9.3 — Overloading the I/O operators

BY ALEX ON OCTOBER 1ST, 2007 | LAST MODIFIED BY ALEX ON JANUARY 23RD, 2020

For classes that have multiple member variables, printing each of the individual variables on the screen can get tiresome fast. For example, consider the following class:

```
class Point
{
private:
    double m_x, m_y, m_z;

public:
    Point(double x=0.0, double y=0.0, double z=0.0): m_x(x), m_y(y), m_z(z)
    {
    }

    double getX() { return m_x; }
    double getY() { return m_y; }
    double getZ() { return m_z; }
};
```

If you wanted to print an instance of this class to the screen, you'd have to do something like this:

```
Point point(5.0, 6.0, 7.0);

std::cout << "Point(" << point.getX() << ", " <<
    point.getY() << ", " <<
    point.getZ() << ")";
```

Of course, it makes more sense to do this as a reusable function. And in previous examples, you've seen us create print() functions that work like this:

```
class Point
{
private:
    double m_x, m_y, m_z;

public:
    Point(double x=0.0, double y=0.0, double z=0.0): m_x(x), m_y(y), m_z(z)
    {
    }

    double getX() { return m_x; }
    double getY() { return m_y; }
    double getZ() { return m_z; }

    void print()
    {
        std::cout << "Point(" << m_x << ", " << m_y << ", " << m_z << ")";
    }
};
```

While this is much better, it still has some downsides. Because print() returns void, it can't be called in the middle of an output statement. Instead, you have to do this:

```
int main()
{
    Point point(5.0, 6.0, 7.0);

```

```
    std::cout << "My point is: ";
    point.print();
    std::cout << " in Cartesian space.\n";
}
```

It would be much easier if you could simply type:

```
Point point(5.0, 6.0, 7.0);
cout << "My point is: " << point << " in Cartesian space.\n";
```

and get the same result. There would be no breaking up output across multiple statements, and no having to remember what you named the print function.

Fortunately, by overloading the << operator, you can!

Overloading operator<<

Overloading operator<< is similar to overloading operator+ (they are both binary operators), except that the parameter types are different.

Consider the expression `std::cout << point`. If the operator is <<, what are the operands? The left operand is the std::cout object, and the right operand is your Point class object. std::cout is actually an object of type std::ostream. Therefore, our overloaded function will look like this:

```
// std::ostream is the type for object std::cout
friend std::ostream& operator<< (std::ostream &out, const Point &point);
```

Implementation of operator<< for our Point class is fairly straightforward -- because C++ already knows how to output doubles using operator<<, and our members are all doubles, we can simply use operator<< to output the member variables of our Point. Here is the above Point class with the overloaded operator<<.

```
#include <iostream>

class Point
{
private:
    double m_x, m_y, m_z;

public:
    Point(double x=0.0, double y=0.0, double z=0.0): m_x(x), m_y(y), m_z(z)
    {
    }

    friend std::ostream& operator<< (std::ostream &out, const Point &point);
};

std::ostream& operator<< (std::ostream &out, const Point &point)
{
    // Since operator<< is a friend of the Point class, we can access Point's members directly
    out << "Point(" << point.m_x << ", " << point.m_y << ", " << point.m_z << ")"; // actual o

    return out; // return std::ostream so we can chain calls to operator<<
}

int main()
{
    Point point1(2.0, 3.0, 4.0);

    std::cout << point1;

    return 0;
}
```

This is pretty straightforward -- note how similar our output line is to the line in the print() function we wrote previously. The most notable difference is that std::cout has become parameter out (which will be a reference to std::cout when the function is called).

The trickiest part here is the return type. With the arithmetic operators, we calculated and returned a single answer by value (because we were creating and returning a new result). However, if you try to return std::ostream by value, you'll get a compiler error. This happens because std::ostream specifically disallows being copied.

In this case, we return the left hand parameter as a reference. This not only prevents a copy of std::ostream from being made, it also allows us to "chain" output commands together, such as `std::cout << point << std::endl;`

You might have initially thought that since operator<< doesn't return a value to the caller that we should define the function as returning void. But consider what would happen if our operator<< returned void. When the compiler evaluates `std::cout << point << std::endl;`, due to the precedence/associativity rules, it evaluates this expression as `(std::cout << point) << std::endl;`. `std::cout << point` would call our void-returning overloaded operator<< function, which returns void. Then the partially evaluated expression becomes: `void << std::endl;`, which makes no sense!

By returning the out parameter as the return type instead, `(std::cout << point)` returns std::cout. Then our partially evaluated expression becomes: `std::cout << std::endl;`, which then gets evaluated itself!

Any time we want our overloaded binary operators to be chainable in such a manner, the left operand should be returned (by reference). Returning the left-hand parameter by reference is okay in this case -- since the left-hand parameter was passed in by the calling function, it must still exist when the called function returns. Therefore, we don't have to worry about referencing something that will go out of scope and get destroyed when the operator returns.

