

## More on inheritance...

We know that a derived class D should inherit from a base class B only if D “isa” B. What about the following case:

```
class Bird {
public:
    virtual void fly(); // birds can fly
    ...
};
```

```
class Penguin : public Bird {
public:
    ...
};
```

**But Penguins can't fly! How do we fix this? Well, you could override the fly() method to return a run-time error, and abort....**

```
class Penguin : public Bird {
public:
    virtual void fly() { assert(FALSE); }
    ...
};
```

```
Penguin p;
...
p.fly(); // program aborts!
```

When using inheritance, there is a tendency for code and data to move to the top class in the inheritance hierarchy. Sometimes, it is necessary to introduce some additional abstractions in the inheritance structure that allow you to partition data and methods in a more logical manner.

So, you could add another layer of abstraction....

```
class Bird {
public:
    .... // omit the fly() method
};

class FlyingBird : public Bird {
public:
    virtual void fly();
    ...
};

class NonFlyingBird : public Bird { ... };

class Penguin : public NonFlyingBird { ... };
```

The advantage is that the compiler can now do static type checking to ensure that a penguin doesn't fly.

```
Penguin p;
...
p.fly(); // compiler flags this as an error
```

The lesson to take from this is that it is usually better to reorganize your classes and introduce some additional abstraction that facilitate compile-time checking of the way someone utilizes a set of classes.

## Base Class Access Specifiers

```
class A {  
public:  
    int f();  
};
```

```
class B : public A {  
public:  
    // A::f() is visible to clients of class B  
};
```

```
class C : private A {  
public:  
    // A::f() is NOT visible to clients of class C  
};
```

```
class D : A { // no access specifier using a class implies private inheritance by default  
public:  
    // A::f() is NOT visible to clients of class D  
};
```

```
struct E : A { // no access specifier using a struct implies public inheritance by default  
  
    // A::f() is visible to clients of class E  
};
```

```
class D : public A, B, C { ... }; // D inherits A public, B & C private.  
class D : public A, private B, public C { ... };
```

**In retrospect, the language designers probably should have required an explicit specifier. Most good compilers will warn you that you are inheriting private if you do not provide an explicit access specifier.**

## Access Declarations for Functions

A mechanism to allow you to selectively “open up” some methods from a class that is privately inherited.

```
class X {
public:
    void f();
    void f(int);
    void g();
};

class Y : private X {
public:
    X::f;    // makes X::f() and X::f(int) accessible, X::g() remains hidden
};
```

**You cannot adjust access to an overloaded function name that is defined at two different access levels**

```
class X {
protected:
    void f();
public:
    void f(int);
    void g();
};

class Y : private X {
public:
    X::f;    // ERROR - X::f() protected and X::f(int) public
};
```

## Access Declarations for Functions

```
class X {
public:
    void f();
};

class Y : private X {
public:
    void f(int);
    X::f;    // ERROR: two declarations for f
};
```

Instead, you would have to write the following:

```
class Y : private X {
public:
    void f(int);
    void f() { X::f(); }
};
```

Note that you can not use access declarations to promote a protected function inherited from a base class to be a public function in the derived class. That is, if `X::f()` above were in a protected section, it would be an error to write `X::f` in the public section of `Y`. Likewise, you cannot reduce access. That is, you cannot demote `X::f()` by placing an access declaration for `X::f` in the protected section of class `Y`.

## Friend-ness is neither transitive nor inherited

C a friend of B and B a friend of A does NOT imply that C is a friend of A

```
class A {
private:
    friend class B;
    int a;
};

class B {
private:
    friend class C;
};

class C {
public:

    void f(A* p) {p->a++;}    // ERROR - cannot access private A::a
};
```

D a subclass of B and B a friend of A does NOT imply that D is a friend of class A

```
class D : public B {
public:
    void f(A* p){p->a++;}    // ERROR - cannot access private A::a
};
```

**Would it have been better to allow friend access to be inherited?**

A class with private data accessible by a trusted friend class

```
class BankAccount {
private:
    friend class Teller;
    double balance;
    ...
};
```

A trusted friend class can access private data of a secure class

```
class BankTeller {
public:

    void traced_deposit(double d) {
        /* trace deposit transaction*/
        ...;
        balance += d;
    }
    ...
};
```

Someone can then define an untrusted subclass of the trusted class and mess up the private data!

```
class Hacker : public BankTeller {
public:
    void untraced_deposit(double d) { balance += d; }
};
```

## Back Door Access to Private Virtual Functions

Access to a virtual function is determined by its declaration and access is not affected by the access rules of a function that later overrides it.

```
class B {
public:
    virtual void f() {...}
};

class D : public B {
private:
    virtual void f() { ... } // override B::f()
};

void g()
{
    D p;
    B* b = &p;
    D* d = &p;

    b->f(); // calls B::f() which is public, but private D::f() is invoked
           // newer ANSI C++ compilers may flag this as an error

    d->f(); // COMPILE-TIME ERROR - D::f() is private
}
```

Some C++ compilers try to detect and restrict this kind of “back door” access, but many do not. Similarly, if you inherit private from a class, some compilers will not let you initialize a base class pointer to a derived class object. I.e., if D inherits private from B, then it is not possible to write:

```
B* b = new D(); // if D inherits private from B, some compilers will not permit this
```