

Introduction to C++: Part 3

Tutorial Outline: Part 3

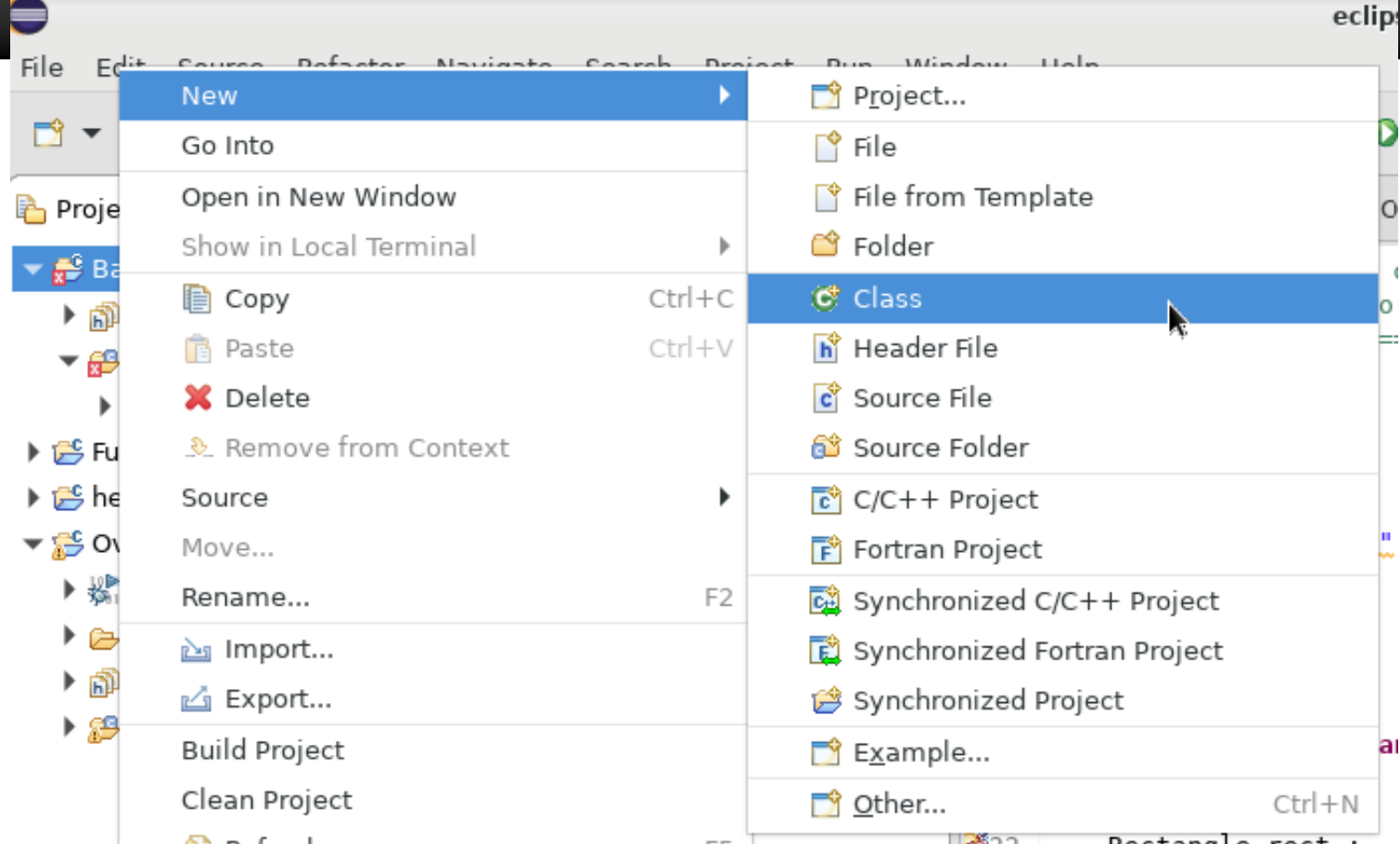
- Defining Classes
- Class inheritance
- Public, private, and protected access
- Virtual functions

A first C++ class

- Open project **Basic_Rectangle**.
- We'll add our own custom class to this project.
- A C++ class consists of 2 files: a header file (.h) and a source file (.cpp)
- The header file contains the definitions for the types and names of members, methods, and how the class relates to other classes (if it does).
- The source file contains the code that implements the functionality of the class
- Sometimes there is a header file for a class but no source file.

