

# Introduction to Programming using C++

## Lecture Eight: Toward OO Design

Carl Gwilliam

[gwilliam@hep.ph.liv.ac.uk](mailto:gwilliam@hep.ph.liv.ac.uk)

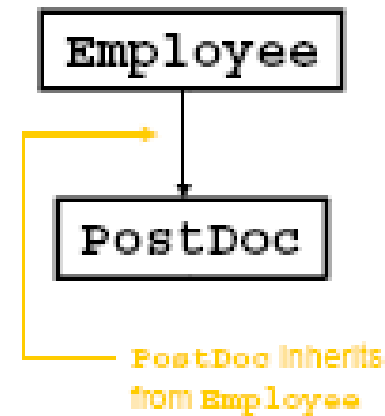
<http://hep.ph.liv.ac.uk/~gwilliam/cppcourse>

# The Pillars of OO Design

- An Object Orientated program should have the following three features:
  - Encapsulation:  
The implementation of an object is hidden from the it's interface.
  - Delegation:  
Effort is delegated to well defined single purpose objects.
  - Categorisation:  
The subject of this lecture...
- An OO designed program determines how the modeled system responds to a given event.
- An OO program is not written to answer a specific question, it is written to answer any number of questions that may be asked in the future.

# Inheritance

- In C++, categorisation is given by **public inheritance**.
- **Inheritance hierarchies** capture the relationship between objects: an object inherits all the characteristics of other objects higher up inheritance hierarchy.
- Inheritance = *is a kind of* relationship



## Alternatively:

- Each object should be distinct and provide a well defined service but there can be great deal of overlap in functionality between different objects.
- Commonality across multiple classes can be extracted and placed in a “super” class.
- Inheritance can provide reusability AND categorisation but the two concepts are not the same.

# Indicating Inheritance in C++

- Inheritance is signified in the **derived** class by placing the **base** class, from which it derives, after a colon:

```
Class One {  
public:  
    :  
protected:  
    :  
private:  
    :  
};
```

```
Class Two: public Class One {  
public:  
    :  
protected:  
    :  
private:  
    :  
};
```

- The **derived** class inherits the **public** and **protected** (but not private) members of the **base** class.
- We have a new keyword, **protected**. These members can be accessed by base class & derived classes **only**.

# Revisiting the **Rectangle** Class

- The rectangle could be viewed as a specialisation of a more general shape: the **parallelogram**.
- A rectangle *is a kind of* parallelogram. Therefore the **Rectangle** class can be written to inherit from a **Parallelogram** class.

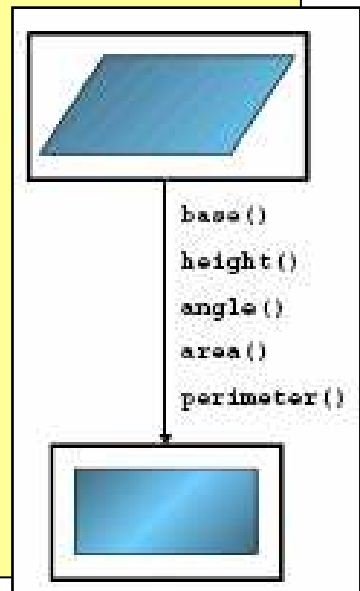
```
class Rectangle {    [Rectangle.h]
public:
    Rectangle();
    Rectangle(Point ll, Point ur);
    Rectangle(double length,
               double width);
    Rectangle(Point ll, double length,
               double width);
    ~Rectangle();
    double length();
    double width();
    double area();
    bool isOverlap(Rectangle a);
private:
    Point itsLowerLeft;
    Point itsLowerRight;
    Point itsUpperLeft;
    Point itsUpperRight;
};
```

# Applying Inheritance

- Rectangle class **inherits** the member data and all the class methods from the Parallelogram class.
- The Rectangle class still needs to contain its own c'tors and d'tors.

```
#include "Parallelogram.h"
// rectangle class declaration
class Rectangle : public Parallelogram {
public:
    Rectangle();
    Rectangle(Point ll, Point ur);
    :
    ~Rectangle(); // d'tor automatically
};                // calls base d'tor
```

```
//parallelogram class declaration
class Parallelogram {
public:
    Parallelogram();
    Parallelogram(double len_a,
        double len_b,double angle);
    ~Parallelogram();
    //angle between non-para sides
    double angle();
    double base();
    double height();
    double area();
    double perimeter();
protected:
    Point itsLowerLeft;
    :
    double itsAngle;
}; [Shapes2.cpp]
```



# Derived Class Constructors

What happens when the **rectangle** constructor is called?

