Introduction to C++: Part 3

BOSTON UNIVERSITY

Tutorial Outline: Part 3

- Intro to the Standard Template Library
- Class inheritance
- Public, private, and protected access
- Virtual functions

The Standard Template Library

- The STL is a large collection of containers and algorithms that are part of C++.
  - It provides many of the basic algorithms and data structures used in computer science.

- As the name implies, it consists of generic code that you specialize as needed.

- The STL is:
  - Well-vetted and tested.
  - Well-documented with lots of resources available for help.

Containers

- There are 16 types of containers in the STL:

Container	Description
array	1D list of elements.
vector	1D list of elements
deque	Double ended queue
forward_list	Linked list
list	Double-linked list
stack	Last-in, first-out list.
queue	First-in, first-out list.
priority_queue	1st element is always the largest in the container

Container	Description
set	Unique collection in a specific order
multiset	Elements stored in a specific order, can have duplicates.
map	Key-value storage in a specific order
multimap	Like a map but values can have the same key.
unordered_set	Same as set, sans ordering
unordered_multiset	Same as multisetset, sans ordering
unordered_map	Same as map, sans ordering
unordered_multimap	Same as multimap, sans ordering

BOSTON UNIVERSITY

Algorithms

- There are 85+ of these.
  - Example: find, count, replace, sort, is_sorted, max, min, binary_search, reverse
- Algorithms manipulate the data stored in containers but is not tied to STL containers
  - These can be applied to your own collections or containers of data
- Example:

```
vector<int> v(3);                 // Declare a vector of 3 elements.
v[0] = 7;
v[1] = 3;
v[2] = v[0] + v[1];               // v[0] == 7, v[1] == 3, v[2] == 10
reverse(v.begin(), v.end()) ;  // v[0] == 10, v[1] == 3, v[2] == 7
```

- The implementation is hidden and the necessary code for reverse() is generated from templates at compile time.

vector<T>

- A very common and useful class in C++ is the vector class. Access it with:

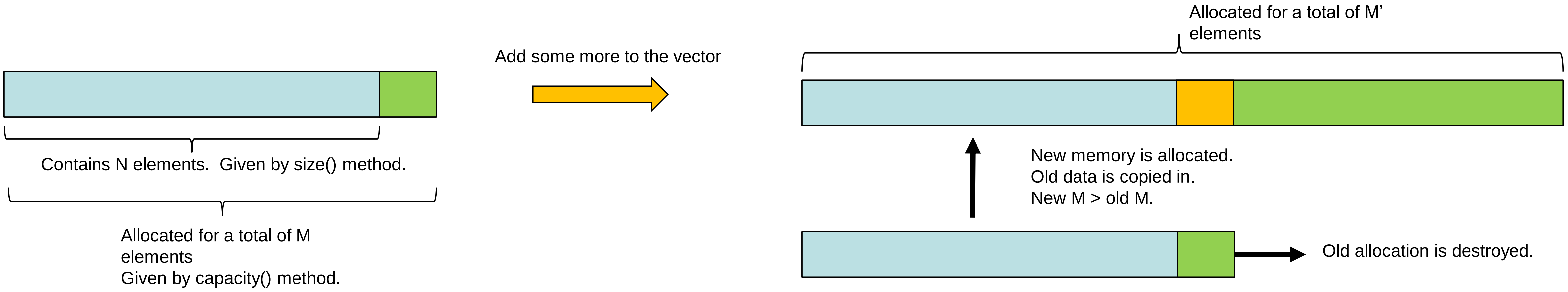
```
#include <vector>
```

- Vector has many methods:
  - Various constructors
  - Ways to iterate or loop through its contents
  - Copy or assign to another vector
  - Query vector for the number of elements it contains or its backing storage size.
- Example usage: `vector<float> my_vec ;`
- Or: `vector<float> my_vec(50) ;`

vector<T>

- Hidden from the programmer is the *backing store*
- Object oriented design in action!

- This is how the vector stores its data internally.



Destructors

- vector<t> can hold objects of any type:
  - Primitive (aka basic) types: int, float, char, etc.
  - Objects: string, your own classes, file stream objects (ex. ostream), etc.
  - Pointers: int*, string*, etc.
  - But NOT references!
- When a vector is destroyed:
  - If it holds primitive types or pointers it just deallocates its backing store.
  - If it holds objects it will call each object's destructor before freeing its backing store.

vector<t> with objects

- Select an object in a vector.
- The members and methods can be accessed directly.

- Elements can be accessed with brackets and an integer starting from 0.

```
// a vector with memory preallocated to
// hold 1000 objects.
vector<MyClass> my_vec(1000);

// Now make a vector with 1000 MyClass objects
// that are initialized using the MyClass constructor
vector<MyClass> my_vec2(1000,MyClass(arg1,arg2));

// Access an object's method.
my_vec2[100].some_method() ;
// Or a member
my_vec2[10].member_integer = 100 ;


// Clear out the entire vector
my_vec2.clear()
// but that might not re-set the backing store…
// Let's check the docs:
// http://www.cplusplus.com/reference/vector/vector/clear/
```

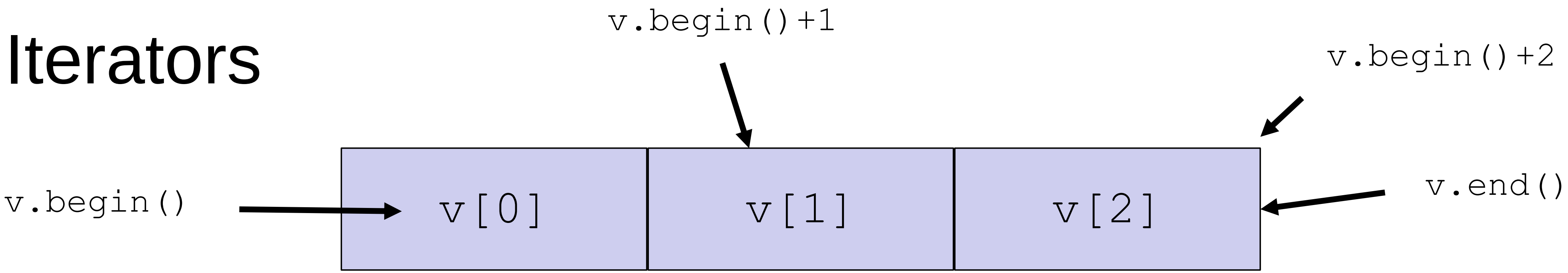
BOSTON UNIVERSITY

Looping

```
for (int index = 0 ; index < vec.size() ; ++index)
{
        // ++index means "add 1 to the value of index"
        cout << vec[index] << " " ;
}
```

- Loop with a “for” loop, referencing the value of vec using brackets.
- 1st time through:
  - index = 0
  - Print value at vec[0]
  - index gets incremented by 1
- 2nd time through:
  - Index = 1
  - Etc
- After last time through
  - Index now equal to vec.size()
  - Loop exits
- Careful! Using an out of range index will likely cause a memory error that crashes your program.

Iterators



- Iterators are generalized ways of keeping track of positions in a container.
- 3 types: forward iterators, bidirectional, random access
- Forward iterators can only be incremented (as seen here)
- Bidirectional can be added or subtracted to move both directions
- Random access can be used to access the container at any location

BOSTON UNIVERSITY

Looping

```
for (vector<int>::iterator itr = vec.begin(); itr != vec.end() ; ++itr)
{
      cout << *itr << " " ;
      // iterators are pointers!
}
```

- Loop with a "for" loop, referencing the value of vec using an **iterator** type.
- `vector<int>::iterator` is a type that iterates through a vector of int's.
- 1st time through:
  - itr points at 1st element in vec
  - Print value pointed at by itr: *itr
  - itr is incremented to the next element in the vector
- Iterators are very useful C++ concepts. They work on any STL container!
  - **No need to worry about the # of elements!**
  - Exact iterator behavior depends on the type of container but they are guaranteed to always reach every value.

