24.5 — Inheritance and access specifiers

ALEX SEPTEMBER 11, 2023

In the previous lessons in this chapter, you've learned a bit about how base inheritance works. In all of our examples so far, we've used public inheritance. That is, our derived class publicly inherits the base class.

In this lesson, we'll take a closer look at public inheritance, as well as the two other kinds of inheritance (private and protected). We'll also explore how the different kinds of inheritance interact with access specifiers to allow or restrict access to members.

To this point, you've seen the private and public access specifiers, which determine who can access the members of a class. As a quick refresher, public members can be accessed by anybody. Private members can only be accessed by member functions of the same class or friends. This means derived classes can not access private members of the base class directly!

```
class Base
{
private:
    int m_private {}; // can only be accessed by Base members and friends (not derived classes)
public:
    int m_public {}; // can be accessed by anybody
};
```

This is pretty straightforward, and you should be quite used to it by now.

The protected access specifier

When dealing with inherited classes, things get a bit more complex.

C++ has a third access specifier that we have yet to talk about because it's only useful in an inheritance context. The **protected** access specifier allows the class the member belongs to, friends, and derived classes to access the member. However, protected members are not accessible from outside the class.

```
class Base
{
public:
    int m_public {}; // can be accessed by anybody
protected:
    int m_protected {}; // can be accessed by Base members, friends, and derived classes
private:
    int m_private {}; // can only be accessed by Base members and friends (but not derived classes)
};

class Derived: public Base
{
public:
    Derived()
    {
        m_public = 1; // allowed: can access public base members from derived class
        m_protected = 2; // allowed: can access protected base members from derived class
        m_private = 3; // not allowed: can not access private base members from derived class
    }
};

int main()
{
    Base base;
    base.m_public = 1; // allowed: can access public members from outside class
    base.m_protected = 2; // not allowed: can not access protected members from outside class
    base.m_private = 3; // not allowed: can not access private members from outside class

    return 0;
}
```

In the above example, you can see that the protected base member m_protected is directly accessible by the derived class, but not by the public.

So when should I use the protected access specifier?

With a protected attribute in a base class, derived classes can access that member directly. This means that if you later change anything about that protected attribute (the type, what the value means, etc...), you'll probably need to change both the base class AND all of the derived classes.

Therefore, using the protected access specifier is most useful when you (or your team) are going to be the ones deriving from your own classes, and the number of derived classes is reasonable. That way, if you make a change to the implementation of the base class, and updates to the derived classes are necessary as a result, you can make the updates yourself (and have it not take forever, since the number of derived classes is limited).

Making your members private means the public and derived classes can't directly make changes to the base class. This is good for insulating the public or derived classes from implementation changes, and for ensuring invariants are maintained properly. However, it also means your class may need a larger public (or protected) interface to support all of the functions that the public or derived classes need for operation, which has its own cost to build, test, and maintain.

In general, it's better to make your members private if you can, and only use protected when derived classes are planned and the cost to build and maintain an interface to those private members is too high.

> **Best practice**
>
> Favor private members over protected members.

Different kinds of inheritance, and their impact on access

First, there are three different ways for classes to inherit from other classes: public, protected, and private.

To do so, simply specify which type of access you want when choosing the class to inherit from:

```
// Inherit from Base publicly
class Pub: public Base
{
};

// Inherit from Base protectedly
class Pro: protected Base
{
};

// Inherit from Base privately
class Pri: private Base
{
};

class Def: Base // Defaults to private inheritance
{
};
```

If you do not choose an inheritance type, C++ defaults to private inheritance (just like members default to private access if you do not specify otherwise).

That gives us 9 combinations: 3 member access specifiers (public, private, and protected), and 3 inheritance types (public, private, and protected).

So what's the difference between these? In a nutshell, when members are inherited, the access specifier for an inherited member may be changed (in the derived class only) depending on the type of inheritance used. Put another way, members that were public or protected in the base class may change access specifiers in the derived class.

This might seem a little confusing, but it's not that bad. We'll spend the rest of this lesson exploring this in detail.

Keep in mind the following rules as we step through the examples:

- A class can always access its own (non-inherited) members.
- The public accesses the members of a class based on the access specifiers of the class it is accessing.
- A derived class accesses inherited members based on the access specifier inherited from the parent class. This varies depending on the access specifier and type of inheritance used.

Public inheritance

Public inheritance is by far the most commonly used type of inheritance. In fact, very rarely will you see or use the other types of inheritance, so your primary focus should be on understanding this section. Fortunately, public inheritance is also the easiest to understand. When you inherit a base class publicly, inherited public members stay public, and inherited protected members stay protected. Inherited private members, which

were inaccessible because they were private in the base class, stay inaccessible.

Access specifier in base class	Access specifier when inherited publicly
Public	Public
Protected	Protected
Private	Inaccessible

Here's an example showing how things work:

```
class Base
{
public:
    int m_public {};
protected:
    int m_protected {};
private:
    int m_private {};
};

class Pub: public Base // note: public inheritance
{
    // Public inheritance means:
    // Public inherited members stay public (so m_public is treated as public)
    // Protected inherited members stay protected (so m_protected is treated as protected)
    // Private inherited members stay inaccessible (so m_private is inaccessible)
public:
    Pub()
    {
        m_public = 1; // okay: m_public was inherited as public
        m_protected = 2; // okay: m_protected was inherited as protected
        m_private = 3; // not okay: m_private is inaccessible from derived class
    }
};

int main()
{
    // Outside access uses the access specifiers of the class being accessed.
    Base base;
    base.m_public = 1; // okay: m_public is public in Base
    base.m_protected = 2; // not okay: m_protected is protected in Base
    base.m_private = 3; // not okay: m_private is private in Base

    Pub pub;
    pub.m_public = 1; // okay: m_public is public in Pub
    pub.m_protected = 2; // not okay: m_protected is protected in Pub
    pub.m_private = 3; // not okay: m_private is inaccessible in Pub

    return 0;
}
```

This is the same as the example above where we introduced the protected access specifier, except that we've instantiated the derived class as well, just to show that with public inheritance, things work identically in the base and derived class.

