

12.10 — Printing inherited classes using operator<<

BY ALEX ON NOVEMBER 23RD, 2016 | LAST MODIFIED BY ALEX ON SEPTEMBER 21ST, 2017

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

```
1  class Base
2  {
3  public:
4      Base() {}
5
6      virtual void print() const { std::cout << "Base"; }
7  };
8
9  class Derived : public Base
10 {
11 public:
12     Derived() {}
13
14     virtual void print() const override { std::cout << "Derived"; }
15 };
16
17 int main()
18 {
19     Derived d;
20     Base &b = d;
21     b.print(); // will call Derived::print()
22
23     return 0;
24 }
```

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`:

```
1  #include <iostream>
2  int main()
3  {
4      Derived d;
5      Base &b = d;
6
7      std::cout << "b is a ";
8      b.print(); // messy, we have to break our print statement to call this function
9      std::cout << '\n';
10
11     return 0;
12 }
```

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:

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

The challenges with operator<<

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

```
1  #include <iostream>
2  class Base
3  {
4  public:
5      Base() {}
6
7      virtual void print() const { std::cout << "Base"; }
```

```

8
9     friend std::ostream& operator<<(std::ostream &out, const Base &b)
10    {
11        out << "Base";
12        return out;
13    }
14 };
15
16 class Derived : public Base
17 {
18 public:
19     Derived() {}
20
21     virtual void print() const override { std::cout << "Derived"; }
22
23     friend std::ostream& operator<<(std::ostream &out, const Derived &d)
24     {
25         out << "Derived";
26         return out;
27     }
28
29 };
30
31 int main()
32 {
33     Base b;
34     std::cout << b << '\n';
35
36     Derived d;
37     std::cout << d << '\n';
38
39     return 0;
40 }

```

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:

```

1  int main()
2  {
3      Derived d;
4      Base &bref = d;
5      std::cout << bref << '\n';
6
7      return 0;
8  }

```

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:

```
1  #include <iostream>
2  class Base
3  {
4  public:
5      Base() {}
6
7      // Here's our overloaded operator<<
8      friend std::ostream& operator<<(std::ostream &out, const Base &b)
9      {
10         // Delegate printing responsibility for printing to member function print()
11         return b.print(out);
12     }
13
14     // We'll rely on member function print() to do the actual printing
15     // Because print is a normal member function, it can be virtualized
16     virtual std::ostream& print(std::ostream& out) const
17     {
18         out << "Base";
19         return out;
20     }
21 };
22
23 class Derived : public Base
24 {
25 public:
26     Derived() {}
27
28     // Here's our override print function to handle the Derived case
29     virtual std::ostream& print(std::ostream& out) const override
30     {
31         out << "Derived";
32         return out;
33     }
34 };
35
36 int main()
37 {
38     Base b;
39     std::cout << b << '\n';
40
41     Derived d;
42     std::cout << d << '\n'; // note that this works even with no operator<< that explicitly handles Der
43 ived objects
44 }
```

```

45     Base &bref = d,
46     std::cout << bref << '\n';
47
48     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](#)



[12.9 -- Dynamic casting](#)

Share this:



Facebook



Twitter



G+ Google



Pinterest

[C++ TUTORIAL](#) | [PRINT THIS POST](#)

16 comments to 12.10 — Printing inherited classes using operator<<



ARON

April 20, 2018 at 11:09 pm · Reply

[code]

```
friend std::ostream& operator<<(std::ostream &out, const Base &b)
```

```
{
```

```
// Delegate printing responsibility for printing to member function print()
```