BOSTON UNIVERSITY

Looping

```
for (auto itr = vec.begin() ; itr != vec.end() ; ++itr)
{
        cout << *itr << " " ;
}
```

- Let the *auto* type asks the C++ compiler to figure out the iterator type automatically.

```
for (auto itr = vec.begin(), auto vec_end = vec.end() ; itr != vec_end ; ++itr)
{
        cout << *itr << " " ;
}
```

- An extra modification: Assigning the vec_end variable avoids calling vec.end() on every loop.

BOSTON UNIVERSITY

Looping

```
for(const auto &element : vec)
{
        cout << element << " " ;
}
```

- Another iterator-based loop: iterator behavior and accessing an element are handled automatically by the compiler
- Uses a reference so the element is not copied.
- The ***const auto &*** prevents changes to the element in the vector.
- If you don't use *const* then the loop can update the vector elements via the reference.

- Less typing == less chance for program bugs.

Iterator notes

- There is small performance penalty for using iterators…but are they safer to use.
- They allow substitution of one container for another (list<> for vector<>, etc.)
- With templates you can write a function that accepts any STL container type.

```
template<typename T>
void dump_string(T &t)
{
    for( auto itr=t.begin() ; itr!=t.end() ; itr++) {
        cout << *itr << endl;
    }
}
```

```
list<float> lst ;
lst.push_back(-5.0) ;
lst.push_back(12.0) ;
vector<double> vec(2) ;
vec[0] = 1.0 ;
vec[1] = 2.0 ;

dump_string<list<float> >(lst) ;
dump_string<vector<double> >(lst) ;
```

BOSTON UNIVERSITY

STL Demo

- Open project *STL_Demo*

- Let’s walk through the functions with the debugger and see some vectors in action.

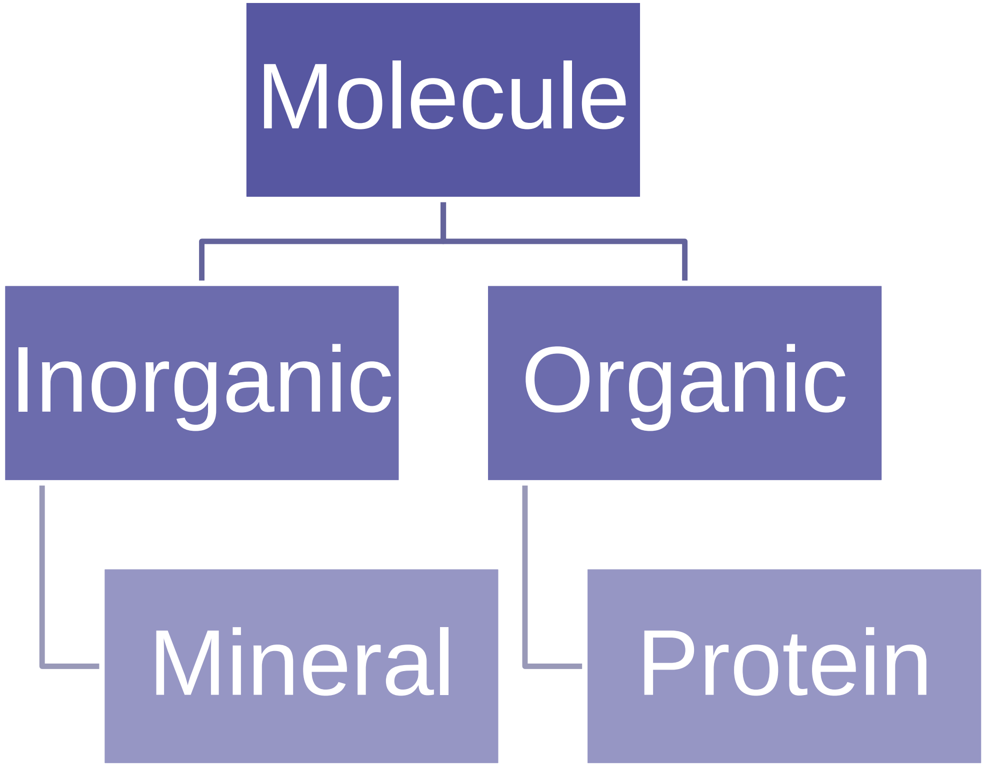
Tutorial Outline: Part 3

- Intro to the Standard Template Library
- Class inheritance
- Public, private, and protected access
- Virtual functions

Inheritance

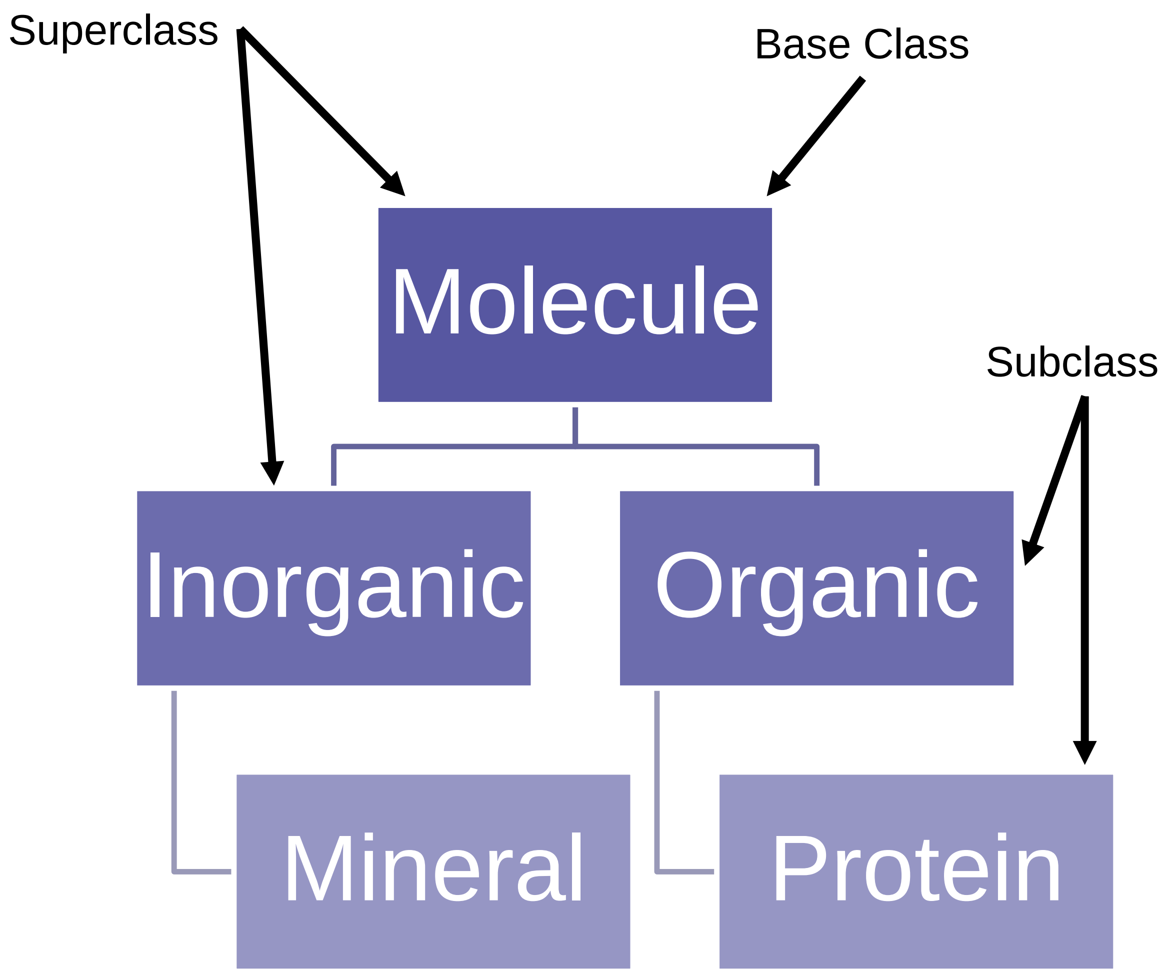
- Inheritance is the ability to form a hierarchy of classes where they share common members and methods.
  - Helps with: code re-use, consistent programming, program organization

- This is a powerful concept!