- Inherited data members are initialised by calling the **base** constructor from constructor of the derived class.
- Base constructors are called from **outside** the body of the constructor of the derived class (extension of initialisation list):

```
#include "Parallelogram.h"
// default c'tor for unit rectangle
Rectangle::Rectangle(): Parallelogram(1,1,90.)
{ ... }
:
// constructor for length and width
Rectangle::Rectangle(double length, double width):
    Parallelogram(length,width,90.)
{ ... }
```

# Overriding Class Methods

- Functionality of **rectangle** class can be extended by adding new methods.
- The new class method **area()** **overrides** the method inherited from the Parallelogram class.
- As soon as a member function is created with same name as one in base class **all** overloaded instances of base class method are hidden.
- The **area()** method from the base class can be still called using the full **scope** of the function.

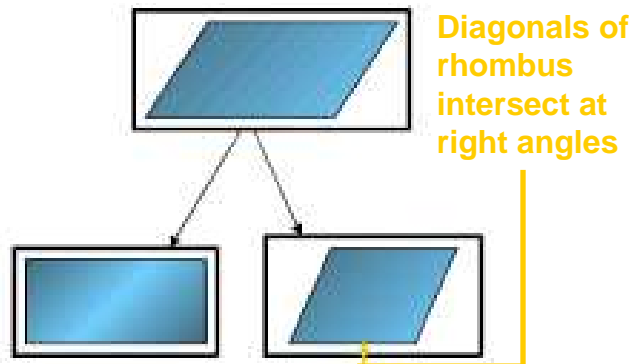
```
//rectangle class declaration
class Rectangle : public
    Parallelogram {
public:
    : //overrides parallelogram...
    double area();    //... area()
    double length(); //same as base
    double width();
    bool isOverlap(Rectangle a);
    } [Shapes3.h]
```

```
Rectangle beta(Point(2,2,0),Point(5,5,0));
cout << "beta base: "
    << beta.base()    << endl;
cout << "beta length: "
    << beta.length() << endl;
cout << "beta area: "
    << beta.area()    << endl;
cout << "beta area: "
    << beta.Parallelogram::area()<<endl;
```



# More than One Derived Class

- Inheritance becomes useful once multiple classes inherit from a single base class.
- **Rhombus** class, as well as **Rectangle** can inherit from **Parallelogram** class.



Diagonal line returned via line c'tor:

// rhombus class declaration

```
class Rhombus : public Parallelogram {
public:
    Rhombus();
    Rhombus(double length, double angle);
    Rhombus(Point ll, double length,
            double angle);
    Rhombus(Point ll, Point lr, double angle);
    ~Rhombus();
    double length() {return itsLength;}
    Line diagonalLine(int corner=1);
    Vect diagonalVect(int corner=1);
    //overrides base class method :
    double perimeter() {return 4*itsLength;}
protected:
    double itsLength;
};
```

**[Shapes4.h]**

```
return Line(itsLowerRight,itsUpperLeft);
```

# Runtime Object Initialisation

How can an object be initialised if type is only decided when run program?

- Declare object in if statement?
  - pointer to heap won't persist beyond end of if block.
- Declare all shape possibilities outside if statement?
  - Not scaleable when more shapes are introduced.
- What is proper solution?

```
cout << "type of parallelogram?  
1 - rectangle, 2 - rhombus\n";  
int request; cin >> request;  
??* newShape = NULL; //what is type?  
if (request == 1) {  
    newShape = new Rectangle(1,2);  
} else if (request == 2) {  
    newShape = new Rhombus(2,45);  
}
```

```
if (request == 1) {  
    Rectangle* newShape = new Rectangle(1,2);  
}
```

```
Rectangle* newShape = NULL;  
Rhombus* newShape2 = NULL;  
if (request == 1) {  
    newShape = new Rectangle(1,2);  
}  
:
```

# Dynamic Binding

- This problem can be solved by exploiting the inheritance relation between the **Rectangle (+ Rhombus)** and the **Parallelogram**.
- If pointer **newShape** is declared with a type of **Parallelogram** it can refer to either a **Rectangle** or **Rhombus** because both of these objects *are kinds of* **Parallelograms**
- Attaching the pointer to an object at runtime is known at **late** or **dynamic binding** (c.f. **compile-time** or **static**).
- But if pointer **newShape** refers to a parallelogram how can it access class methods of rectangle or rhombus?

```
cout << "type of parallelogram? 1 –  
rectangle, 2 - rhombus\n";  
int request; cin >> request;  
Parallelogram* newShape = NULL;  
if (request == 1) {  
    newShape = new Rectangle(1,2);  
} else if (request == 2) {  
    newShape = new Rhombus(2,45);  
}
```