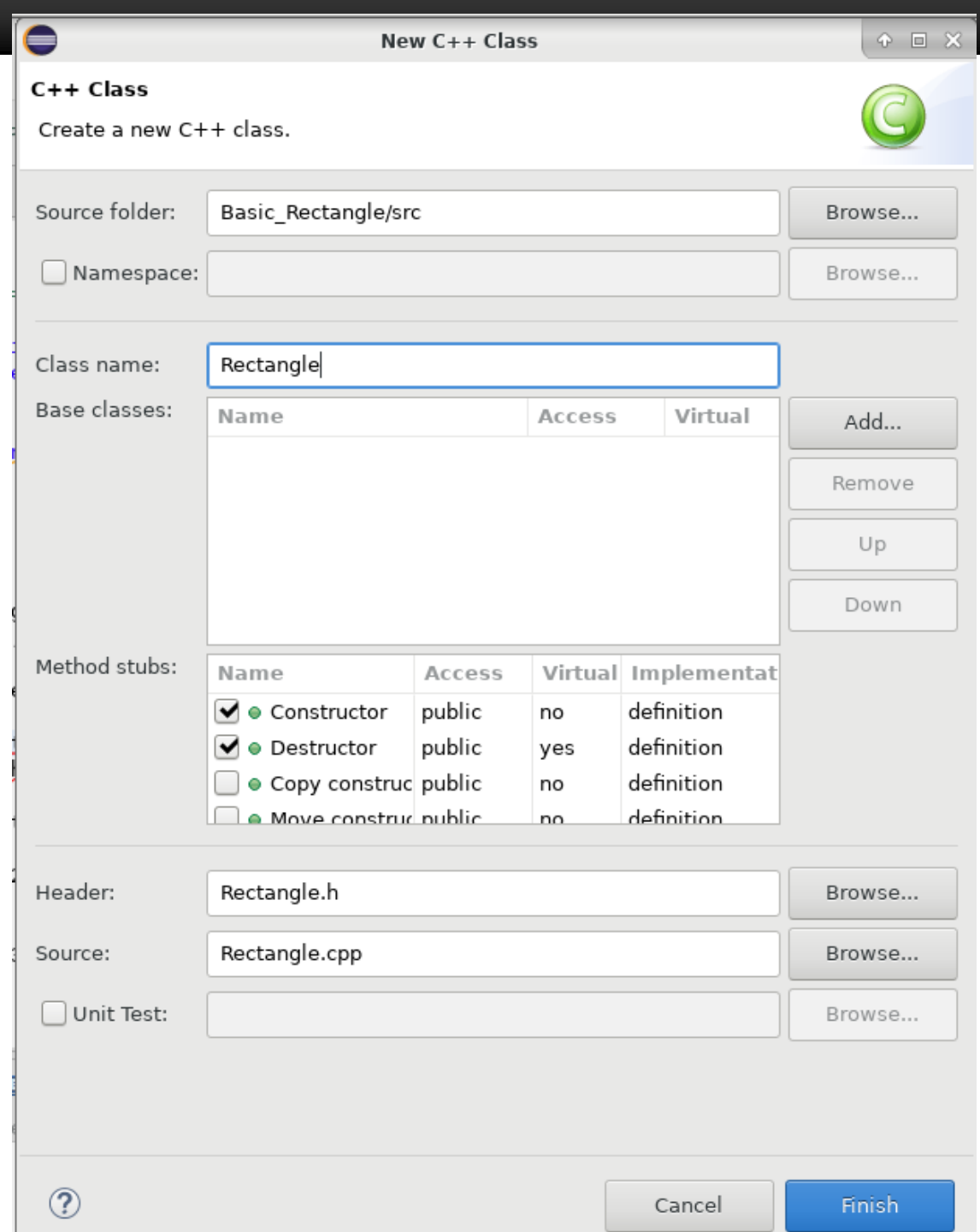


Using Eclipse



- An IDE is very useful for setting up code that follows patterns and configuring the build system to compile them.
- This saves time and effort for the programmer.
- Right-click on the Basic_Rectangle project and choose *New*→*Class*

- Give it the name *Rectangle* and click the Finish button.
- Open the new files *Rectangle.h* and *Rectangle.cpp*



Rectangle.h

```
/*  
 * Rectangle.h  
 *  
 * Created on: Sep 9, 2019  
 * Author: bgregor  
 */  
  
#ifndef RECTANGLE_H_  
#define RECTANGLE_H_  
  
class Rectangle {  
public:  
    Rectangle();  
    virtual ~Rectangle();  
};  
  
#endif /* RECTANGLE_H_ */
```

keyword

Class name

Curly brace

Access
control

Curly brace
and a
semi-colon.

Default declared methods

- `Rectangle()`;
 - A *constructor*. Called when an object of this class is created.
- `~Rectangle()`;
 - A *destructor*. Called when an object of this class is removed from memory, i.e. destroyed.
 - Ignore the *virtual* keyword for now.

```
/*  
 * Rectangle.h  
 *  
 * Created on: Sep 9, 2019  
 * Author: bgregor  
 */  
  
#ifndef RECTANGLE_H_  
#define RECTANGLE_H_  
  
class Rectangle {  
public:  
    Rectangle();  
    virtual ~Rectangle();  
};  
  
#endif /* RECTANGLE_H_ */
```

Rectangle.cpp

Header file included

Class_name:: pattern indicates the method declared in the header is being implemented in code here.

Methods are otherwise regular functions with arguments () and matched curly braces {}.

```
/*  
 * Rectangle.cpp  
 *  
 * Created on: Sep 2, 2019  
 * Author: bgregor  
 */  
  
#include "Rectangle.h"  
  
Rectangle::Rectangle() {  
    // TODO Auto-generated constructor stub  
}  
  
Rectangle::~Rectangle() {  
    // TODO Auto-generated destructor stub  
}
```


Let's add some functionality

- A Rectangle class should store a length and a width.
- To make it useful, let's have it supply an Area() method to compute its own area.
- Edit the header file to look like the code to the right.

