25.9 — Object slicing

Let’s go back to an example we looked at previously:

```
#include <iostream>
#include <string_view>

class Base
{
protected:
    int m_value{};

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

    virtual ~Base() = default;

    virtual std::string_view getName() const { return "Base"; }
    int getValue() const { return m_value; }
};

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

   std::string_view getName() const override { return "Derived"; }
};

int main()
{
    Derived derived{ 5 };
    std::cout << "derived is a " << derived.getName() << " and has value " <<
derived.getValue() << '\n';

    Base& ref{ derived };
    std::cout << "ref is a " << ref.getName() << " and has value " << ref.getValue()
<< '\n';

    Base* ptr{ &derived };
    std::cout << "ptr is a " << ptr->getName() << " and has value " << ptr-
>getValue() << '\n';

    return 0;
}
```

In the above example, ref references and ptr points to derived, which has a Base part, and a Derived part. Because ref and ptr are of type Base, ref and ptr can only see the Base part of derived -- the Derived part of derived still exists, but simply can’t be seen through ref or ptr.

However, through use of virtual functions, we can access the most-derived version of a function. Consequently, the above program prints:

```
derived is a Derived and has value 5
ref is a Derived and has value 5
ptr is a Derived and has value 5
```

But what happens if instead of setting a Base reference or pointer to a Derived object, we simply *assign* a Derived object to a Base object?

```
int main()
{
    Derived derived{ 5 };
    Base base{ derived }; // what happens here?
    std::cout << "base is a " << base.getName() << " and has value " <<
base.getValue() << '\n';

    return 0;
}
```

Remember that derived has a Base part and a Derived part. When we assign a Derived object to a Base object, only the Base portion of the Derived object is copied. The Derived portion is not. In the example above, base receives a copy of the Base portion of derived, but not the Derived portion. That Derived portion has effectively been “sliced off”. Consequently, the assigning of a Derived class object to a Base class object is called **object slicing** (or slicing for short).

Because base was and still is just a Base, Base’s virtual pointer still points to Base. Thus, base.getName() resolves to Base::getName().

The above example prints:

```
base is a Base and has value 5
```

Used conscientiously, slicing can be benign. However, used improperly, slicing can cause unexpected results in quite a few different ways. Let’s examine some of those cases.

Slicing and functions

Now, you might think the above example is a bit silly. After all, why would you assign derived to base like that? You probably wouldn’t. However, slicing is much more likely to occur accidentally with functions.

Consider the following function:

```
void printName(const Base base) // note: base passed by value, not reference
{
    std::cout << "I am a " << base.getName() << '\n';
}
```

This is a pretty simple function with a const base object parameter that is passed by value. If we call this function like such:

```
int main()
{
    Derived d{ 5 };
    printName(d); // oops, didn't realize this was pass by value on the calling end

    return 0;
}
```

When you wrote this program, you may not have noticed that base is a value parameter, not a reference. Therefore, when called as printName(d), while we might have expected base.getName() to call virtualized function getName() and print “I am a Derived”, that is not what happens. Instead, Derived object d is sliced and only the Base portion is copied into the base parameter. When base.getName() executes, even though the getName() function is virtualized, there’s no Derived portion of the class for it to resolve to. Consequently, this program prints:

```
I am a Base
```

In this case, it’s pretty obvious what happened, but if your functions don’t actually print any identifying information like this, tracking down the error can be challenging.

Of course, slicing here can all be easily avoided by making the function parameter a reference instead of a pass by value (yet another reason why passing classes by reference instead of value is a good idea).

```
void printName(const Base& base) // note: base now passed by reference
{
    std::cout << "I am a " << base.getName() << '\n';
}

int main()
{
    Derived d{ 5 };
    printName(d);

    return 0;
}
```

This prints:

```
I am a Derived
```

Slicing vectors

Yet another area where new programmers run into trouble with slicing is trying to implement polymorphism with std::vector. Consider the following program:

```
#include <vector>

int main()
{
	std::vector<Base> v{};
	v.push_back(Base{ 5 });    // add a Base object to our vector
	v.push_back(Derived{ 6 }); // add a Derived object to our vector

	// Print out all of the elements in our vector
	for (const auto& element : v)
		std::cout << "I am a " << element.getName() << " with value " << element.getValue() << '\n';

	return 0;
}
```

This program compiles just fine. But when run, it prints:

```
I am a Base with value 5
I am a Base with value 6
```

Similar to the previous examples, because the std::vector was declared to be a vector of type Base, when Derived(6) was added to the vector, it was sliced.

