

## 25.11 — Printing inherited classes using operator<<

👤 ALEX<sup>1</sup> ⌚ FEBRUARY 18, 2024

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

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

[COPY](#)

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`:

```
1  #include <iostream>
2
3  int main()
4  {
5      Derived d{};
6      Base& b{ d };
7
8      std::cout << "b is a ";
9      b.print(); // messy, we have to break our print statement to call this function
10     std::cout << '\n';
11
12     return 0;
13 }
```

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 |
3 | class Base
4 | {
5 | public:
6 |     virtual void print() const { std::cout << "Base"; }
7 |
8 |     friend std::ostream& operator<<(std::ostream& out, const Base& b)
9 |     {
10 |         out << "Base";
11 |         return out;
12 |     }
13 | };
14 |
15 | class Derived : public Base
16 | {
17 | public:
18 |     void print() const override { std::cout << "Derived"; }
19 |
20 |     friend std::ostream& operator<<(std::ostream& out, const Derived& d)
21 |     {
22 |         out << "Derived";
23 |         return out;
24 |     }
25 | };
26 |
27 | int main()
28 | {
29 |     Base b{};
30 |     std::cout << b << '\n';
31 |
32 |     Derived d{};
33 |     std::cout << d << '\n';
34 |
35 |     return 0;
36 | }
```

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 [21.5 -- Overloading operators using member functions](https://www.learncpp.com/cpp-tutorial/overloading-operators-using-member-functions/) (<https://www.learncpp.com/cpp-tutorial/overloading-operators-using-member-functions/>)<sup>2</sup>).

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<<:`

```

1  #include <iostream>
2
3  class Base
4  {
5  public:
6      // Here's our overloaded operator<<
7      friend std::ostream& operator<<(std::ostream& out, const Base& b)
8      {
9          // Call virtual function identify() to get the string to be printed
10         out << b.identify();
11         return out;
12     }
13
14     // We'll rely on member function identify() to return the string to be printed
15     // Because identify() is a normal member function, it can be virtualized
16     virtual std::string identify() const
17     {
18         return "Base";
19     }
20 };
21
22 class Derived : public Base
23 {
24 public:
25     // Here's our override identify() function to handle the Derived case
26     std::string identify() const override
27     {
28         return "Derived";
29     }
30 };
31
32 int main()
33 {
34     Base b{};
35     std::cout << b << '\n';
36
37     Derived d{};
38     std::cout << d << '\n'; // note that this works even with no operator<< that
39                             // explicitly handles Derived objects
40
41     Base& bref{ d };
42     std::cout << bref << '\n';
43
44     return 0;
45 }

```

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:



```

1 #include <iostream>
2
3 class Base
4 {
5 public:
6     // Here's our overloaded operator<<
7     friend std::ostream& operator<<(std::ostream& out, const Base& b)
8     {
9         // Delegate printing responsibility for printing to virtual member function
10    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 // Some class or struct with an overloaded operator<<
24 struct Employee
25 {
26     std::string name{};
27     int id{};
28
29     friend std::ostream& operator<<(std::ostream& out, const Employee& e)
30     {
31         out << "Employee(" << e.name << ", " << e.id << ")";
32         return out;
33     }
34 };
35
36 class Derived : public Base
37 {
38 private:
39     Employee m_e{}; // Derived now has an Employee member
40
41 public:
42     Derived(const Employee& e)
43         : m_e{ e }
44     {
45     }
46
47     // Here's our override print() function to handle the Derived case
48     std::ostream& print(std::ostream& out) const override
49     {
50         out << "Derived: ";
51
52         // Print the Employee member using the stream object
53         out << m_e;
54
55         return out;
56     }
57 };
58
59 int main()
60 {
61     Base b{};
62     std::cout << b << '\n';
63
64     Derived d{ Employee{"Jim", 4}};
65     std::cout << d << '\n'; // note that this works even with no operator<< that
66     explicitly handles Derived objects
67
68     Base& bref{ d };
69     std::cout << bref << '\n';
70
71 }

```

```
70 |     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!



## Next lesson

## 25.x Chapter 25 summary and quiz

3



[Back to table of contents](#)

4



## Previous lesson

## 25.10 Dynamic casting

5

6



B

U

URL

## INLINE CODE

### C++ CODE BLOCK

**HELP!**

 Name\*

Name\*

@ Email\* | ?

Email\*




Notify me about replies:



POST COMMENT

 Find a mistake? Leave a comment above!<sup>?</sup>

 Avatars from <https://gravatar.com/><sup>7</sup> are connected to your provided email address.