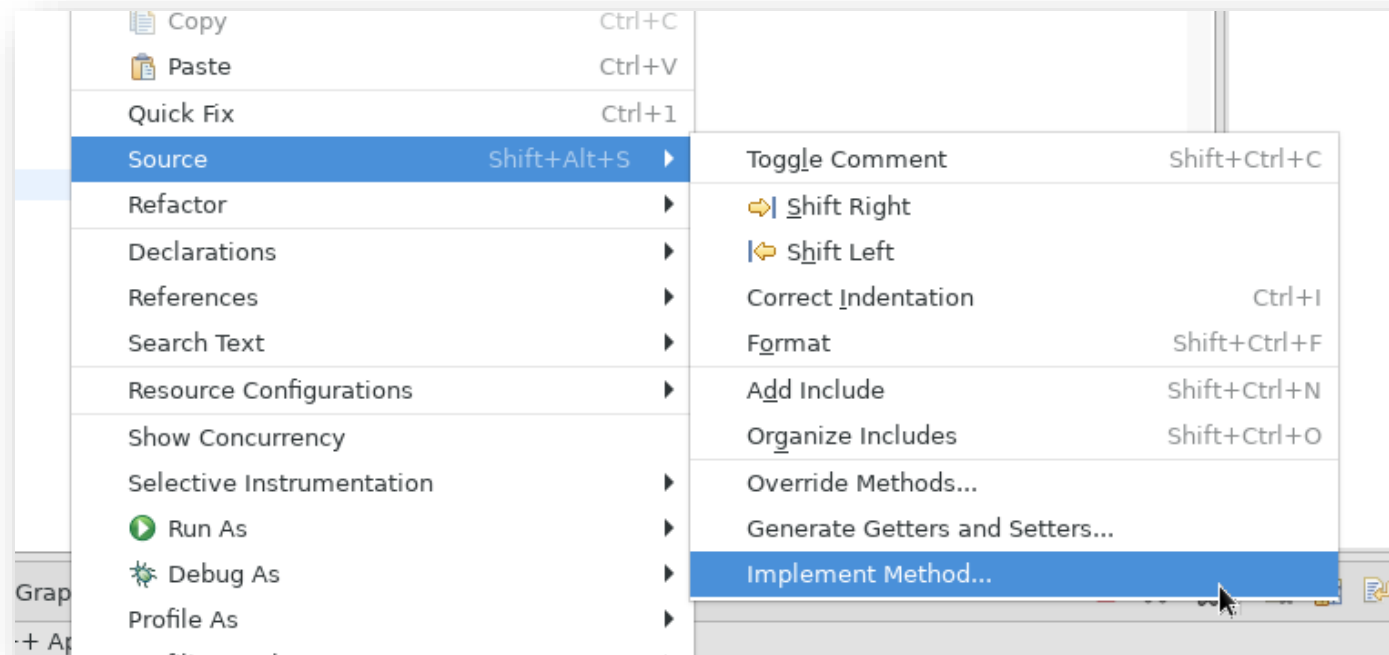
```
class Rectangle {  
public:  
    Rectangle();  
    virtual ~Rectangle();  
  
    float m_length ;  
    float m_width ;  
  
    float Area() ;  
    float ScaledArea(const float scale);  
  
};
```

Encapsulation

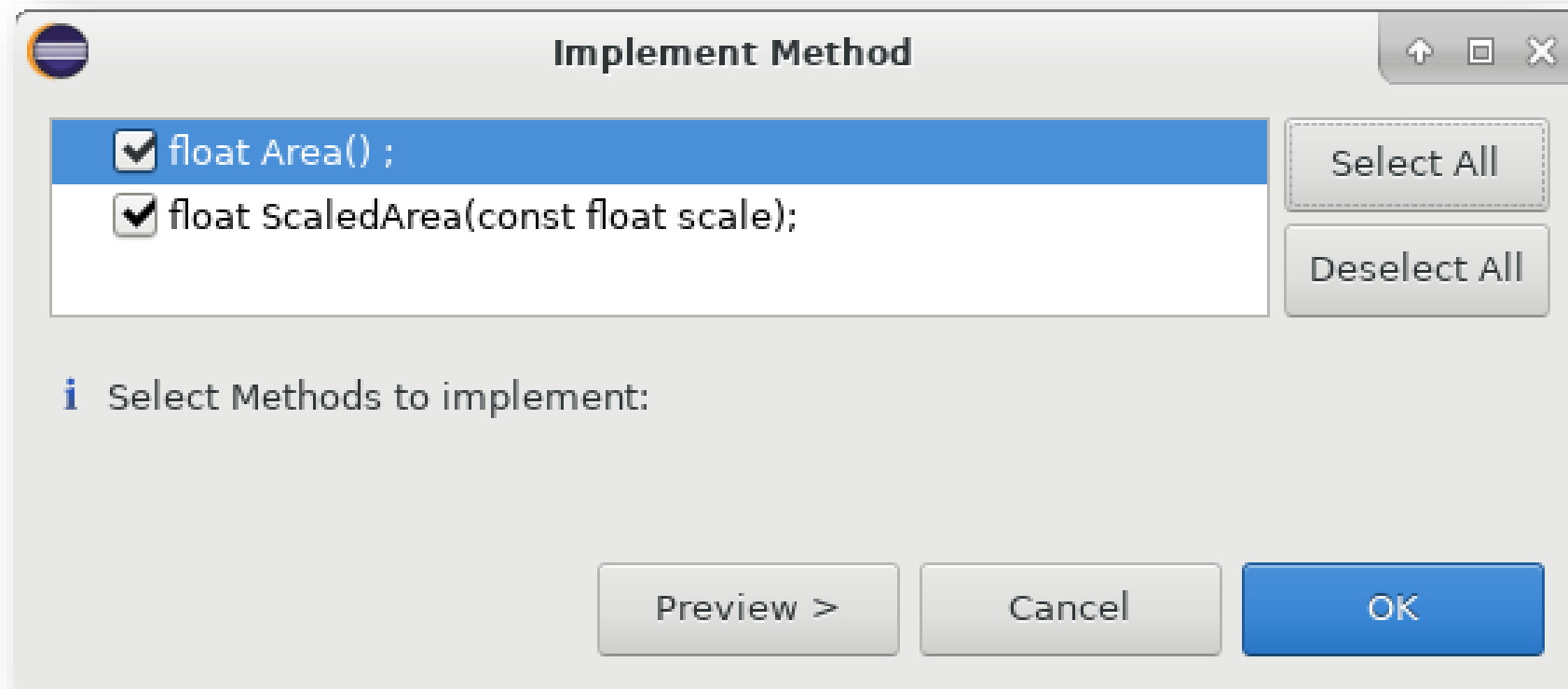
- Bundling the data and area calculation for a rectangle into a single class is an example of the concept of *encapsulation*.