Fixing this is a little more difficult. Many new programmers try creating a std::vector of references to an object, like this:

```
std::vector<Base&> v{};
```

Unfortunately, this won't compile. The elements of std::vector must be assignable, whereas references can't be reassigned (only initialized).

One way to address this is to make a vector of pointers:

```
#include <iostream>
#include <vector>

int main()
{
	std::vector<Base*> v{};

	Base b{ 5 }; // b and d can't be anonymous objects
	Derived d{ 6 };

	v.push_back(&b); // add a Base object to our vector
	v.push_back(&d); // add a Derived object to our vector

	// Print out all of the elements in our vector
	for (const auto* element : v)
		std::cout << "I am a " << element->getName() << " with value " <<
element->getValue() << '\n';

	return 0;
}
```

This prints:

```
I am a Base with value 5
I am a Derived with value 6
```

which works! A few comments about this. First, nullptr is now a valid option, which may or may not be desirable. Second, you now have to deal with pointer semantics, which can be awkward. But the upside is that using pointers allows us to put dynamically allocated objects in the vector (just don’t forget to explicitly delete them).

Another option is to use std::reference_wrapper, which is a class that mimics an reassignable reference:

```
#include <functional> // for std::reference_wrapper
#include <iostream>
#include <string_view>
#include <vector>

class Base
{
protected:
    int m_value{};

public:
    Base(int value)
        : m_value{ value }
    {
    }
    virtual ~Base() = default;

    virtual std::string_view getName() const { return "Base"; }
    int getValue() const { return m_value; }
};

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

    std::string_view getName() const override { return "Derived"; }
};

int main()
{
        std::vector<std::reference_wrapper<Base>> v{}; // a vector of reassignable
references to Base

        Base b{ 5 }; // b and d can't be anonymous objects
        Derived d{ 6 };

        v.push_back(b); // add a Base object to our vector
        v.push_back(d); // add a Derived object to our vector

        // Print out all of the elements in our vector
        // we use .get() to get our element out of the std::reference_wrapper
        for (const auto& element : v) // element has type const
std::reference_wrapper<Base>&
                std::cout << "I am a " << element.get().getName() << " with value "
<< element.get().getValue() << '\n';

        return 0;
}
```

The Frankenobject

In the above examples, we've seen cases where slicing lead to the wrong result because the derived class had been sliced off. Now let's take a look at another dangerous case where the derived object still exists!

Consider the following code:

```
int main()
{
    Derived d1{ 5 };
    Derived d2{ 6 };
    Base& b{ d2 };

    b = d1; // this line is problematic

    return 0;
}
```

The first three lines in the function are pretty straightforward. Create two Derived objects, and set a Base reference to the second one.

The fourth line is where things go astray. Since b points at d2, and we're assigning d1 to b, you might think that the result would be that d1 would get copied into d2 -- and it would, if b were a Derived. But b is a Base, and the operator= that C++ provides for classes isn't virtual by default. Consequently, the assignment operator that copies a Base is invoked, and only the Base portion of d1 is copied into d2.

As a result, you'll discover that d2 now has the Base portion of d1 and the Derived portion of d2. In this particular example, that's not a problem (because the Derived class has no data of its own), but in most cases, you'll have just created a Frankenobject -- composed of parts of multiple objects. Worse, there's no easy way to prevent this from happening (other than avoiding assignments like this as much as possible).

Conclusion

Although C++ supports assigning derived objects to base objects via object slicing, in general, this is likely to cause nothing but headaches, and you should generally try to avoid slicing. Make sure your function parameters are references (or pointers) and try to avoid any kind of pass-by-value when it comes to derived classes.