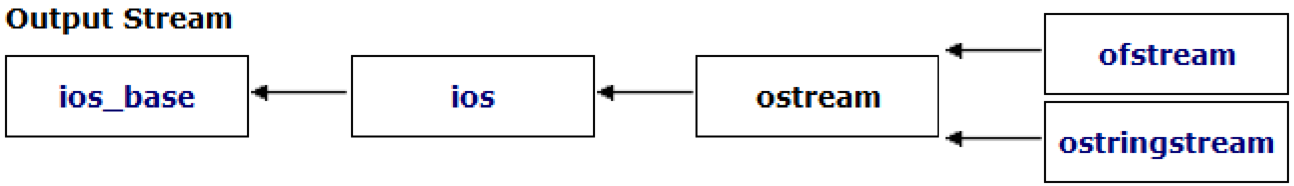
Inheritance

- The class being derived *from* is referred to as the **base**, **parent**, or **super** class.
- The class being derived is the **derived**, **child**, or **sub** class.
- For consistency, we'll use superclass and subclass in this tutorial. A base class is the one at the top of the hierarchy.



BOSTON UNIVERSITY

Inheritance in Action



- Streams in C++ are series of characters – the C+ I/O system is based on this concept.

- **cout** is an object of the class *ostream*. It is a write-only series of characters that prints to the terminal.

- There are two subclasses of ostream:
  - *ofstream* – write characters to a file
  - *ostringstream* – write characters to a string

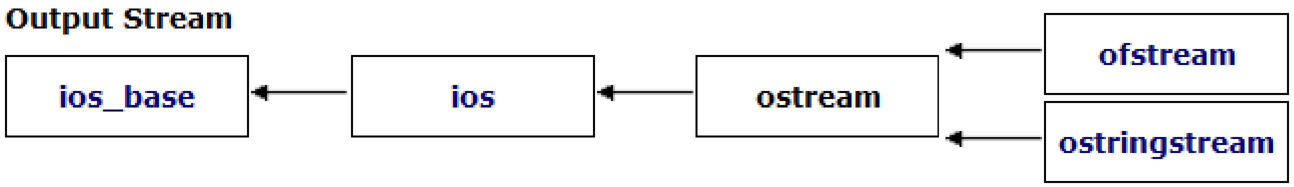
- Writing to the terminal is straightforward:

```
cout  << some_variable ;
```

- How might an object of class *ofstream* or *ostringstream* be used if we want to write characters to a file or to a string?

BOSTON UNIVERSITY

Inheritance in Action

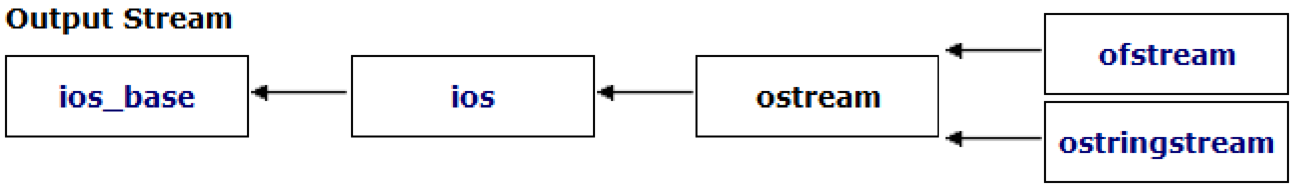


- For *ofstream* and *ofstringstream* the << operator is inherited from *ostream* and behaves the same way for each from the programmer's point of view.

- The *ofstream class* adds a constructor to open a file and a close() method.
- *ofstringstream* adds a method to retrieve the underlying string, str()

- If you wanted a class to write to something else, like a USB port…
  - Maybe look into inheriting from ostream!
    - Or *its* underlying class, *basic_ostream* which handles types other than characters…

Inheritance in Action




```
#include <iostream>  // cout
#include <fstream>  // ofstream
#include <sstream> // ostringstream

using namespace std ;
void some_func(string msg) {
        cout << msg ; // to the terminal
        // The constructor opens a file for writing
        ofstream my_file("filename.txt") ;
        // Write to the file.
        my_file << msg ;
        // close the file.
        my_file.close() ;
        ostringstream oss ;
        // Write to the stringstream
        oss << msg ;
        // Get the string from stringstream
        cout << oss.str()  ;
}
```

Public, protected, private

"There are only two things wrong with C++: The initial concept and the implementation."
– Bertrand Meyer (inventor of the Eiffel OOP language)

- Public and private were added by NetBeans to the Rectangle class.
- These are used to control access to parts of the class with inheritance.

```
class Rectangle
{
    public:
        Rectangle();
        Rectangle(float width, float length) ;
        virtual ~Rectangle();

        float m_width ;
        float m_length ;

        float Area() ;

    protected:

    private:
};
```

BOSTON UNIVERSITY

C++ Access Control and Inheritance

Access	public	protected	private
Same class	Yes	Yes	Yes
Subclass	Yes	Yes	No
Outside classes	Yes	No	No

```
class Super {
public:
    int i;
protected:
    int j ;
private:
    int k ;
};
```

