

When deriving a class D from a base class B, we may specify that access to the base class is any of the following: `public`, `protected`, `private`.

The base class access specifier controls the access derived types will have to base members and the conversion of a pointer to the derived type to a pointer to the base type.

The most common specification is:

- `public`
- public members of B become public in D
  - protected members of B become private members of D, accessible only to members and friends of D and of classes derived from D
  - private members of B are inaccessible to D
  - any function can convert a D\* to a B\*

Access privileges for the other access specifiers:

- `protected`
- public and protected members of B become protected members of D, accessible only to members and friends of D and of classes derived from D
  - private members of B are inaccessible to D
  - only members and friends of D, and of classes derived from D, function can convert a D\* to a B\*
- `private`
- public and protected members of B become private members of D, accessible only to members and friends of D, but NOT to classes derived from D
  - private members of B are inaccessible to D
  - only members and friends of D can convert a D\* to a B\*

Private inheritance is appropriate when the public interface of the base class is not needed by the user of the derived class, or if it is desirable to hide the public interface of the base class from the user.

Of course, this will also render any protected members of the base class inaccessible to classes derived from the derived class. For that reason private inheritance is used much less often than public inheritance.

Similarly, protected inheritance is appropriate when the public interface of the base class must be hidden from the user of the derived class, but the protected and public interface of the base class is useful in the implementation of classes derived from the derived class.

# Private Inheritance Example

Consider implementing a stack class, given that there is a tested, reliable linked list class available:

```
class LList {
private:
    Node* Head, Curr, Tail;
public:
    LList();
    bool Prefix(const Item& Data); // insert at front of list
    bool Append(const Item& Data); // insert at tail of list
    Item delFirst();                // delete front node
    Item delLast();                 // delete tail node
    . . .
    bool isEmpty() const;
    bool isFull() const;
    ~LList();
};
```

The list class has all the functionality we need, and then some... we need to hide the dangerous parts of the list interface...

# Private Inheritance Example

By deriving the Stack class using private inheritance, we gain the capabilities of the list class but can hide the list interface behind an appropriate interface:

```
class Stack : private List {
public:
    Stack();
    Item Pop();
    bool Push(const Item& Data);
    . . .
    ~Stack();
};
```

```
Item Stack::Pop() {
    return delFirst();
};
```

The Stack class interface just serves as a "front" for the broader List interface, which is entirely hidden from the user.

Classes derived from Stack cannot access the "inappropriate" List interface either... it's completely buried.

A `Polynomial` class could be derived from an instantiation of the queue template `QueueT` seen earlier:

```
class Polynomial : protected QueueT<double> {
private:
    . . .
public:
    Polynomial();
    . . .
    Polynomial& operator+(Polynomial& RHS);
    ~Polynomial();
};
```

The queue is used to hold the coefficients of the polynomial, in order, with zeros stored for missing terms... straightforward and it corresponds nicely to the way most polynomial manipulations work.

We could use `private` inheritance here, but a derived type (such as `QuadraticPolynomial`) would be seriously inconvenienced.

`private` and `protected` inheritance access rules are somewhat complex.

It may be difficult to keep track of all the implications within a large hierarchy.

Aggregation is often, but not always, a more natural way to deal with the problems that motivated using a `private` or `protected` base class.

Sometimes a member function from the base type simply doesn't make sense within the context of a derived type. What do we do?

```
class Rectangle {
private:
    Location NW;
    int      Length, Width;

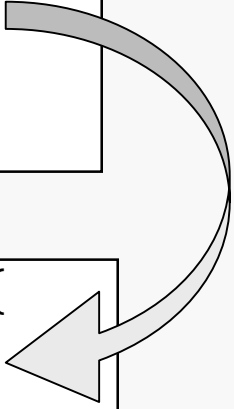
public:
    void ReScale(int Factor)
        {Length = Factor*Length;
         Width  = Factor*Width;}
    void ReSize(int L, int W)
        {Length = L; Width = W;}
    . . .
};
```

We don't want to allow a Square to not have

Length == Width

How to prevent that...?

```
class Square : public Rectangle {
public:
    . . .
};
```



There are three strategies:

1. Override the base member function so it's harmless.
2. Use private inheritance so the base method isn't visible to the user of the derived class.
3. Revise the inheritance hierarchy to make it more appropriate.

Let's look at all three...

