24.8 — Hiding inherited functionality

Changing an inherited member's access level

C++ gives us the ability to change an inherited member's access specifier in the derived class. This is done by using a *using declaration* to identify the (scoped) base class member that is having its access changed in the derived class, under the new access specifier.

For example, consider the following Base:

```
#include <iostream>

class Base
{
private:
    int m_value {};

public:
    Base(int value)
        : m_value { value }
    {
    }

protected:
    void printValue() const { std::cout << m_value; }
};
```

Because Base::printValue() has been declared as protected, it can only be called by Base or its derived classes. The public can not access it.

Let's define a Derived class that changes the access specifier of printValue() to public:

```
class Derived: public Base
{
public:
    Derived(int value)
        : Base { value }
    {
    }

    // Base::printValue was inherited as protected, so the public has no access
    // But we're changing it to public via a using declaration
    using Base::printValue; // note: no parenthesis here
};
```

This means that this code will now work:

```
int main()
{
    Derived derived { 7 };

    // printValue is public in Derived, so this is okay
    derived.printValue(); // prints 7
    return 0;
}
```

You can only change the access specifiers of base members the derived class would normally be able to access. Therefore, you can never change the access specifier of a base member from private to protected or public, because derived classes do not have access to private members of the base class.

Hiding functionality

In C++, it is not possible to remove or restrict functionality from a base class other than by modifying the source code. However, in a derived class, it is possible to hide functionality that exists in the base class, so that it can not be accessed through the derived class. This can be done simply by changing the relevant access specifier.

For example, we can make a public member private:

```
#include <iostream>

class Base
{
public:
        int m_value{};
};

class Derived : public Base
{
private:
        using Base::m_value;

public:
        Derived(int value) : Base { value }
        {
        }
};

int main()
{
        Derived derived{ 7 };
        std::cout << derived.m_value; // error: m_value is private in Derived

        Base& base{ derived };
        std::cout << base.m_value; // okay: m_value is public in Base

        return 0;
}
```

This allowed us to take a poorly designed base class and encapsulate its data in our derived class. Alternatively, instead of inheriting Base's members publicly and making m_value private by overriding its access specifier, we could have inherited Base privately, which would have caused all of Base's member to be inherited privately in the first place.

However, it is worth noting that while m_value is private in the Derived class, it is still public in the Base class. Therefore the encapsulation of m_value in Derived can still be subverted by casting to Base& and directly accessing the member.

Perhaps surprisingly, given a set of overloaded functions in the base class, there is no way to change the access specifier for a single overload. You can only change them all:

```
#include <iostream>

class Base
{
public:
    int m_value{};

    int getValue() const { return m_value; }
    int getValue(int) const { return m_value; }
};

class Derived : public Base
{
private:
        using Base::getValue; // make ALL getValue functions private

public:
        Derived(int value) : Base { value }
        {
        }
};

int main()
{
        Derived derived{ 7 };
        std::cout << derived.getValue();  // error: getValue() is private in Derived
        std::cout << derived.getValue(5); // error: getValue(int) is private in
Derived

        return 0;
}
```

Deleting functions in the derived class

You can also mark member functions as deleted in the derived class, which ensures they can’t be called at all through a derived object:

```
#include <iostream>
class Base
{
private:
	int m_value {};

public:
	Base(int value)
		: m_value { value }
	{
	}

	int getValue() const { return m_value; }
};

class Derived : public Base
{
public:
	Derived(int value)
		: Base { value }
	{
	}


	int getValue() const = delete; // mark this function as inaccessible
};

int main()
{
	Derived derived { 7 };

	// The following won't work because getValue() has been deleted!
	std::cout << derived.getValue();

	return 0;
}
```

In the above example, we've marked the getValue() function as deleted. This means that the compiler will complain when we try to call the derived version of the function. Note that the Base version of getValue() is still accessible though. We can call Base::getValue() in one of two ways:

```
int main()
{
	Derived derived { 7 };

	// We can call the Base::getValue() function directly
	std::cout << derived.Base::getValue();

	// Or we can upcast Derived to a Base reference and getValue() will resolve
to Base::getValue()
	std::cout << static_cast<Base&>(derived).getValue();

	return 0;
}
```

If using the casting method, we cast to a Base& rather than a Base to avoid making a copy of the Base portion of `derived`.