# Virtual Functions

- **newShape** is a pointer to a **Parallelogram**, not to a **Rectangle**. When **area()** is called it is the **Parallelogram** method that is invoked.

```
Parallelogram* newShape = new Rectangle(1,2);
```

- Methods overridden by derived class can be accessed if the **base** method is made **virtual**.

```
// parallelogram declaration
class Parallelogram {
public:
    virtual double area();
    virtual double perimeter();
    :
}; [Shapes5.h]
```

```
// rectangle class declaration
class Rectangle : public
    Parallelogram {
public:
    double area();
    :
}; [Shapes5.h]
```

# Virtual Functions

- This is because calls to **virtual** functions are decided (**binded**) at **run-time** not compile-time, so it can check what object is actually pointed at.

```
myShape->area();           //rectangle methods  
myShape->perimeter()       // are invoked  
//can call base class method by using it's full scope  
myShape->Parallelogram::area();    [UseShapes.cpp]
```

- Be aware, this makes virtual function calls slower
- If method is virtual but not overridden by derived class then base class method will be implemented instead.
- Virtual functions enable **polymorphism** in objects.

# Virtual Destructors

- **newShape** refers to an object on the heap so memory has to be released using **delete**. But rectangle d'tor not called as ptr of type Parallelogram => could cause a memory leak.

```
Parallelogram* newShape = new Rectangle(1,2);  
delete newShape;
```

- Again this is fixed by maxing the **base** d'tor **virtual**, which ensures the proper sequence of d'tors is called

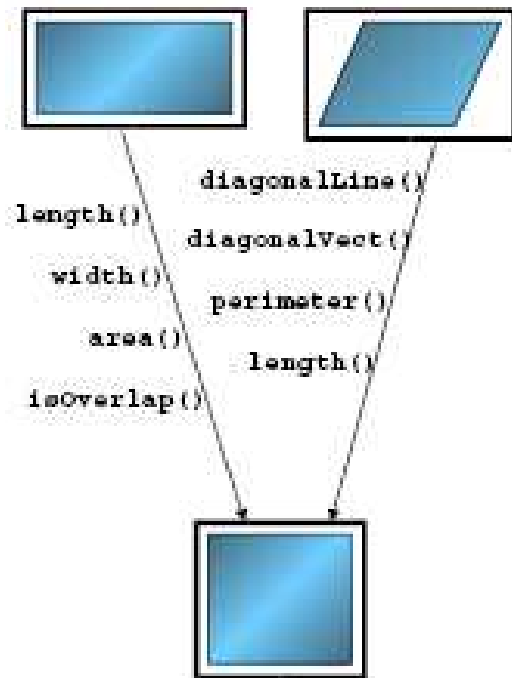
```
// parallelogram declaration  
class Parallelogram {  
public:  
    :  
    virtual ~Parallelogram();  
    :  
};
```

```
// rectangle class declaration  
class Rectangle :  
    public Parallelogram {  
public:  
    ~Rectangle(); // automatically  
    :             // calls base d'tor  
};               // as well after
```

- Base destructor will still be called even if the pointer is **cast** to derived object.

# Multiple Inheritance

- A **Square** class can be constructed that inherits from both **Rectangle** and **Rhombus**:



```
// rectangle class declaration
class Rectangle : public Parallelogram {
:
};
```

```
// rhombus class declaration
class Rhombus : public Parallelogram {
:
};
```

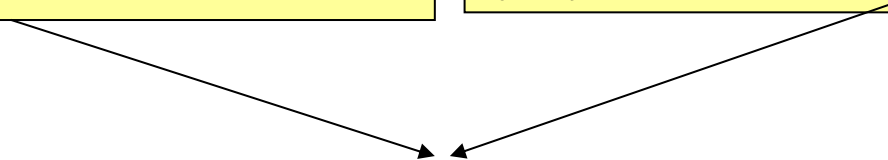
```
// square class declaration
class Square : public Rectangle,
               public Rhombus {
public:
    Square();
    Square(double size);
    Square(Point ll, double size);
    ~Square(); //all functionality inherited
};           //from rectangle & rhombus
```

# Multiple Inheritance

- The constructors for each of the base classes need to be called in turn:

```
// constructor for rhombus  
Rhombus::Rhombus(double length,  
                 double angle):  
    Parallelogram(length,length,90.)  
    {...}
```

```
// constructor for rectangle  
Rectangle::Rectangle(double length,  
                    double width):  
    Parallelogram(length,width,90.)  
    {...}
```



```
// constructor for square  
Square::Square(double size):  
    Rectangle(size,size),Rhombus(size,90)  
    {...}
```