**Manas Ghosh**

🕒 June 16, 2025 12:33 pm PDT

In the last example explanation, `Base::operator<<` and `Employee::operator<<` look like `operator<<` is in scope of `Base` and `Employee` which is not the case as `operator<<` is friend here so it's in global scope. Explanation regarding this would be better to avoid any confusion.

👍 0    ➡ Reply

**Piotr Bojkoff**

🕒 October 28, 2024 12:19 pm PDT

So, what is generally considered a better practice: making a `toString()` member function and feeding its output to whatever stream might desire it, or making a `print()` member function and then feeding it to a `std::stringstream` if the need arises for a string?

👍 0    ➡ Reply

**Alex**

Author

➡ Reply to [Piotr Bojkoff](#)<sup>8</sup> 🕒 November 2, 2024 2:00 pm PDT

The former is a little more straightforward, so I'd go with that. Especially since modern C++ is moving away from stream-based I/O (in favor of `std::format` and `std::print`).

👍 0    ➡ Reply

**Piotr Bojkoff**➡ Reply to [Piotr Bojkoff](#)<sup>8</sup> 🕒 October 28, 2024 12:22 pm PDT

- and then feeding a `std::stringstream` to it

👍 0    ➡ Reply

**Shrinivas**

🕒 November 20, 2023 10:21 am PST

Why do we need to send the ostream object? The print function can print to standard output using cout right?

👍 2    ➡ Reply

**Alex**

Author

➡ Reply to [Shrinivas](#)<sup>9</sup> 🕒 November 20, 2023 2:02 pm PST

It allows the user of the function to specify which output stream they want (e.g. they could print to `std::cerr` instead of `std::cout`).

👍 2    ➡ Reply



**Strain**

➡ Reply to [Alex](#)<sup>10</sup>    ⌚ February 1, 2024 2:16 am PST

It could also be something unrelated to the console, like `std::stringstream` or really anything that derives from `std::ostream`.

👍 0    ➡ Reply



**Emeka Daniel**

⌚ May 22, 2023 10:00 am PDT

Awesome chapter Alex. I think this should be the last chapter reaching up to **.10**.

👍 1    ➡ Reply



**u262d**

⌚ April 5, 2023 12:15 am PDT

that heavy use of "friend"...

there are coding guidelines that forbids using friend. what to do then?

👍 0    ➡ Reply



**Alex**    Author

➡ Reply to [u262d](#)<sup>11</sup>    ⌚ April 5, 2023 10:06 pm PDT

Throw those coding guidelines in the trash. It's much better to make `operator<<` a friend when it needs to access private data than add access functions for the sole purpose of making `operator<<` a non-friend.

But if the guidelines won't let you do the better thing, do the worse thing.

👍 1    ➡ Reply



**Emeka Daniel**

➡ Reply to [Alex](#)<sup>12</sup>    ⌚ May 22, 2023 9:57 am PDT

:) (laughter emoji).

👍 0    ➡ Reply



**u262d**

➡ Reply to [Alex](#)<sup>12</sup>    ⌚ April 7, 2023 12:24 pm PDT

cite from company\_guidelines: friend classes MUST be avoided, when used it must be permitted by an architect.

so i cannot quite throw them in the wastebin

👍 0

➡ Reply



Alex

Author

➡ Reply to [u262d](#)<sup>13</sup> ⌚ April 9, 2023 11:49 am PDT

Friend classes and friend functions are different things. If only friend classes are restricted, you should still be able to use friend functions.

👍 2

➡ Reply



Nitin

⌚ February 27, 2023 5:41 am PST

Minor typo: The latter would have more challenging to do in the prior example!

would have 'been'..

Hope you don't mind my pointing out minor typos like these across various chapters. In case you only prefer technical discussions in the comments section, please let me know (I won't take it personally ) and I'll refrain from future typo reportings.

👍 0

➡ Reply



Alex

Author

➡ Reply to [Nitin](#)<sup>14</sup> ⌚ February 27, 2023 1:52 pm PST

Reporting typos is fine. Thank you for asking. FWIW, I'll generally delete such comments after fixing the typo so the comment section doesn't get cluttered with typo fix suggestions.