Public inheritance is what you should be using unless you have a specific reason not to.

Best practice

Use public inheritance unless you have a specific reason to do otherwise.

Protected inheritance

Protected inheritance is the least common method of inheritance. It is almost never used, except in very particular cases. With protected inheritance, the public and protected members become protected, and private members stay inaccessible.

Because this form of inheritance is so rare, we'll skip the example and just summarize with a table:

Access specifier in base class	Access specifier when inherited protectedly
Public	Protected
Protected	Protected
Private	Inaccessible

Private inheritance

With private inheritance, all members from the base class are inherited as private. This means private members are inaccessible, and protected and public members become private.

Note that this does not affect the way that the derived class accesses members inherited from its parent! It only affects the code trying to access those members through the derived class.

```
class Base
{
public:
    int m_public {};
protected:
    int m_protected {};
private:
    int m_private {};
};

class Pri: private Base // note: private inheritance
{
    // Private inheritance means:
    // Public inherited members become private (so m_public is treated as private)
    // Protected inherited members become private (so m_protected is treated as private)
    // Private inherited members stay inaccessible (so m_private is inaccessible)
public:
    Pri()
    {
        m_public = 1; // okay: m_public is now private in Pri
        m_protected = 2; // okay: m_protected is now private in Pri
        m_private = 3; // not okay: derived classes can't access private members in the base class
    }
};

int main()
{
    // Outside access uses the access specifiers of the class being accessed.
    // In this case, the access specifiers of base.
    Base base;
    base.m_public = 1; // okay: m_public is public in Base
    base.m_protected = 2; // not okay: m_protected is protected in Base
    base.m_private = 3; // not okay: m_private is private in Base

    Pri pri;
    pri.m_public = 1; // not okay: m_public is now private in Pri
    pri.m_protected = 2; // not okay: m_protected is now private in Pri
    pri.m_private = 3; // not okay: m_private is inaccessible in Pri

    return 0;
}
```

To summarize in table form:

Access specifier in base class	Access specifier when inherited privately
Public	Private
Protected	Private
Private	Inaccessible

Private inheritance can be useful when the derived class has no obvious relationship to the base class, but uses the base class for implementation internally. In such a case, we probably don't want the public interface of the base class to be exposed through objects of the derived class (as it would be if we inherited publicly).

In practice, private inheritance is rarely used.

A final example

```
class Base
{
public:
    int m_public {};
protected:
    int m_protected {};
private:
    int m_private {};
};
```

Base can access its own members without restriction. The public can only access m_public. Derived classes can access m_public and m_protected.

```
class D2 : private Base // note: private inheritance
{
    // Private inheritance means:
    // Public inherited members become private
    // Protected inherited members become private
    // Private inherited members stay inaccessible
public:
    int m_public2 {};
protected:
    int m_protected2 {};
private:
    int m_private2 {};
};
```

D2 can access its own members without restriction. D2 can access Base's m_public and m_protected members, but not m_private. Because D2 inherited Base privately, m_public and m_protected are now considered private when accessed through D2. This means the public can not access these variables when using a D2 object, nor can any classes derived from D2.

```
class D3 : public D2
{
    // Public inheritance means:
    // Public inherited members stay public
    // Protected inherited members stay protected
    // Private inherited members stay inaccessible
public:
    int m_public3 {};
protected:
    int m_protected3 {};
private:
    int m_private3 {};
};
```

D3 can access its own members without restriction. D3 can access D2's m_public2 and m_protected2 members, but not m_private2. Because D3 inherited D2 publicly, m_public2 and m_protected2 keep their access specifiers when accessed through D3. D3 has no access to Base's m_private, which was already private in Base. Nor does it have access to Base's m_protected or m_public, both of which became private when D2 inherited them.

Summary

The way that the access specifiers, inheritance types, and derived classes interact causes a lot of confusion. To try and clarify things as much as possible:

First, a class (and friends) can always access its own non-inherited members. The access specifiers only affect whether outsiders and derived classes can access those members.

Second, when derived classes inherit members, those members may change access specifiers in the derived class. This does not affect the derived classes' own (non-inherited) members (which have their own access specifiers). It only affects whether outsiders and classes derived from the derived class can access those inherited members.

Here's a table of all of the access specifier and inheritance types combinations:

Access specifier in base class	Access specifier when inherited publicly	Access specifier when inherited privately	Access specifier when inherited protectedly
Public	Public	Private	Protected
Protected	Protected	Private	Protected
Private	Inaccessible	Inaccessible	Inaccessible

As a final note, although in the examples above, we've only shown examples using member variables, these access rules hold true for all members (e.g. member functions and types declared inside the class).

Next lesson
24.6 Adding new functionality to a derived class

Back to table of contents

Previous lesson
24.4 Constructors and initialization of derived classes