing!

- Q: How much code can be in the header file?
- A: Most of it with some exceptions.
  - .h files are not compiled into .o files so a header with a lot of code gets re-compiled every time it's referenced in a source file.

```
#ifndef SHAPE_H
#define SHAPE_H


class Shape
{
    public:
        Shape()  {}
        virtual ~Shape()  {}

        virtual float Area()=0 ;
    protected:

    private:
};

#endif // SHAPE_H
```

BOSTON UNIVERSITY

Give it a try

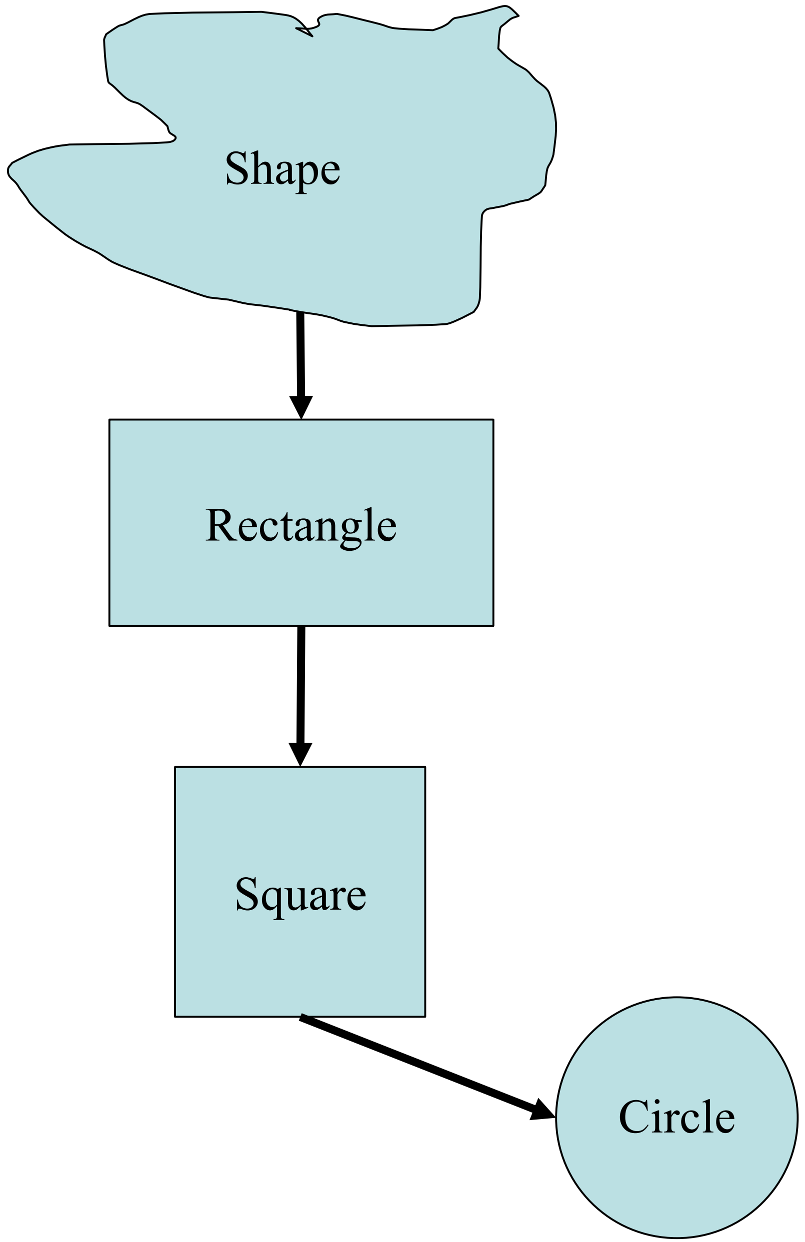
- Add inheritance from Shape to the Rectangle class
- Add a Circle class, inheriting from wherever you like.
- Implement Area() for the Circle

- If you just want to see a solution, open the project “Shapes with Circle solved”