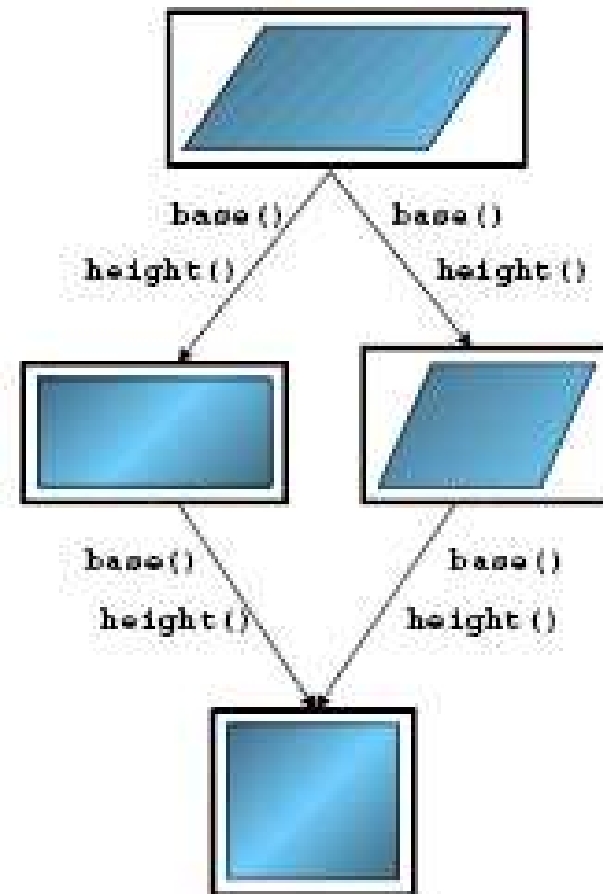
- But multiple inheritance can be dangerous. Can you see why?



# Base Class Ambiguity

- The reason is that the class method **base()** has been inherited via both **Rectangle** and **Rhombus**.
- The call to **base()** from **Square** is ambiguous!
- Further, a **Square** object actually contains **two** sub-objects of Parallelogram
- This ambiguity can be solved by **virtual inheritance** ...

## Dreaded Diamond:



# Virtual Inheritance

- Each **directly** derived class now inherits **virtually** from Parallelogram base class.
- This ensures the most derived class, Square, only inherits **one** sub-object of type Parallelogram.
- It doesn't affect Rectangle or Rhombus classes.
- Base constructor must now be called **directly** from most derived class (unless using default).
- Programs are much easier to read if object relationships are implemented through single public inheritance. Don't use multiple inheritance unless you really have to.

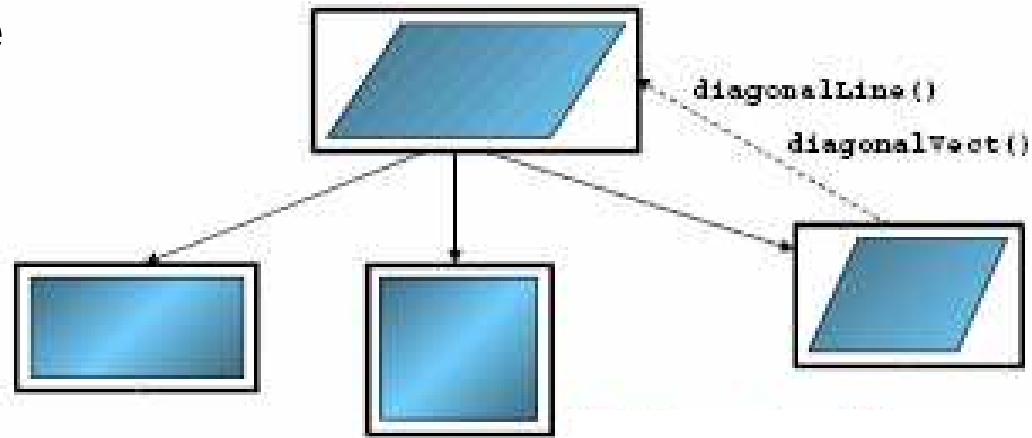
```
// rectangle class declaration  
class Rectangle : virtual public  
    Parallelogram {  
    :  
};
```

```
// rhombus class declaration  
class Rhombus : virtual public  
    Parallelogram {  
    :  
};
```

```
// empty square constructor  
}; Square::Square():  
    Rectangle(1,1),  
    Rhombus(1,90),  
    Parallelogram(1,1,90)  
    {...} [Shapes6.h]
```

# Reassessing Inheritance Hierarchy

- Multiple inheritance is not required for the **Square** class.  
*A square is a kind of parallelogram:*



But how does square inherit functionality of **rectangle** and **rhombus**?