The code for the two methods is needed

- Right-click in the Rectangle.h window and choose Source→Implement Methods



- Click *Select All* then click OK.



Fill in the methods

- Step 1: add some comments.
- Step 2: add some code.

```
float Rectangle::Area() {  
    return m_length * m_width ;  
}  
  
float Rectangle::ScaledArea(const float scale) {  
    // Calculate the area and multiply it  
    // by the scale argument. Return it.  
}
```

- **Member variables can be accessed as though they were passed to the method.**
- Methods can also call each other.
- Fill in the Area() method and then **write your own ScaledArea()**. Don't forget to compile!

Using the new class

- Open *Basic_Rectangle.cpp*
- Add an include statement for the new `Rectangle.h`
- Create a `Rectangle` object and call its methods.
- We'll do this together...

Special methods

- There are several methods that deal with creating and destroying objects.
- These include:
 - *Constructors* – called when an object is created. Can have many defined per class.
 - *Destructor* – one per class, called when an object is destroyed
 - *Copy* – called when an object is created by copying an existing object
 - *Move* – a feature of C++11 that is used in certain circumstances to avoid copies.

Construction and Destruction

- The *constructor* is called when an object is created.
- This is used to initialize an object:
 - Load values into member variables
 - Open files
 - Connect to hardware, databases, networks, etc.

- The *destructor* is called when an object goes *out of scope*.
- Example:

```
void function() {  
    ClassOne c1 ;  
}
```

- Object c1 is created when the program reaches the first line of the function, and destroyed when the program leaves the function.

When an object is instantiated...

- The rT object is created in memory.
- When it is created its *constructor* is called to do any necessary initialization.
- The constructor can take any number of arguments like any other function but it *cannot* return any values.
- What if there are multiple constructors?
 - The compiler follows standard function overload rules.

```
#include "Rectangle.h"

int main(int argc, char** argv) {
    Rectangle rT ;
    rT.m_width = 1.0 ;

    return 0;
}
```

```
Rectangle::Rectangle() {
    }
```

Note the constructor has no return type!

A second constructor

rectangle.h

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

        /* etc */
};
```

rectangle.cpp

```
#include "rectangle.h"

/* C++11 style */
Rectangle::Rectangle(const float width,
                    const float length) :
    m_width(width),
    m_length(length)
{
    /* extra code could go here */
}
```

- Adding a second constructor is similar to overloading a function.
- Here the modern C++11 style is used to set the member values – this is called a *member initialization list*

Member Initialization Lists

- Syntax:

Members assigned and separated with commas. The order doesn't matter.

```
MyClass(int A, OtherClass &B, float C):  
    m_A(A),  
    m_B(B),  
    m_C(C) {  
        /* other code can go here */  
}
```

Colon goes here

Additional code can be added in the code block.

And now use both constructors

- Both constructors are now used. The new constructor initializes the values when the object is created.
- Constructors are used to:
 - Initialize members
 - Open files
 - Connect to databases
 - Etc.

```
#include <iostream>

using namespace std;

#include "Rectangle.h"

int main(int argc, char** argv)
{
    Rectangle rT ;
    rT.m_width = 1.0 ;
    rT.m_length = 2.0 ;

    cout << rT.Area() << endl ;

    Rectangle rT_2(2.0,2.0) ;
    cout << rT_2.Area() << endl ;

    return 0;
}
```

Default values

- C++11 added the ability to define default values in headers in an intuitive way.
- Pre-C++11 default values would have been coded into constructors.
- If members with default values get their value set in constructor than the default value is ignored.
 - i.e. no “double setting” of the value.

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

    Rectangle(const Rectangle& orig);
    virtual ~Rectangle();

    float m_length = 0.0 ;
    float m_width = 0.0 ;

    float Area() ;
    float ScaledArea(const float scale);

private:
};
```

Default constructors and destructors

- The two methods created by Eclipse automatically are explicit versions of the **default** C++ constructors and destructors.
- Every class has them – if you don't define them then empty ones that do nothing will be created for you by the compiler.
 - If you really don't want the default constructor you can delete it with the *delete* keyword.
 - Also in the header file you can use the *default* keyword if you like to be clear that you are using the default.

```
class Foo {  
    public:  
        Foo() = delete ;  
        // Another constructor  
        // must be defined!  
        Foo(int x) ;  
};  
  
class Bar {  
    public:  
        Bar() = default ;  
};
```

