12.10 — Printing inherited classes using operator<<

BY ALEX ON NOVEMBER 23RD, 2016 | LAST MODIFIED BY ALEX ON JANUARY 23RD, 2020

Consider the following program that makes use of a virtual function:

```
class Base
{
public:
    Base() {}

    virtual void print() const { std::cout << "Base";  }
};

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

    virtual void print() const override { std::cout << "Derived"; }
};

int main()
{
    Derived d;
    Base &b = d;
    b.print(); // will call Derived::print()

    return 0;
}
```

By now, you should be comfortable with the fact that b.print() will call Derived::print() (because b is pointing to a Derived class object, Base::print() is a virtual function, and Derived::print() is an override).

While calling member functions like this to do output is okay, this style of function doesn't mix well with std::cout:

```
#include <iostream>
int main()
{
    Derived d;
    Base &b = d;

        std::cout << "b is a ";
        b.print(); // messy, we have to break our print statement to call this function
        std::cout << '\n';

    return 0;
}
```

In this lesson, we'll look at how to override operator<< for classes using inheritance, so that we can use operator<< as expected, like this:

```
std::cout << "b is a " << b << '\n'; // much better
```

The challenges with operator<<

Let's start by overloading operator<< in the typical way:

```
#include <iostream>
class Base
```

```
{
public:
    Base() {}

    virtual void print() const { std::cout << "Base";  }

    friend std::ostream& operator<<(std::ostream &out, const Base &b)
        {
            out << "Base";
            return out;
        }
};

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

    virtual void print() const override { std::cout << "Derived"; }

    friend std::ostream& operator<<(std::ostream &out, const Derived &d)
        {
            out << "Derived";
            return out;
        }

};

int main()
{
    Base b;
    std::cout << b << '\n';

    Derived d;
    std::cout << d << '\n';

    return 0;
}
```

Because there is no need for virtual function resolution here, this program works as we'd expect, and prints:

```
Base
Derived
```

Now, consider the following main() function instead:

```
int main()
{
    Derived d;
    Base &bref = d;
    std::cout << bref << '\n';

    return 0;
}
```

This program prints:

```
Base
```

That's probably not what we were expecting. This happens because our version of operator<< that handles Base objects isn't virtual, so std::cout << bref calls the version of operator<< that handles Base objects rather than Derived objects.

Therein lies the challenge.

Can we make Operator << virtual?

If this issue is that operator<< isn't virtual, can't we simply make it virtual?

The short answer is no. There are a number of reasons for this.

First, only member functions can be virtualized -- this makes sense, since only classes can inherit from other classes, and there's no way to override a function that lives outside of a class (you can overload non-member functions, but not override them). Because we typically implement operator<< as a friend, and friends aren't considered member functions, a friend version of operator<< is ineligible to be virtualized. (For a review of why we implement operator<< this way, please revisit lesson **9.4 -- Overloading operators using member functions**).

Second, even if we could virtualize operator<< there's the problem that the function parameters for Base::operator<< and Derived::operator<< differ (the Base version would take a Base parameter and the Derived version would take a Derived parameter). Consequently, the Derived version wouldn't be considered an override of the Base version, and thus be ineligible for virtual function resolution.

So what's a programmer to do?

The solution

The answer, as it turns out, is surprisingly simple.

First, we set up operator<< as a friend in our base class as usual. But instead of having operator<< do the printing itself, we delegate that responsibility to a normal member function that *can* be virtualized!

Here's the full solution that works:

```
#include <iostream>
class Base
{
public:
    Base() {}

    // Here's our overloaded operator<<
    friend std::ostream& operator<<(std::ostream &out, const Base &b)
    {
        // Delegate printing responsibility for printing to member function print()
        return b.print(out);
    }

    // We'll rely on member function print() to do the actual printing
    // Because print is a normal member function, it can be virtualized
    virtual std::ostream& print(std::ostream& out) const
    {
        out << "Base";
        return out;
    }
};

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

```

```cpp
    // Here's our override print function to handle the Derived case
    virtual std::ostream& print(std::ostream& out) const override
    {
        out << "Derived";
        return out;
    }
};

int main()
{
    Base b;
    std::cout << b << '\n';

    Derived d;
    std::cout << d << '\n'; // note that this works even with no operator<< that explicitly handles Derived objects

    Base &bref = d;
    std::cout << bref << '\n';

    return 0;
}
```

The above program works in all three cases:

```
Base
Derived
Derived
```

Let’s examine how in more detail.

First, in the Base case, we call operator<<, which calls virtual function print(). Since our Base reference parameter points to a Base object, b.print() resolves to Base::print(), which does the printing. Nothing too special here.

In the Derived case, the compiler first looks to see if there’s an operator<< that takes a Derived object. There isn’t one, because we didn’t define one. Next the compiler looks to see if there’s an operator<< that takes a Base object. There is, so the compiler does an implicit upcast of our Derived object to a Base& and calls the function (we could have done this upcast ourselves, but the compiler is helpful in this regard). This function then calls virtual print(), which resolves to Derived::print().

Note that we don’t need to define an operator<< for each derived class! The version that handles Base objects works just fine for both Base objects and any class derived from Base!

The third case proceeds as a mix of the first two. First, the compiler matches variable bref with operator<< that takes a Base. That calls our virtual print() function. Since the Base reference is actually pointing to a Derived object, this resolves to Derived::print(), as we intended.

Problem solved.

12.x -- Chapter 12 comprehensive quiz

Index