25.11 — Printing inherited classes using operator<<

learncpp.com/cpp-tutorial/printing-inherited-classes-using-operator/

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

```
#include <iostream>

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

class Derived : public Base
{
public:
	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 referencing 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:
	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:
	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 21.5 -- 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?

A 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 rather than have `operator<<` determine what to print, we will instead have it call a normal member function that *can* be virtualized! This virtual function will do the work of determining what to print for each class.

In this first solution, our virtual member function (which we call `identify()`) returns a `std::string`, which is printed by `Base::operator<<`:

```
#include <iostream>

class Base
{
public:
	// Here's our overloaded operator<<
	friend std::ostream& operator<<(std::ostream& out, const Base& b)
	{
		// Call virtual function identify() to get the string to be printed
		out << b.identify();
		return out;
	}

	// We'll rely on member function identify() to return the string to be
printed
	// Because identify() is a normal member function, it can be virtualized
	virtual std::string identify() const
	{
		return "Base";
	}
};

class Derived : public Base
{
public:
	// Here's our override identify() function to handle the Derived case
	std::string identify() const override
	{
		return "Derived";
	}
};

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;
}
```

This prints the expected result:

```
Base
Derived
Derived
```

Let’s examine how this works in more detail.

In the case of `Base b`, `operator<<` is called with parameter `b` referencing the Base object. Virtual function call `b.identify()` thus resolves to `Base::identify()`, which returns “Base” to be printed. Nothing too special here.

In the case of `Derived d`, 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). Because parameter `b` is referencing a Derived object, virtual function call `b.identify()` resolves to `Derived::identify()`, which returns “Derived” to be printed.

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 reference. Because parameter `b` is referencing a Derived object, `b.identify()` resolves to `Derived::identify()`, which returns “Derived”.

Problem solved.

A more flexible solution

The above solution works great, but has two potential shortcomings:

1. It makes the assumption that the desired output can be represented as a single std::string.
2. Our `identify()` member function does not have access to the stream object.

The latter is problematic in cases where we need a stream object, such as when we want to print the value of a member variable that has an overloaded operator<<.

Fortunately, it’s straightforward to modify the above example to resolve both of these issues. In the previous version, virtual function `identify()` returned a string to be printed by `Base::operator<<`. In this version, we’ll instead define virtual member function `print()` and delegate responsibility for printing *directly* to that function.

Here’s an example that illustrates the idea:

```
#include <iostream>

class Base
{
public:
	// Here's our overloaded operator<<
	friend std::ostream& operator<<(std::ostream& out, const Base& b)
	{
		// Delegate printing responsibility for printing to virtual 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;
	}
};

// Some class or struct with an overloaded operator<<
struct Employee
{
	std::string name{};
	int id{};

	friend std::ostream& operator<<(std::ostream& out, const Employee& e)
	{
		out << "Employee(" << e.name << ", " << e.id << ")";
		return out;
	}
};

class Derived : public Base
{
private:
	Employee m_e{}; // Derived now has an Employee member

public:
	Derived(const Employee& e)
		: m_e{ e }
	{
	}

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

		// Print the Employee member using the stream object
```

```
                out << m_e;

                return out;
        }
};

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

        Derived d{ Employee{"Jim", 4}};
        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;
}
```

This outputs:

```
Base
Derived: Employee(Jim, 4)
Derived: Employee(Jim, 4)
```

In this version, `Base::operator<<` doesn't do any printing itself. Instead, it just calls virtual member function `print()` and passes it the stream object. The `print()` function then uses this stream object to do its own printing. `Base::print()` uses the stream object to print "Base". More interestingly, `Derived::print()` uses the stream object to print both "Derived: " and to call `Employee::operator<<` to print the value of member `m_e`. The latter would have been more challenging to do in the prior example!