Inheritance

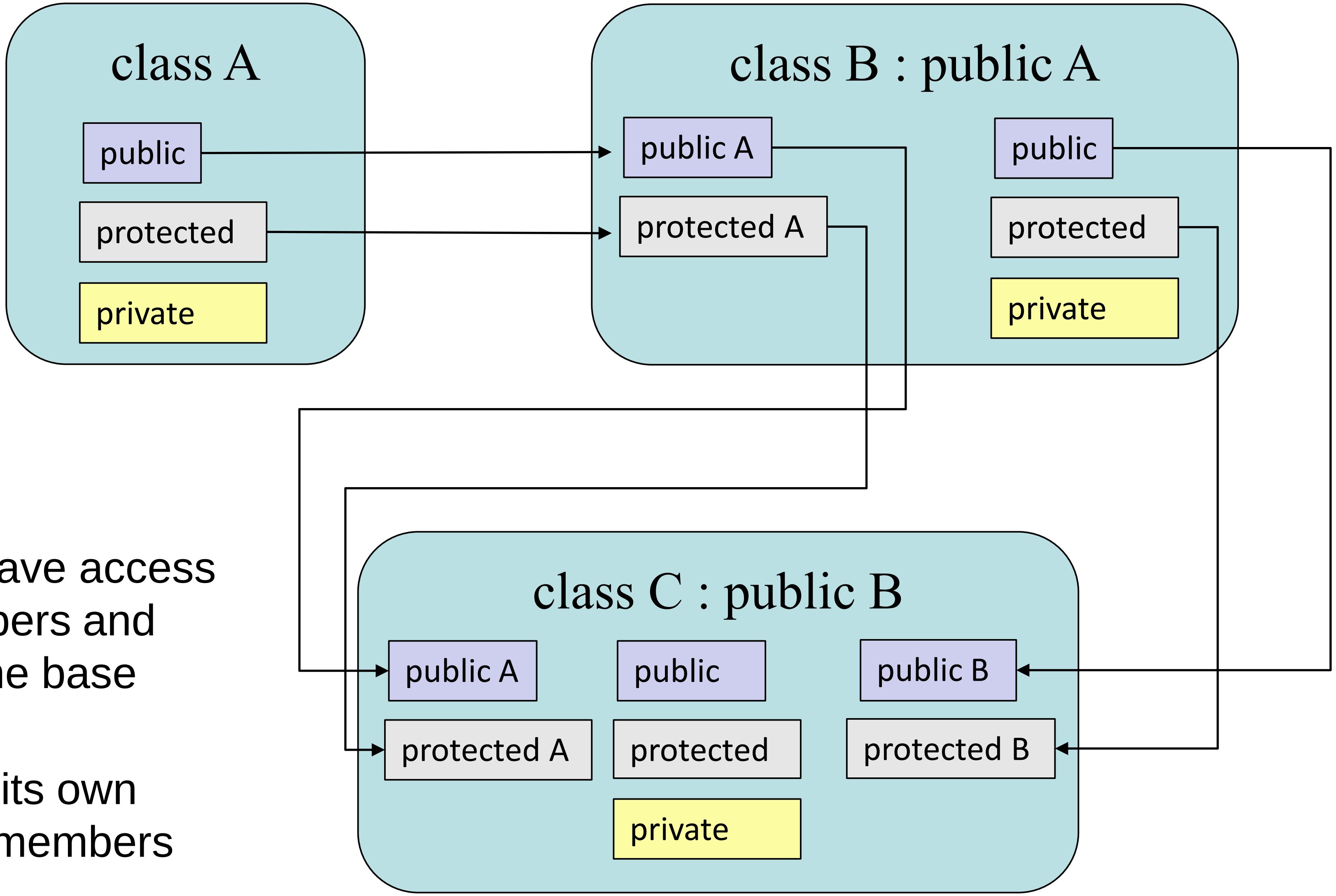
```
class Sub : public Super {
// in methods, could access
// i and j from Parent only.
};
```

Outside code

```
Sub myobj ;
Myobj.i = 10 ; // public - ok
Myobj.j = 3 ; // protected - Compiler error
Myobj.k = 1 ; // private - Compiler error
```

BOSTON UNIVERSITY

Inheritance

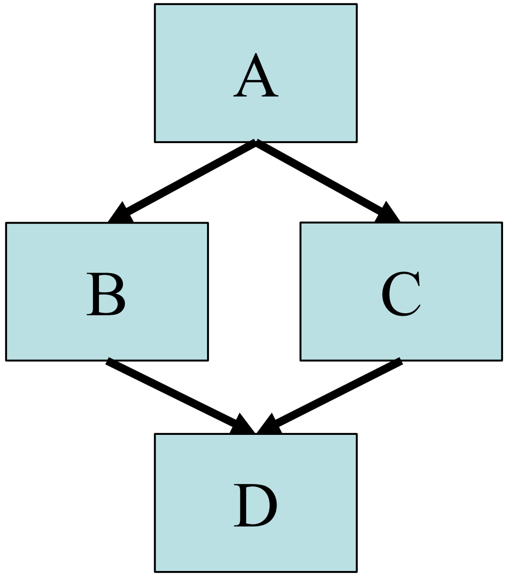


- With inheritance subclasses have access to private and protected members and methods all the way back to the base class.
- Each subclass can still define its own public, protected, and private members and methods along the way.

BOSTON UNIVERSITY

Single vs Multiple Inheritance

- C++ supports creating relationships where a subclass inherits data members and methods from a single superclass: single inheritance
- C++ also support inheriting from multiple classes simultaneously: Multiple inheritance
- **This tutorial will only cover single inheritance.**
- Generally speaking…
  - Multiple inheritance requires a **large** amount of design effort
  - It's an easy way to end up with overly complex, fragile code
  - Java and C# (both came after C++) exclude multiple inheritance *on purpose* to avoid problems with it.



- With multiple inheritance a hierarchy like this is possible to create…this is nicknamed the **Deadly Diamond of Death.**

BOSTON UNIVERSITY

C++ Inheritance Syntax

- Inheritance syntax pattern:

  ```
  class SubclassName : public SuperclassName
  ```

- Here the *public* keyword is used.
  - Methods implemented in class Sub can access any public or protected members and methods in Super but cannot access anything that is private.

- Other inheritance types are *protected* and *private.*

```
class Super {
public:
    int i;
protected:
    int j ;
private:
    int k ;
};

class Sub : public Super {
// ...
};
```

Square

- Let's make a subclass of Rectangle called Square.
- Open the NetBeans project *Shapes*
- This has the Rectangle class from Part 2 implemented.
- Add a class named *Square.*
- Make it inherit from Rectangle.

BOSTON UNIVERSITY

Square.h

```
#ifndef SQUARE_H
#define SQUARE_H

#include "Rectangle.h"


class Square : public Rectangle
{
    public:
        Square();
        virtual ~Square();

    protected:

    private:
};

#endif // SQUARE_H
```

Square.cpp

```
#include “Square.h"

Square::Square()
{}

Square::~Square()
{}
```

- Note that subclasses are free to add any number of new methods or members, they are not limited to those in the superclass.

- Class Square inherits from class Rectangle

BOSTON UNIVERSITY

A new Square constructor is needed.

- A square is, of course, just a rectangle with equal length and width.
- The area can be calculated the same way as a rectangle.
- Our Square class therefore needs just one value to initialize it and it can re-use the Rectangle.Area() method for its area.

- Go ahead and try it:
  - Add an argument to the default constructor in Square.h
  - Update the constructor in Square.cpp to do…?
  - Remember Square can access the public members and methods in its superclass

Solution 1

```
#ifndef SQUARE_H
#define SQUARE_H

#include "Rectangle.h"


class Square : public Rectangle
{
    public:
        Square(float width);
        virtual ~Square();

    protected:

    private:
};

#endif // SQUARE_H
```

```
#include "Square.h"

Square::Square(float length):
m_width (length), m_length(length)
{

}
```

- Square can access the public members in its superclass.
- Its constructor can then just assign the length of the side to the Rectangle m_width and m_length.
- This is unsatisfying – while there is nothing *wrong* with this it's not the OOP way to do things.
- Why re-code the perfectly good constructor in Rectangle?