Custom constructors and destructors

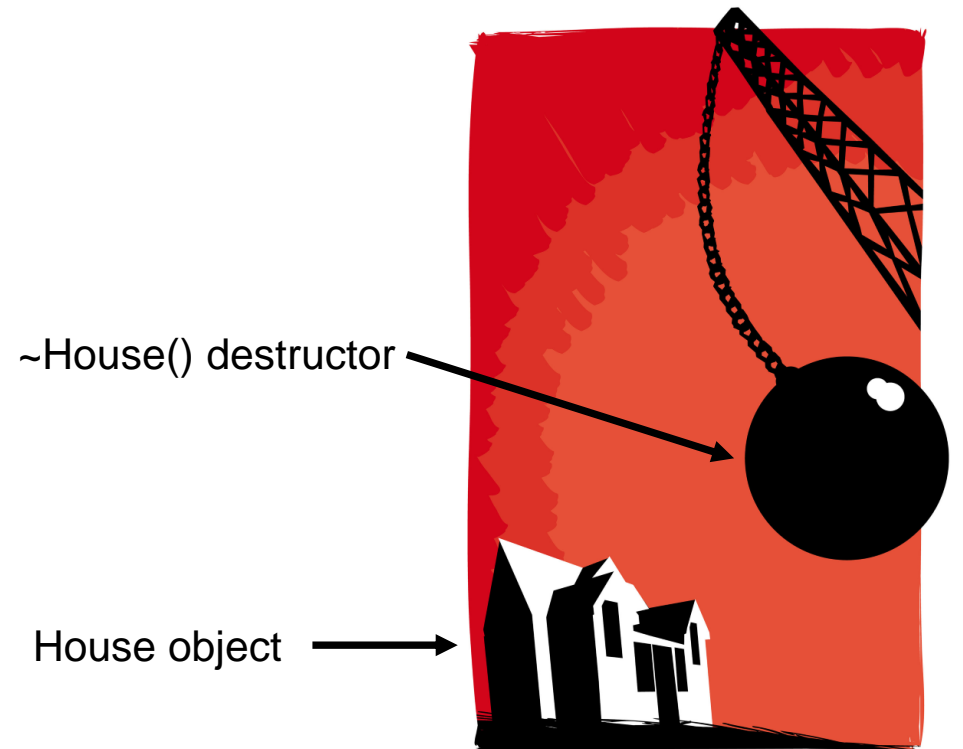
- You must define your own constructor when you want to initialize an object with arguments.
- A custom destructor is **always** needed when internal members in the class need special handling.
 - Examples: manually allocated memory, open files, hardware drivers, database or network connections, custom data structures, etc.

Destructors

- Destructors are called when an object is destroyed.
- Destructors have no return type.
- There is only **one** destructor allowed per class.
- Objects are destroyed when they go out of *scope*.
- Destructors are never called explicitly by the programmer. Calls to destructors are inserted automatically by the compiler.

This class just has 2 floats as members which are automatically removed from memory by the compiler.

```
Rectangle::~~Rectangle()  
{  
  
}
```



Destructors

- Example:

```
class Example {  
    public:  
        Example() = delete ;  
        Example(int count) ;  
  
        virtual ~Example() ;  
  
        // A pointer to some memory  
        // that will be allocated.  
        float *values = nullptr ;  
};
```

```
Example::Example(int count) {  
    // Allocate memory to store "count"  
    // floats.  
    values = new float[count];  
}  
  
Example::~~Example() {  
    // The destructor must free this  
    // memory. Only do so if values is not  
    // null.  
    if (values) {  
        delete[] values ;  
    }  
}
```

Scope

- Scope is the region where a variable is valid.
- Constructors are called when an object is created.
- Destructors are only ever called implicitly.

```
int main() { // Start of a code block
    // in main function scope
    float x ; // No constructors for built-in types
    ClassOne c1 ; // c1 constructor ClassOne() is called.
    if (1){ // Start of an inner code block
        // scope of c2 is this inner code block
        ClassOne c2 ; //c2 constructor ClassOne() is called.
    } // c2 destructor ~ClassOne() is called.
    ClassOne c3 ; // c3 constructor ClassOne() is called.
} // leaving program, call destructors for c3 and c1 ~ClassOne()
// variable x: no destructor for built-in type
```

Copy, Assignment, and Move Constructors

- The compiler will automatically create constructors to deal with copying, assignment, and moving. NetBeans filled in an empty default copy constructor for us.
- How do you know if you need to write one?
 - When the code won't compile and the error message says you need one!
 - OR unexpected things happen when running.
- You may require custom code when...
 - dealing with open files inside an object
 - The class manually allocated memory
 - Hardware resources (a serial port) opened inside an object
 - Etc.

```
Rectangle rT_1(1.0,2.0) ;  
// Now use the copy constructor  
Rectangle rT_2(rT_1) ;  
// Do an assignment, with the  
// default assignment operator  
rT_2 = rT_1 ;
```

Templates and classes

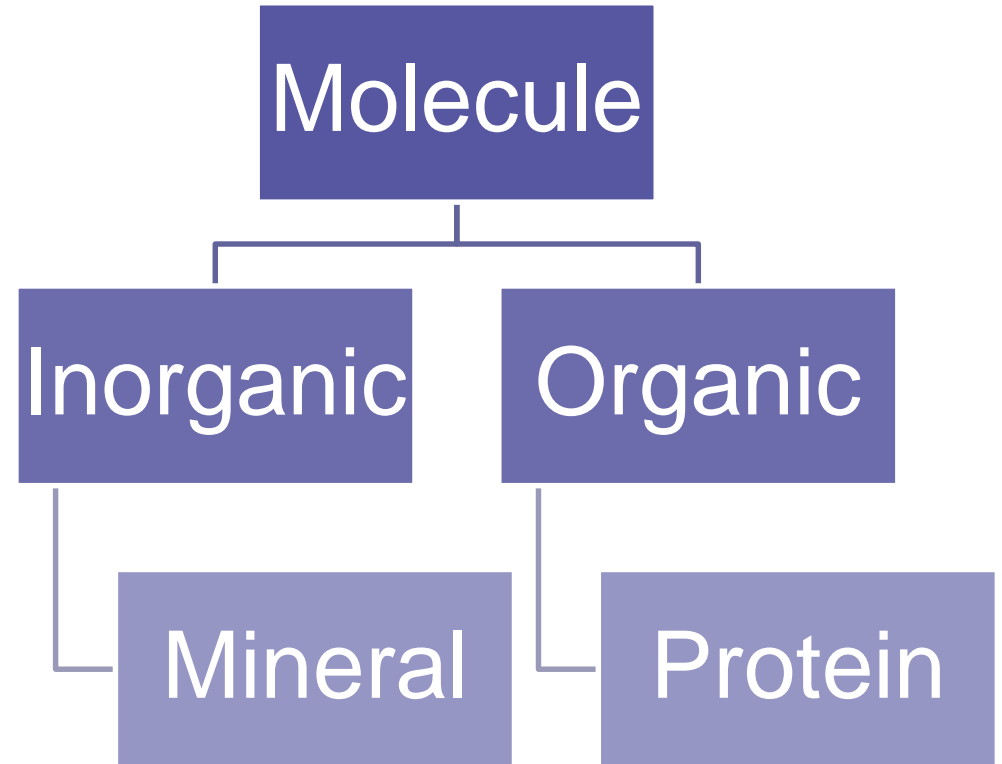
- Classes can also be created via templates in C++
- Templates can be used for type definitions with:
 - Entire class definitions
 - Members of the class
 - Methods of the class
- Templates can be used with class inheritance as well.
- This topic is way beyond the scope of this tutorial!