## Overriding an Embarrassing Base Method Controlling Inheritance 10

```
void Square::ReSize(int L, int W) {  
    if (L == W) {  
        Length = L;  
        Width  = W;  
    }  
}
```

or

```
void Square::ReSize(int L, int W) {  
    Rectangle::ReSize(L, L);  
}
```

What are the pros and cons for this solution?

# Use Private Inheritance

This will render `Rectangle::ReSize()` invisible to the user who declares an object of type `Square`.

That eliminates any chance the user could incorrectly use the inappropriate base class member function.

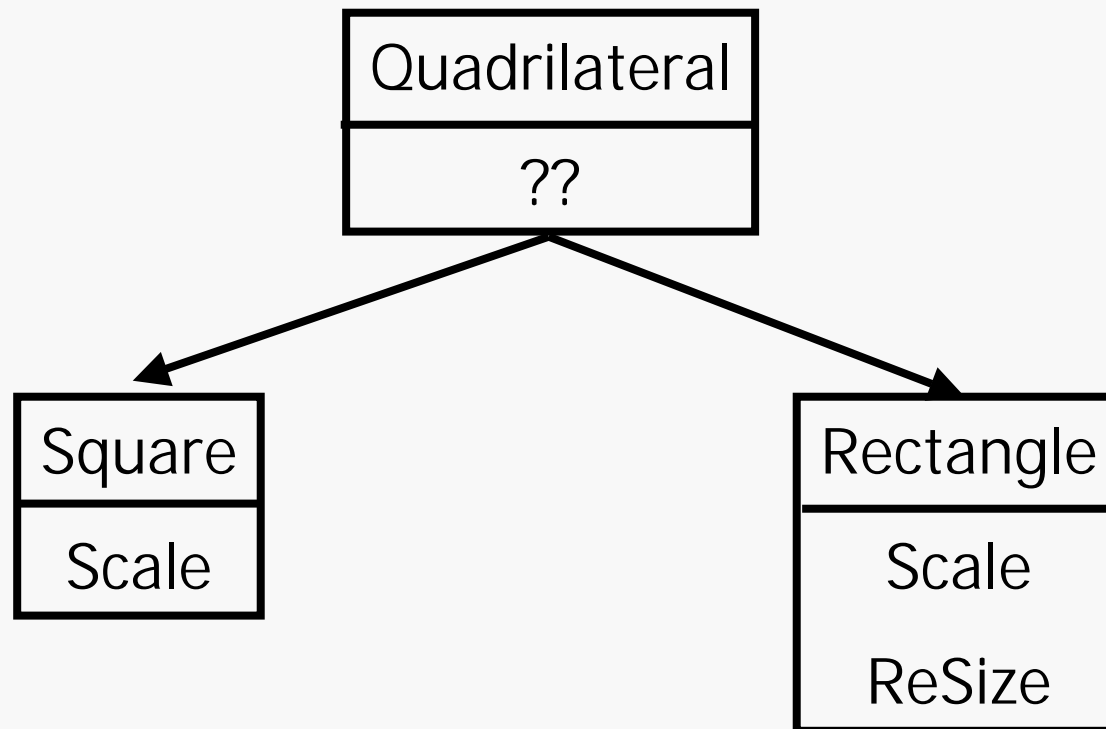
What are the pros and cons for this solution?

```
class Square : Rectangle { // default mode is private
public:
    . . .
};
```

# Revise Inheritance Hierarchy

It doesn't really make sense to say that a square is a rectangle (HS geometry books notwithstanding) ...

However, it DOES make sense to say that squares and rectangles are kinds of quadrilaterals:



If the base class has a member function that the derived class needs to extend or modify, that can be done simply by overriding. This doesn't necessarily indicate a problem with the design of the hierarchy.

If the base class has a member function that the derived class needs to hide from users, that is usually an indication that the base type hasn't been chosen carefully enough.

The "inappropriate base member" problem is often best solved by revising the inheritance hierarchy.

However, that can lead to another problem: cluttermorphism.

Inheritance provides a number of benefits with respect to development:

- reusability of common implementation
- representation of natural logical relationships among types

Inheritance also carries a cost:

- designing modifications to base class require understanding the effect on all derived classes
- designing modifications to derived class requires understanding of the relationship to the base class (not usually too serious)
- modifications to base class will require re-testing implementations of derived classes to verify nothing is broken