BOSTON UNIVERSITY

The delegating constructor

- C++11 added a new constructor type called the delegating constructor.
- Using member initialization lists you can call one constructor from another.
- Even better: with member initialization lists C++ can call superclass constructors!

```
class class_c {
public:
    int max;
    int min;
    int middle;

    class_c(int my_max) {
        max = my_max > 0 ? my_max : 10;
    }
    class_c(int my_max, int my_min) : class_c(my_max) {
        min = my_min > 0 && my_min < max ? my_min : 1;
    }
    class_c(int my_max, int my_min, int my_middle) :
            class_c (my_max, my_min){
        middle = my_middle < max &&
              my_middle > min ? my_middle : 5;
}
};
```

```
Square::Square(float length) :
      Rectangle(length,length)
{
  // other code could go here.
}
```

Reference:

https://msdn.microsoft.com/en-us/library/dn387583.aspx

BOSTON UNIVERSITY

Solution 2

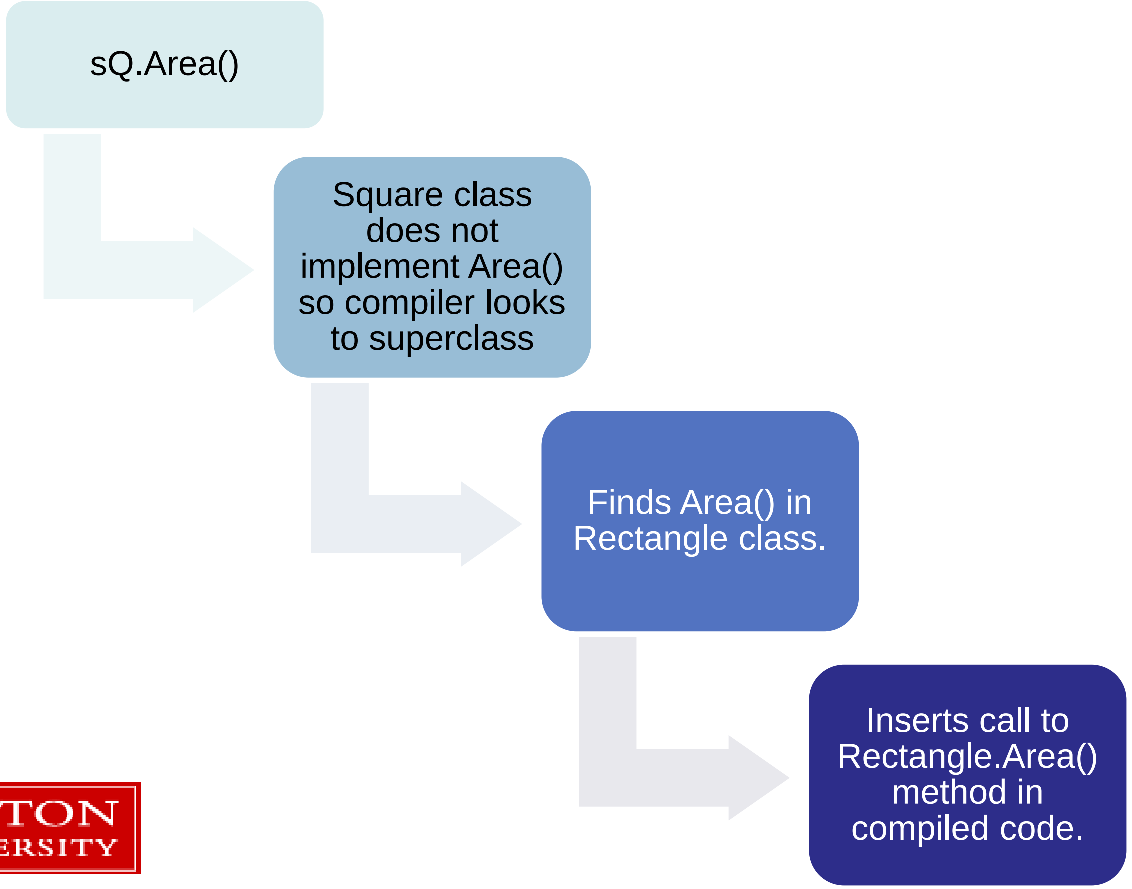
```
#ifndef SQUARE_H
#define SQUARE_H

#include "Rectangle.h"


class Square : public Rectangle
{
    public:
        Square(float width);
        virtual ~Square();

    protected:

    private:
};

#endif // SQUARE_H
```

```
#include "Square.h"

Square::Square(float length) :
     Rectangle(length, length) {}
```

- Square can directly call its superclass constructor and let the Rectangle constructor make the assignment to m_width and m_length.
- This saves typing, time, and **reduces the chance of adding bugs to your code**.
  - The more complex your code, the more compelling this statement is.
- Code re-use is one of the prime reasons to use OOP.

Trying it out in main()

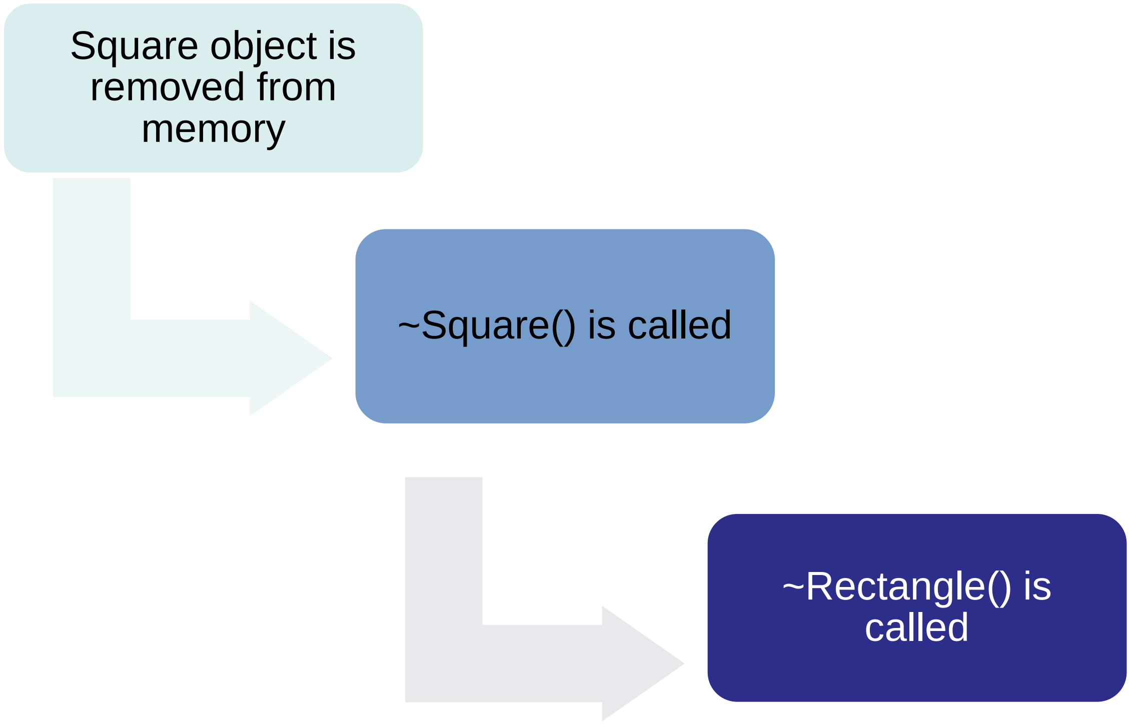
- What happens behind the scenes when this is compiled….




```
#include <iostream>

using namespace std;

#include “Square.h"

int main()
{
    Square sQ(4) ;

    // Uses the Rectangle Area() method!
    cout << sQ.Area() << endl ;

    return 0;
}
```