Tutorial Outline: Part 3

- Defining Classes
- Class inheritance
- Public, private, and protected access
- Virtual functions

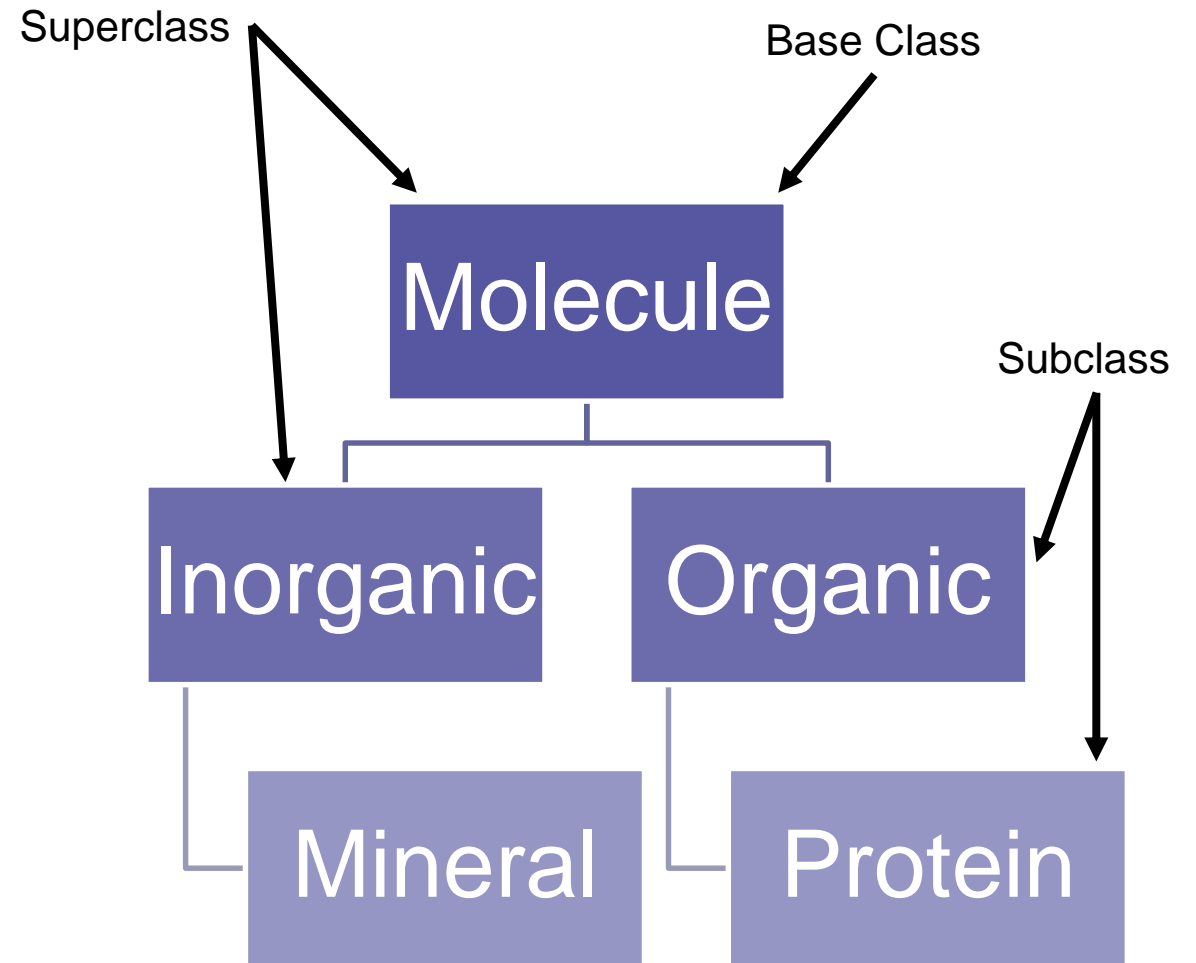
Inheritance

- 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!

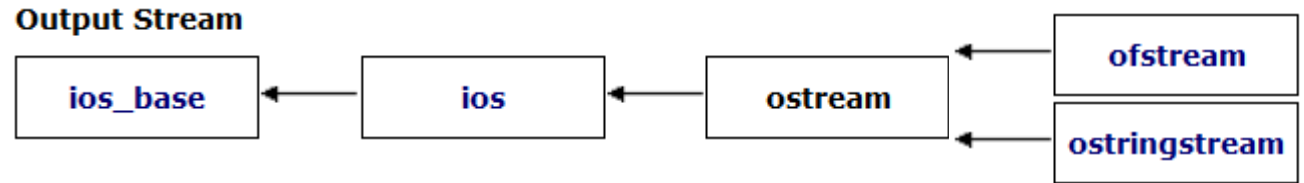


Inheritance

- 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.