👍 2

➡ Reply



Avtem

⌚ January 4, 2023 11:55 pm PST

"\*The solution

The answer, as it turns out, is surprisingly simple.\*"

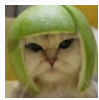
i actually don't think that's the best solution. As soon as i saw the previous solution (that doesn't work) i knew that it won't work and i kept asking why why why didn't you do it this way?:

```

1 #include <iostream>
2 class Base
3 {
4 public:
5     virtual std::string identify() const { return "Base"; }
6 };
7
8 class Derived: public Base
9 {
10 public:
11     std::string identify() const override { return "Derived"; }
12 };
13
14 std::ostream& operator<<(std::ostream& stream, const Base& base)
15 {
16     stream << base.identify();
17     return stream;
18 }
19
20 int main()
21 {
22     Base base;
23     Derived derived;
24     Base& b{derived};
25     std::cout << base << '\n';
26     std::cout << derived << '\n';
27     std::cout << b << '\n';
28 }

```

👍 2    ➡ Reply



**Alex** Author

🗨 Reply to Avtem<sup>15</sup> 🕒 January 5, 2023 7:34 pm PST

Thanks for the great feedback!

In this particular case, I agree your version is a simpler solution. But it also has a few downsides that make it potentially problematic for more complicated cases.

I've updated the article with your solution as a starter, and then present a modified version of my original solution that illustrates a case that would be difficult to solve using your method.

👍 3    ➡ Reply



**Helper831**

🕒 July 1, 2022 6:44 am PDT

```

1 // Here's our overloaded operator<<
2 friend std::ostream& operator<<(std::ostream& out, const Base& b)

```

A friend function isn't necessary here since it doesn't need direct access to any member variables. I think it makes more sense to either declare it as a normal function or make the print function private.

👍 6    ➡ Reply



Jim

🕒 November 8, 2021 9:00 am PST

I'm confused, why do you declare the operator<< overloading as friend in line 7 of your final example code. If you remove the "friend" word it will throw a compile error saying that operator overloading<< takes 1 parameter (it should even in your example i think), so the "friend" word is used to take 2 parameters? or why do we actually need it for, since everything is public? I kinda lost you there...

📝 Last edited 3 years ago by Jim

👍 0

➡ Reply



Alex

Author

👤 Reply to Jim<sup>16</sup> 🕒 November 9, 2021 9:04 am PST

With a non-friend member operator, the left operand becomes the implicit object, so function parameters are only required for non-left operands. With a friend function, there is no implicit operator, so function parameters are required for all operands. Thus friend operators will always have 1 more parameter than member operators.

👍 3

➡ Reply



Jim

👤 Reply to Alex<sup>17</sup> 🕒 November 9, 2021 1:20 pm PST

Oh I remember now, thanks.

👍 1

➡ Reply



Fred

🕒 August 25, 2021 12:04 pm PDT

The override specifier implies virtual. Using both virtual and override would be redundant:

```
1 | virtual void print() const override { std::cout << "Derived"; }  
2 | virtual std::ostream& print(std::ostream& out) const override
```

👍 2

➡ Reply



Alex

Author

👤 Reply to Fred<sup>18</sup> 🕒 August 26, 2021 10:43 am PDT

Fixed, thanks for pointing this out!

👍 1

➡ Reply

# Links

1. <https://www.learncpp.com/author/Alex/>
2. <https://www.learncpp.com/cpp-tutorial/overloading-operators-using-member-functions/>
3. <https://www.learncpp.com/cpp-tutorial/chapter-25-summary-and-quiz/>
4. <https://www.learncpp.com/>
5. <https://www.learncpp.com/cpp-tutorial/dynamic-casting/>
6. <https://www.learncpp.com/printing-inherited-classes-using-operator/>
7. <https://gravatar.com/>
8. <https://www.learncpp.com/cpp-tutorial/printing-inherited-classes-using-operator/#comment-603619>
9. <https://www.learncpp.com/cpp-tutorial/printing-inherited-classes-using-operator/#comment-590028>
10. <https://www.learncpp.com/cpp-tutorial/printing-inherited-classes-using-operator/#comment-590055>
11. <https://www.learncpp.com/cpp-tutorial/printing-inherited-classes-using-operator/#comment-579014>
12. <https://www.learncpp.com/cpp-tutorial/printing-inherited-classes-using-operator/#comment-579042>
13. <https://www.learncpp.com/cpp-tutorial/printing-inherited-classes-using-operator/#comment-579090>
14. <https://www.learncpp.com/cpp-tutorial/printing-inherited-classes-using-operator/#comment-577894>
15. <https://www.learncpp.com/cpp-tutorial/printing-inherited-classes-using-operator/#comment-575858>
16. <https://www.learncpp.com/cpp-tutorial/printing-inherited-classes-using-operator/#comment-562631>
17. <https://www.learncpp.com/cpp-tutorial/printing-inherited-classes-using-operator/#comment-562655>
18. <https://www.learncpp.com/cpp-tutorial/printing-inherited-classes-using-operator/#comment-560591>
19. <https://g.ezoic.net/privacy/learncpp.com>