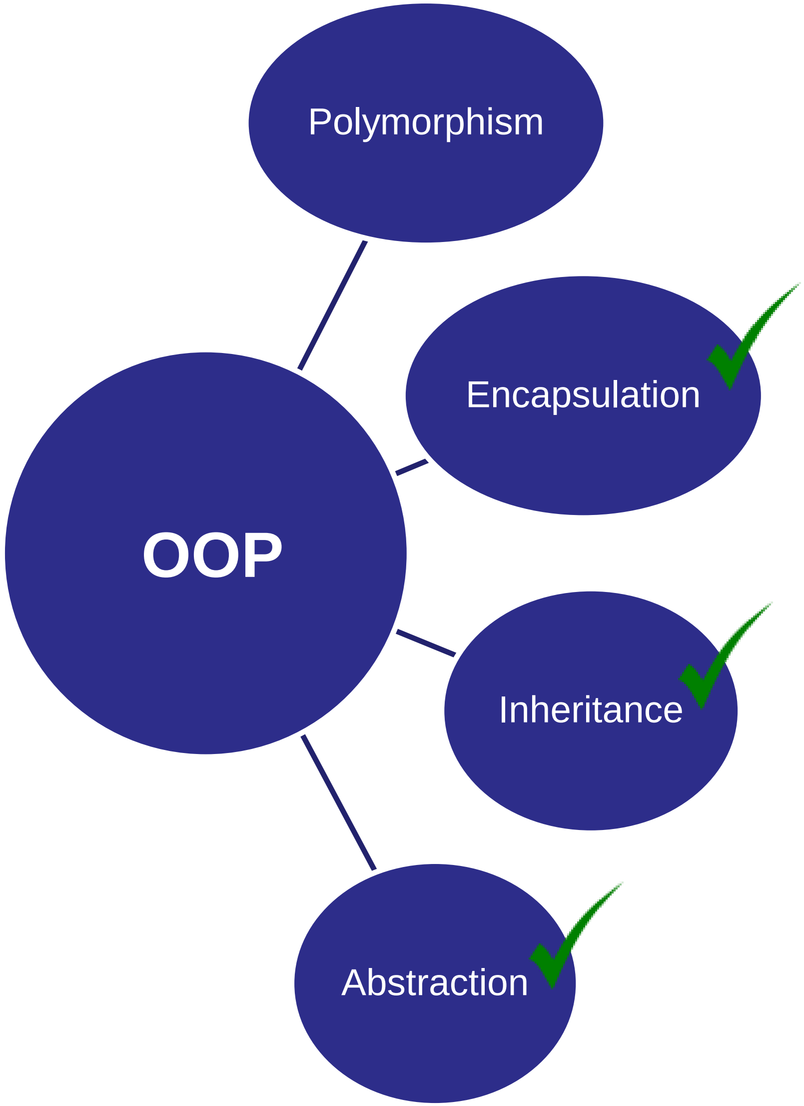
More on Destructors

- When a subclass object is removed from memory, its destructor is called as it is for any object.
- Its superclass destructor is than *also* called .
- Each subclass should only clean up its own problems and let superclasses clean up theirs.



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.

Overriding Methods

- Sometimes a subclass needs to have the same interface to a method as a superclass but 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.

```
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?

```
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!!
```

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

BOSTON UNIVERSITY

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.

BOSTON UNIVERSITY

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

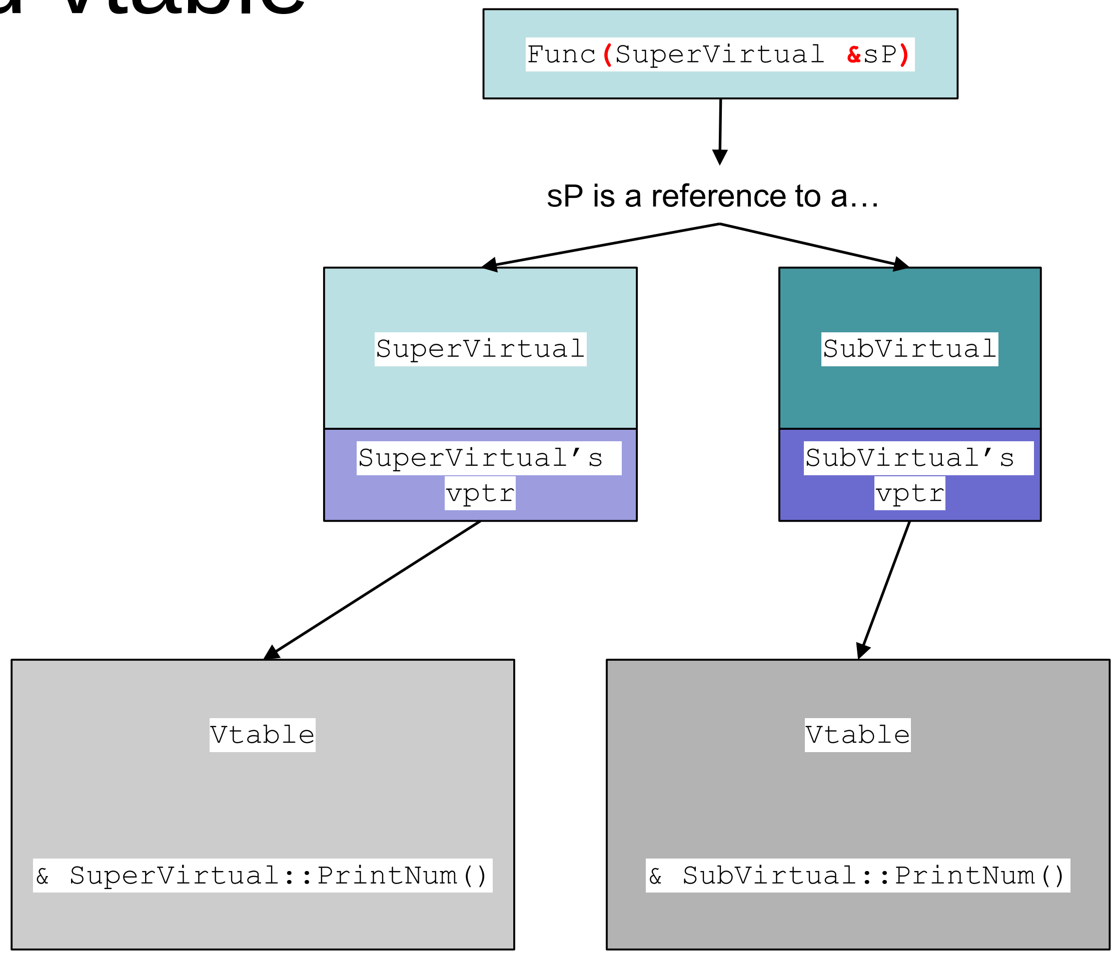
- Leaving out the virtual keyword on a method that is overridden results in the compiler deciding *at compile time* which version (subclass or superclass) of the method to call.
- This is called early or static *binding*.
- At compile time, a function that takes a superclass argument will only call the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

When to make methods virtual

- If a method will be (or might be) overridden in a subclass, make it virtual
  - There is a *minuscule* 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() ;
}
```