Inheritance in Action



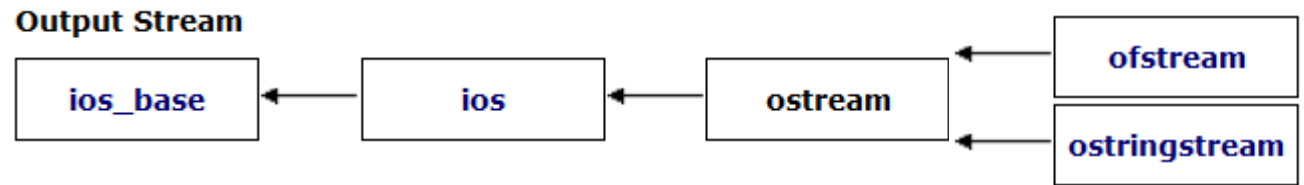
- Streams in C++ are series of characters – the C++ I/O system is based on this concept.
- cout** is an object of the class *ostream*. It is a write-only series of characters that prints to the terminal.
- There are two subclasses of *ostream*:
 - ofstream* – write characters to a file
 - ostringstream* – write characters to a string

- Writing to the terminal is straightforward:

```
cout << some_variable ;
```

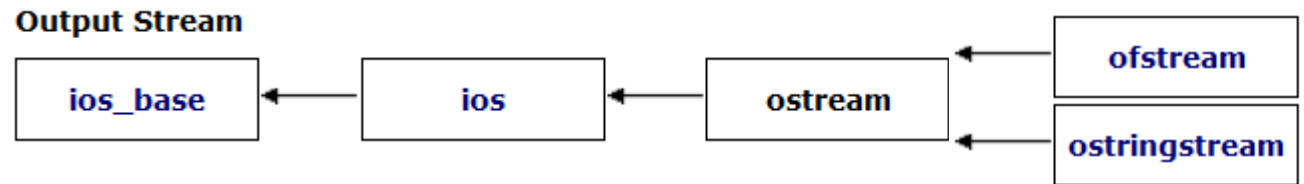
- How might an object of class *ofstream* or *ostringstream* be used if we want to write characters to a file or to a string?

Inheritance in Action



- For *ofstream* and *ostringstream* 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.
- *ostringstream* 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() ;
}
```

Tutorial Outline: Part 3

- Defining Classes
- Class inheritance
- Public, private, and protected access
- Virtual functions

Public, protected, private

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

“There are only two things wrong with C++: The initial concept and the implementation.”

– Bertrand Meyer (inventor of the Eiffel OOP language)

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

        float m_width ;
        float m_length ;

        float Area() ;

    protected:

    private:
};
```

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 ;  
};
```

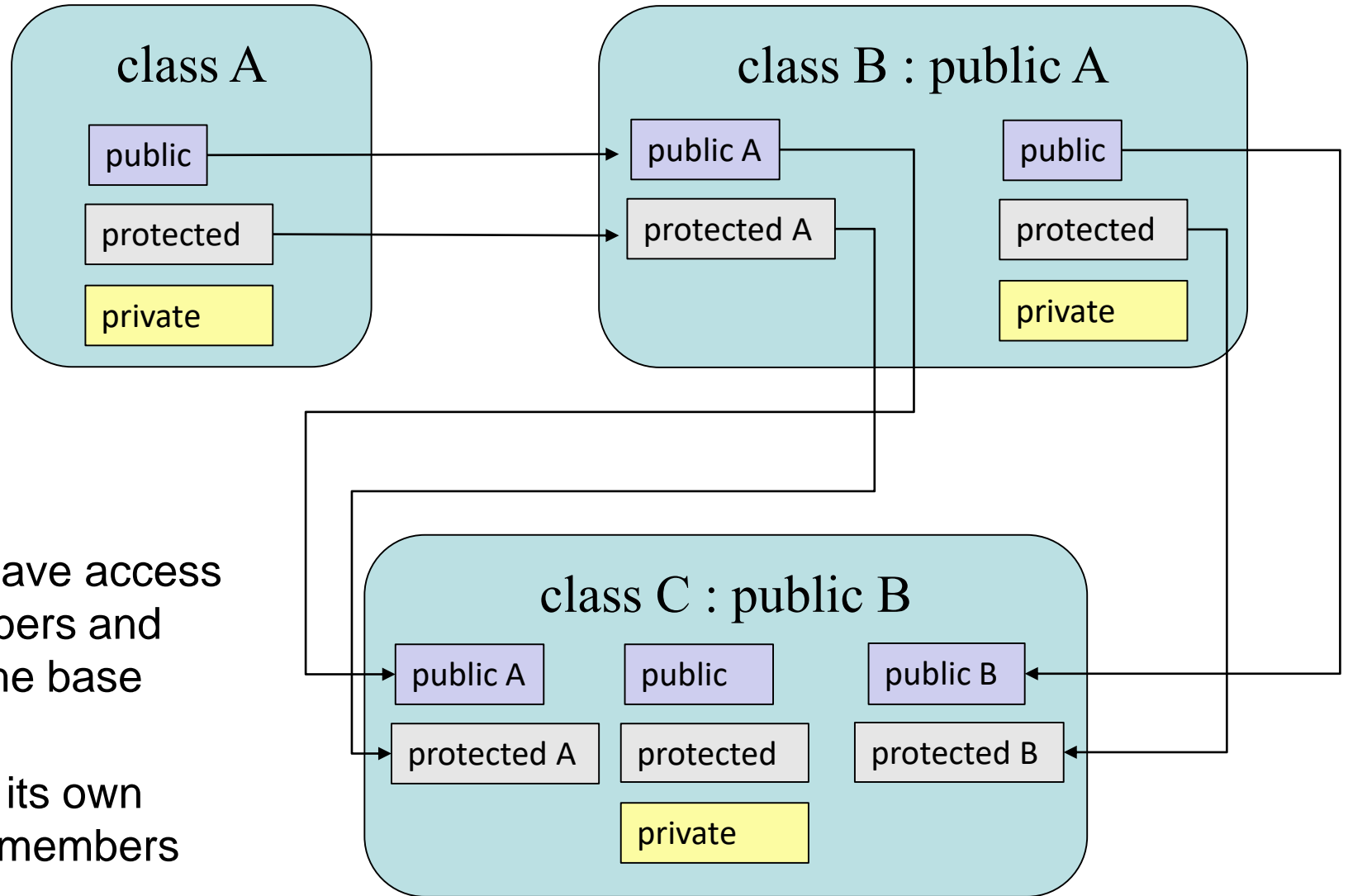
Inheritance