BOSTON UNIVERSITY

A Potential Solution

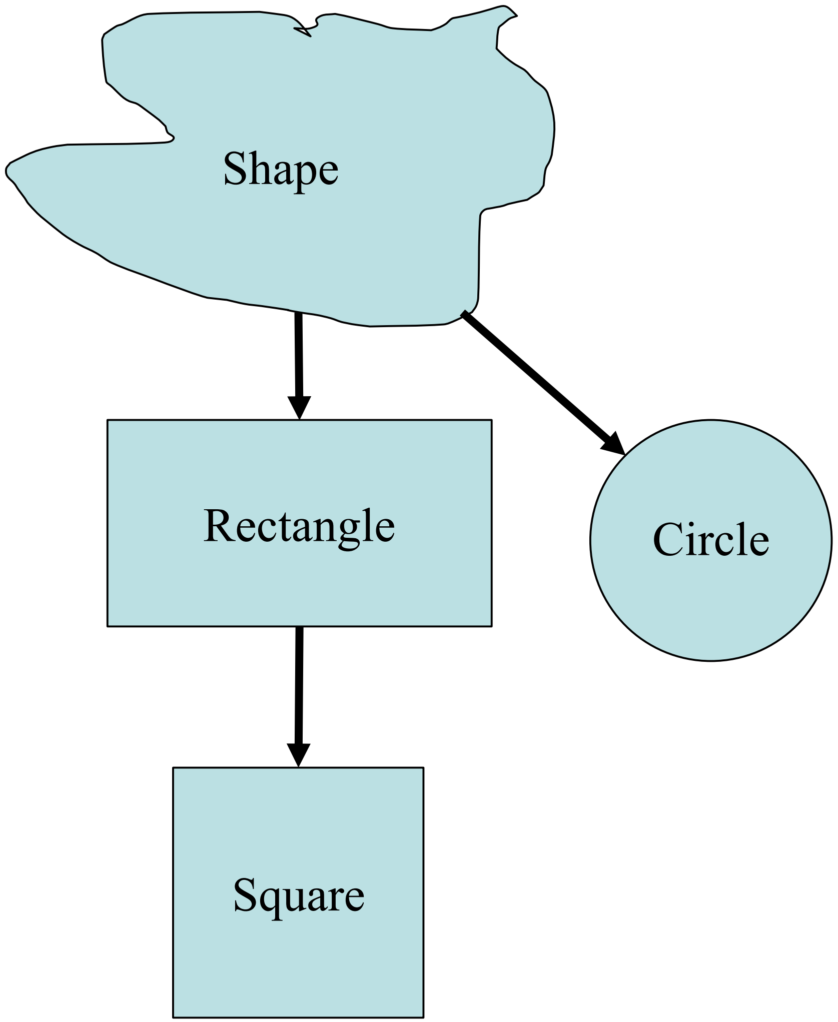
- A Circle has one dimension (radius), like a Square.
  - Would only need to override the Area() method
- But…
  - Would be storing the radius in the members m_width and m_length. This is not a very obvious to someone else who reads your code.
- Maybe:
  - Change m_width and m_length names to m_dim_1 and m_dim_2?
    - Just makes everything more muddled!



BOSTON UNIVERSITY

A Better Solution

- Inherit separately from the Shape base class
  - Seems logical, to most people a circle is not a specialized form of rectangle…
- Add a member m_radius to store the radius.
- Implement the Area() method
- Makes more sense!
- Easy to extend to add an Oval class, etc.



New Circle class

- Also inherits from Shape
- Adds a constant value for π
  - Constant values can be defined right in the header file.
  - If you accidentally try to change the value of PI the compiler will throw an error.

```
#ifndef CIRCLE_H
#define CIRCLE_H

#include "shape.h"


class Circle : public Shape
{
    public:
        Circle();
        Circle(float radius) ;
        virtual ~Circle();

        virtual float Area() ;

        const float PI = 3.14;
        float m_radius ;

    protected:

    private:
};

#endif // CIRCLE_H
```

BOSTON UNIVERSITY

- circle.cpp
- Questions?

```
#include "circle.h"

Circle::Circle()
{
    //ctor
}

Circle::~Circle()
{
    //dtor
}

// Use a member initialization list.
Circle::Circle(float radius) : m_radius{radius}
{}

float Circle::Area()
{
    // Quiz: what happens if this line is
    // uncommented and then compiled:
    //PI=3.14159 ;
    return m_radius * m_radius * PI ;
}
```

BOSTON UNIVERSITY

Quiz time!

- What happens behind the scenes when the function PrintArea is called?
- How about if PrintArea's argument was instead:

```
void PrintArea(Shape shape)
```

```
void PrintArea(Shape &shape) {
    cout <<  "Area: " << shape.Area() << endl ;
}

int main()
{
    Square sQ(4) ;
    Circle circ(3.5) ;
    Rectangle rT(21,2) ;

    // Print everything
    PrintArea(sQ) ;
    PrintArea(rT) ;
    PrintArea(circ) ;
    return 0;
}
```

BOSTON UNIVERSITY

Quick mention…

- Aside from overriding functions it is also possible to override operators in C++.
  - As seen in the C++ string. The + operator concatenates strings:

```
string str = "ABC" ;
str = str + "DEF" ;
//  str is now "ABCDEF"
```

- It's possible to override +,-,=,<,>, brackets, parentheses, etc.

- Syntax:

```
MyClass operator*(const MyClass& mC) {...}
```

- Recommendation:
  - Generally speaking, avoid this. This is an easy way to generate very confusing code.
  - A well-named function will almost always be easier to understand than an operator.
- An exceptions is the assignment operator: operator=

BOSTON UNIVERSITY

Summary

- C++ classes can be created in hierarchies via inheritance, a core concept in OOP.
- Classes that inherit from others can make use of the superclass' public and protected members and methods
  - You write less code!
- Virtual methods should be used whenever methods will be overridden in subclasses.
- Avoid multiple inheritance, use interfaces instead.
- Subclasses can override a superclass method for their own purposes and can still explicitly call the superclass method.
- Abstraction means hiding details when they don't need to be accessed by external code.
  - Reduces the chances for bugs.
- While there is a lot of complexity here – in terms of concepts, syntax, and application – keep in mind that OOP is a highly successful way of building programs!

BOSTON UNIVERSITY