Controlling Inheritance

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: protected members of \( B \) become protected members of \( D \), accessible only to members and friends of \( D \) and of classes derived from \( D \)
- private: private members of \( B \) are inaccessible to \( D \)
- any function can convert a \( D^* \) to a \( B^* \)

Inheritance Modes

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 \( D \)
  - only members and friends of \( D \) and of classes derived from \( D \) 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 \( B^* \)

Using Non-Public Inheritance

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.

Protected Inheritance Example

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

The queue is used to hold the coefficients of the polynomial, in order, with zeros stored for missing terms. It 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 inconvenient.

Private Inheritance Example

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

```cpp
class LList {
private:
  Node* Head, Curr, Tail;
public:
  LList();
  Item delFirst(); // delete front node
  Item delLast(); // delete tail node
  ~LList();
};
```

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

Protected Inheritance Example

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

The queue is used to hold the coefficients of the polynomial, in order, with zeros stored for missing terms. It 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 inconvenient.
Controlling Inheritance

Summary

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.

Hiding Inherited Methods

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

Sometimes, a derived class may not want to expose a base class function. For example, a derived class may hide the base class's default constructor.

private:
  base::location dm;
  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...?

In this example, the derived class hides the base class's ReSize() function.

Handling an Embarrassing Base Method

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

What are the pros and cons for this solution?

This solution changes the base class method to meet the requirements of the derived class. It can be difficult to remember which version of a base class method is being overridden.

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?

Use Private Inheritance

The pros of using private inheritance include:

1. It hides the base class's ReSize() function from the derived class.
2. It prevents the user from accidentally using the inappropriate base class member function.

The cons of using private inheritance include:

1. It may be difficult to keep track of all the implications within a large hierarchy.
2. It may be difficult to remember which version of a base class method is being used.

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:

Quadrilateral

??

Square

Scale

Rectangle

Scale

ReSize

Let's look at all three...
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 and Development

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