```
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
```

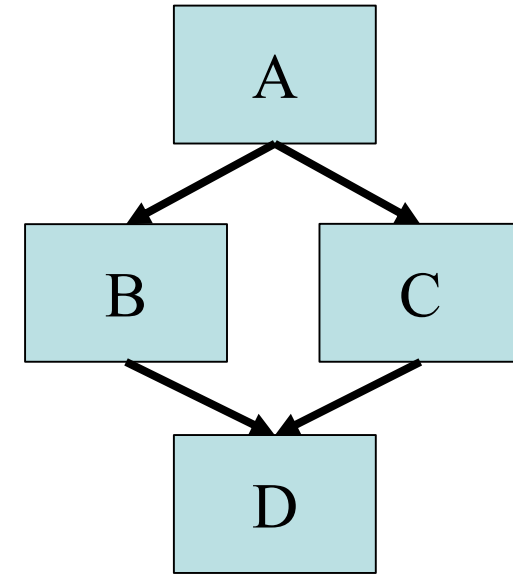
Inheritance



- 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.

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**.

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.

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

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?

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>

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
```

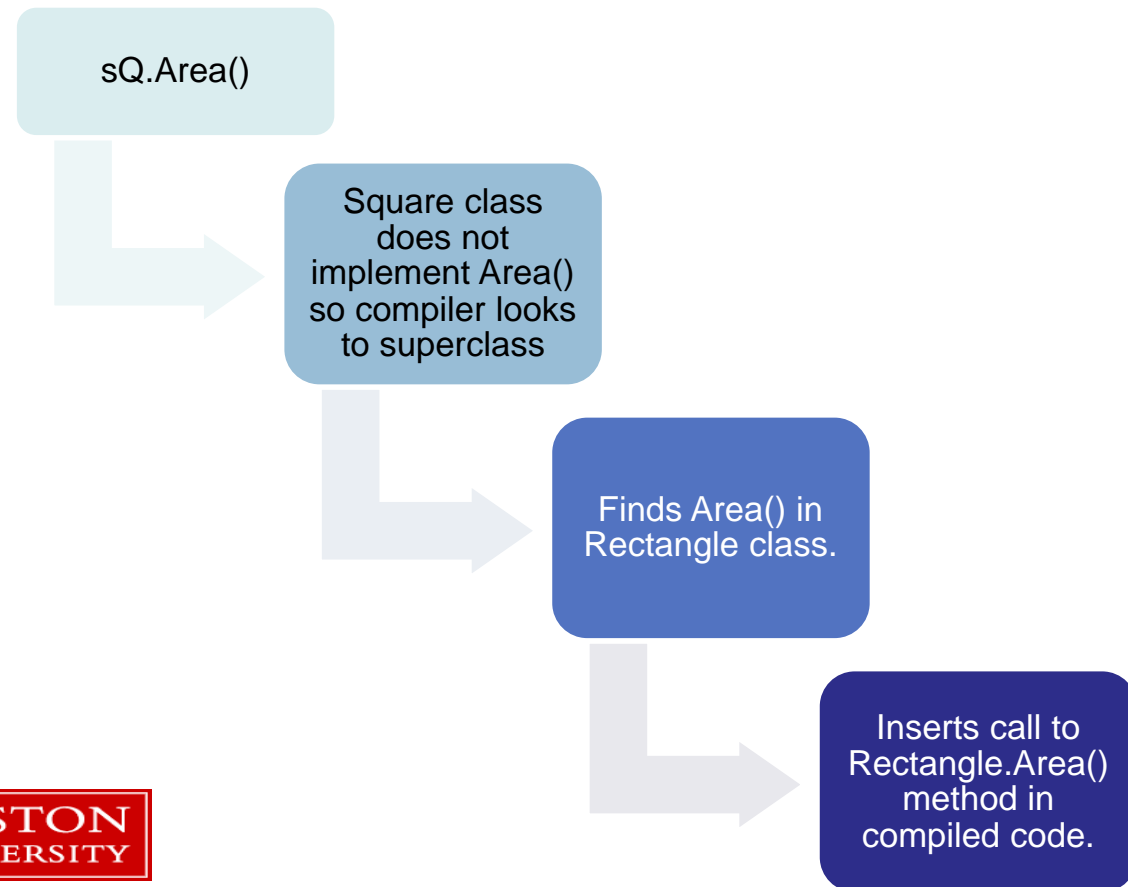
```
#include "Square.h"

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 then *also* called .
- Each subclass should only clean up its own problems and let superclasses clean up theirs.

