Controlling Inheritance

Private Inheritance Example
Non-Public Inheritance Usage
Specialization: Replacing Inherited Methods
Solution: Redefine Base Method
Extending Inherited Methods
Modify HourlyEmployee to Extend Print()
Hiding Inherited Methods
Dealing with an Embarrassing Base Method
Overriding an Embarrassing Base Method
Use Private Inheritance
Revise Inheritance Hierarchy
Inheritance Modes

When deriving a class, we may specify the inheritance to be any of:

- **public**: all members of the base class are inherited with the same access protections as they had in the base class.

- **protected**: public and protected members of the base class become protected members of the derived class.*

- **private**: public and protected members of the base class become private members of the derived class.*

* I.e., users of the derived class have NO access to the public interface of the base class.
class HourlyEmployee : private Employee {
private:
   double Rate;
   double Hours;
public:
   HourlyEmployee();
   HourlyEmployee(string FN, string LN, string ID,
                   double R, double H);
   double getRate() const;
   double getHours() const;
   void setRate(double R);
   void setHours(double H);
   ~HourlyEmployee();
};

HourlyEmployee Fred(“Fred”, “Frid”, “10078”, 9.78, 40);

string Ident = Fred.getID(); // illegal

Now a user of the HourlyEmployee class cannot directly call the inherited Employee member function getID().
class HourlyEmployee : private Employee {
private:
    double Rate;
    double Hours;
public:
    HourlyEmployee();
    HourlyEmployee(string FN, string LN, string ID,
                    double R, double H);
    double getRate() const;
    double getHours() const;
    void setRate(double R);
    void setHours(double H);
    string getID() const;
    string getName() const;
    ~HourlyEmployee();
};

string HourlyEmployee::getID() const {
    return ID;
}

Providing a public HourlyEmployee member function restores access.
Non-Public Inheritance Usage

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 in 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 the derived class.
Problem:
You have a base class.
You’re writing a derived class to provide a specialization.
You don’t like the implementation of a member function in the base class!
You want to redefine the member function.

```cpp
class Employee {
private:
    string FName;
    string LName;
    string ID;

g public:
    . . .
    void setID(string Ident);
    . . .
};

void Employee::setID(string Ident) {
    ID = Ident;
}
```

However: the ID of an hourly employee must begin with a character signifying the employee’s pay rate category (say: A, B, C, . . .).
In the derived class, provide an appropriate implementation, using the same interface. That will **override** the base class version when invoked on an object of the derived type:

```cpp
void HourlyEmployee::setID(string Ident) {
    string IDprefix = getPrefix(Rate);
    ID = IDprefix + Ident;
}
```

The appropriate member function implementation is chosen (at compile time), based upon the type of the invoking object and the inheritance hierarchy. Beginning with the derived class, the hierarchy is searched upward until a matching function definition is found:

```cpp
HourlyEmployee Fred(“Fred”, “Frid”, “10078”, 9.78, 40);
Fred.setID(“214801”);
```
Problem:
You have a base class.
You’re writing a derived class.
A member function in the base class is “not quite” OK.
You want to extend the base member function.

```c++
class Employee {
private:
    string FName;
    string LName;
    string ID;
public:
    ... 
    void Print(ostream& Out); 
    ... 
};
```

void Employee:: Print(ostream& Out) {
    Out << LName << " , " << FName 
    << 't' << ID << endl;
}

However: we’re adding a Department field for hourly and salaried employees, and we want to print that on the next line…
Modify HourlyEmployee to Extend Print()

In the derived class, provide an appropriate implementation, using the same interface. However, now we still want to use the base member function (to avoid code duplication), just add functionality:

```cpp
void HourlyEmployee::Print(ostream& Out) {
    Employee::Print(Out);
    Out << Department << endl;
}
```

Within an inheritance hierarchy, we can invoke a member function of a base type by specifying the name of that type via the scope resolution operator.
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?

class Rectangle {
protected:
    Location NW;
    double Length, Width;

public:
    ...  
    void ReScale(double Factor);
    void ReSize(double L, double 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:
    ...  
};
Dealing with 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

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

or

void Square::ReSize(double L, double 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:

```
Quadrilateral
  ??
  Square
    Scale
  Rectangle
    Scale
    ReSize
```