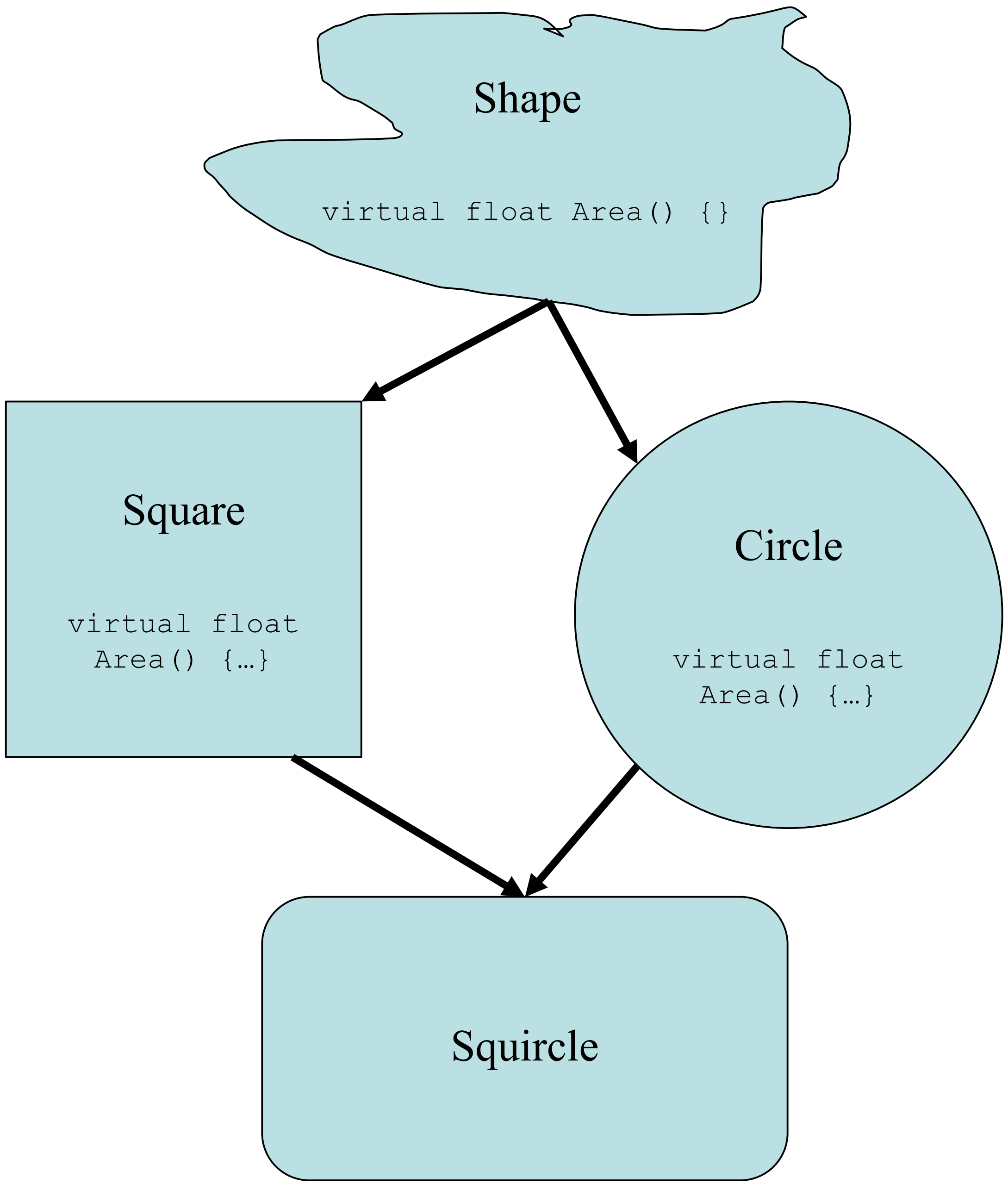
- Called by **value** – early binding to PrintNum even though it's virtual!

- 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.

BOSTON UNIVERSITY

- 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:

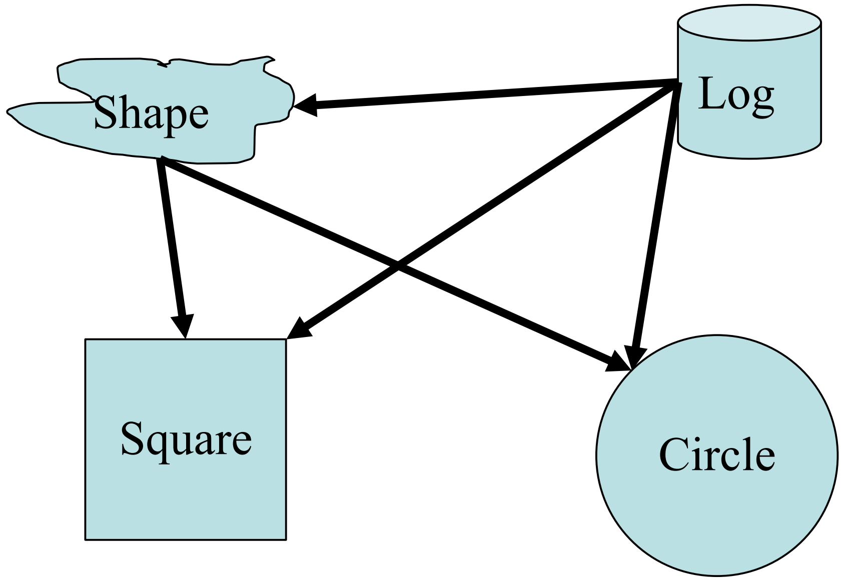
    class Squircle : public Square, public Circle
- The Shape class implements an empty Area() method. The Square and Circle classes override it. Squircle does not.
- Under late binding, which version of Area is accessed from Squircle? Square.Area() or Circle.Area()?



BOSTON UNIVERSITY

Interfaces

- 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 want each class to have a Log() method that writes some info to a file.
  - Implement with an interface.

BOSTON UNIVERSITY

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 ;
```

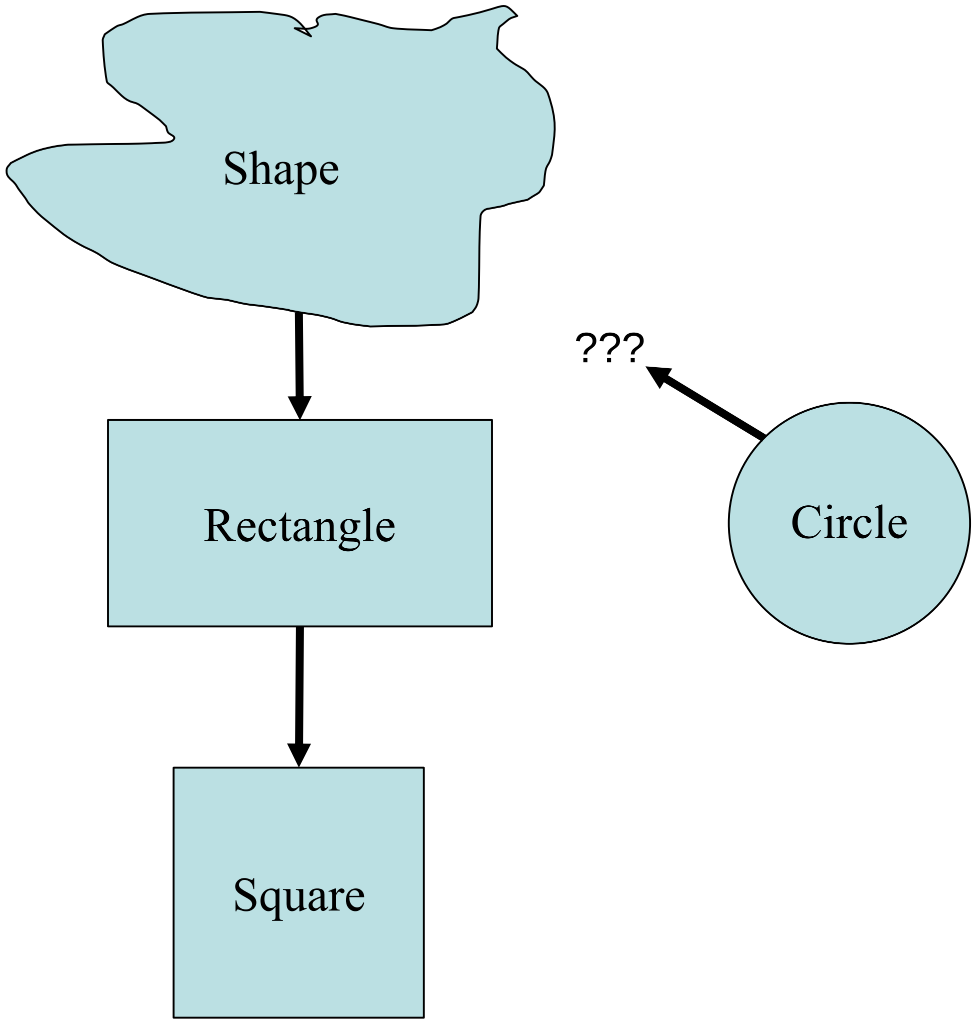
- Once the LogInfo() is uncommented it will compile.