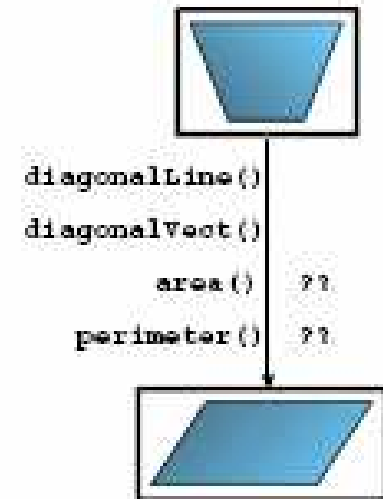
- Consider if the **diagonal()** functions were really unique to the **Rhombus** class.
- The **diagonal()** methods can be moved up inheritance hierarchy so all three derived classes can access them.
- This is known as **percolating functionality upward**.

# The Quadrilateral Class

- A future iteration of the code will make use of several **quadrilaterals**, including the parallelogram.

```
// quadrilateral class declaration
class Quadrilateral {
public:
    Quadrilateral(){}
    virtual ~Quadrilateral(){}
    Line diagonalLine(int corner=1);
    Vect diagonalVect(int corner=1);
    virtual double area() = 0;
    virtual double perimeter() = 0;
protected:
    Point itsLowerLeft;
    Point itsLowerRight;
    Point itsUpperLeft;
    Point itsUpperRight;
}; [Shapes8.h]
```

```
// parallelogram class
class Parallelogram :
    public Quadrilateral {
public:
    Parallelogram();
    :
    virtual ~Parallelogram();
    double base();
    :
    virtual double area();
    virtual double perimeter();
protected:
    double itsAngle;
}; [Shapes8.h]
```



# Abstract Base Classes

- The virtual functions contained in the **Quadrilateral** class had no implementation. These are **pure virtual functions**:

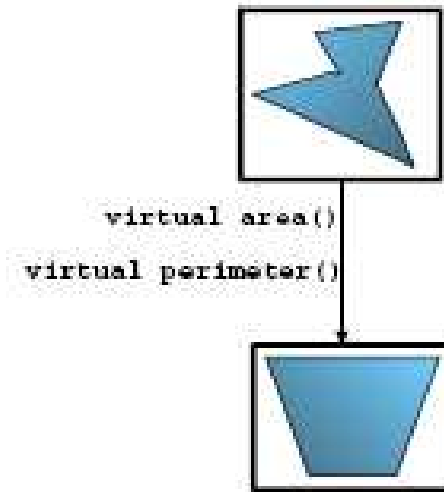
```
virtual double area() = 0;           // must be  
virtual double perimeter() = 0;    // set to zero
```

- A **Quadrilateral** class is included in the program to provide a general definition (**interface**) without providing all the implementation.
- It is expected that a sub-class will implement the methods outlined in the **Quadrilateral** interface (in fact it **must**).
- These classes are known as **Abstract Base Classes**
- Objects cannot be instantiated from ABCs since they would not be fully defined.

# Purely Abstract Base Classes

- A base class that provides no implementation and has no members is known as a **purely abstract base class**.

```
// shape class declaration
class Shape {
public:
    Shape(){}
    virtual ~Shape(){}
    virtual double area() = 0;
    virtual double perimeter() = 0;
};                                [Shapes9.h]
```



- The **Shape** class provides a statement of intent for all “shape-like” classes; Each shape object **must** have access to class methods that return area and perimeter.
- pABCs are often written once the overall design of the classes has been fully established.

# *Has-a-kind-of* Relationships

- C++ distinguishes between *is-a-kind-of* relationships and *has-a-kind-of* relationships:

```
//rectangle declaration
class Rectangle {
:
private:
    Point itsLowerLeft;
:
};
```

```
//rectangle declaration
class Rectangle: private Point {
:
private:
    float itsX; //must redefine Point's data
:                //members as they were private
};
```

Also have **protected** inheritance: public members of base class become private in derived

- The **Rectangle** class makes use of the **Point** class implementation but as it's not a *kind of* Point it doesn't inherit from it. This is called **containment**.
- An alternative to containment is **private inheritance** => public members of base become private to derived class
- Derives implementation of the class but throws away the interface => breaks encapsulation => not in OO style!