```
#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

Putting it all together

- Now let's revisit our Shapes project.
- 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

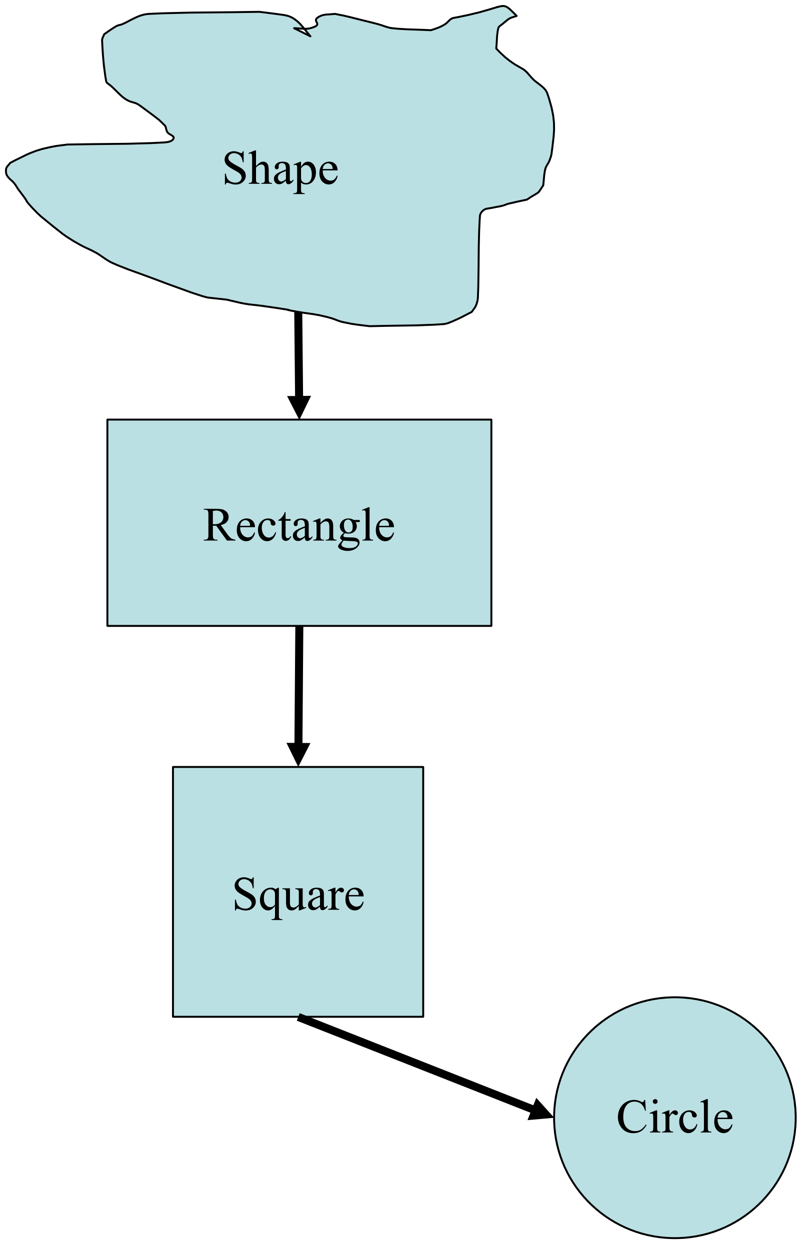
- Add inheritance from Shape to the Rectangle class
- Add a Circle class, inheriting from wherever you like.
- Implement Area() for the Circle

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

BOSTON UNIVERSITY

A Potential Solution

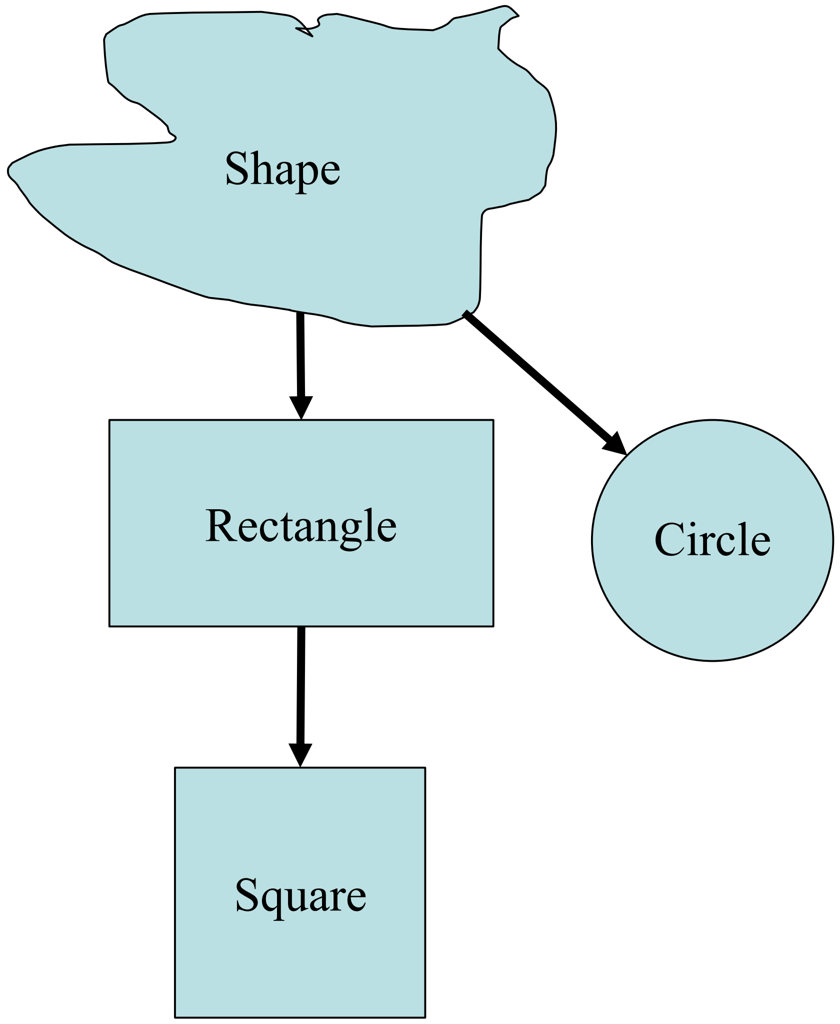
- A Circle has one dimension (radius), like a Square.
  - Would only need to override the Area() method
- But…
  - Would be storing the radius in the members m_width and m_length. This is not a very obvious to someone else who reads your code.
- Maybe:
  - Change m_width and m_length names to m_dim_1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

```cpp
#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!