Just to prove it works, consider the following example, which uses the Point class with the overloaded operator<< we wrote above:

```
#include <iostream>

class Point
{
private:
    double m_x, m_y, m_z;

public:
    Point(double x=0.0, double y=0.0, double z=0.0): m_x(x), m_y(y), m_z(z)
    {
    }

    friend std::ostream& operator<< (std::ostream &out, const Point &point);
};

std::ostream& operator<< (std::ostream &out, const Point &point)
{
    // Since operator<< is a friend of the Point class, we can access Point's members directly
    out << "Point(" << point.m_x << ", " << point.m_y << ", " << point.m_z << ")";

    return out;
}

int main()
{
    Point point1(2.0, 3.5, 4.0);
    Point point2(6.0, 7.5, 8.0);

    std::cout << point1 << " " << point2 << '\n';
```

```
30
31      return 0;
32  }
```

This produces the following result:

```
Point(2, 3.5, 4) Point(6, 7.5, 8)
```

Overloading operator>>

It is also possible to overload the input operator. This is done in a manner analogous to overloading the output operator. The key thing you need to know is that std::cin is an object of type std::istream. Here's our Point class with an overloaded operator>>:

```
1   #include <iostream>
2
3   class Point
4   {
5   private:
6       double m_x, m_y, m_z;
7
8   public:
9       Point(double x=0.0, double y=0.0, double z=0.0): m_x(x), m_y(y), m_z(z)
10      {
11      }
12
13      friend std::ostream& operator<< (std::ostream &out, const Point &point);
14      friend std::istream& operator>> (std::istream &in, Point &point);
15  };
16
17  std::ostream& operator<< (std::ostream &out, const Point &point)
18  {
19      // Since operator<< is a friend of the Point class, we can access Point's members directly
20      out << "Point(" << point.m_x << ", " << point.m_y << ", " << point.m_z << ")";
21
22      return out;
23  }
24
25  std::istream& operator>> (std::istream &in, Point &point)
26  {
27      // Since operator>> is a friend of the Point class, we can access Point's members directly
28      // note that parameter point must be non-const so we can modify the class members with the
29      in >> point.m_x;
30      in >> point.m_y;
31      in >> point.m_z;
32
33      return in;
34  }
```

Here's a sample program using both the overloaded operator<< and operator>>:

```
1   int main()
2   {
3       std::cout << "Enter a point: \n";
4
5       Point point;
6       std::cin >> point;
7
8       std::cout << "You entered: " << point << '\n';
9
```

```
    return 0;
}
```

Assuming the user enters `3.0 4.5 7.26` as input, the program produces the following result:

```
You entered: Point(3, 4.5, 7.26)
```

Conclusion

Overloading operator<< and operator>> make it extremely easy to output your class to screen and accept user input from the console.

Quiz time

Take the Fraction class we wrote in the previous quiz (listed below) and add an overloaded operator<< and operator>> to it.

The following program should compile:

```
int main()
{

    Fraction f1;
    std::cout << "Enter fraction 1: ";
    std::cin >> f1;

    Fraction f2;
    std::cout << "Enter fraction 2: ";
    std::cin >> f2;

    std::cout << f1 << " * " << f2 << " is " << f1 * f2 << '\n'; // note: The result of f1 * f

    return 0;
}
```

And produce the result:

```
Enter fraction 1: 2/3
Enter fraction 2: 3/8
2/3 * 3/8 is 1/4
```

Here's the Fraction class:

```
#include <iostream>

class Fraction
{
private:
    int m_numerator;
    int m_denominator;

public:
    Fraction(int numerator=0, int denominator=1):
        m_numerator(numerator), m_denominator(denominator)
    {
        // We put reduce() in the constructor to ensure any new fractions we make get reduced!
        // Any fractions that are overwritten will need to be re-reduced
        reduce();
    }

```

```
18      // We'll make gcd static so that it can be part of class Fraction without requiring an obj
19      static int gcd(int a, int b)
20      {
21          return b == 0 ? a : gcd(b, a % b);
22      }
23
24      void reduce()
25      {
26          if (m_numerator != 0 && m_denominator != 0)
27          {
28              int gcd = Fraction::gcd(m_numerator, m_denominator);
29              m_numerator /= gcd;
30              m_denominator /= gcd;
31          }
32      }
33
34      friend Fraction operator*(const Fraction &f1, const Fraction &f2);
35      friend Fraction operator*(const Fraction &f1, int value);
36      friend Fraction operator*(int value, const Fraction &f1);
37
38      void print()
39      {
40          std::cout << m_numerator << "/" << m_denominator << "\n";
41      }
42  };
43
44  Fraction operator*(const Fraction &f1, const Fraction &f2)
45  {
46      return Fraction(f1.m_numerator * f2.m_numerator, f1.m_denominator * f2.m_denominator);
47  }
48
49  Fraction operator*(const Fraction &f1, int value)
50  {
51      return Fraction(f1.m_numerator * value, f1.m_denominator);
52  }
53
54  Fraction operator*(int value, const Fraction &f1)
55  {
56      return Fraction(f1.m_numerator * value, f1.m_denominator);
57  }
```

Show Solution

9.4 -- Overloading operators using member functions

Index

9.2a -- Overloading operators using normal functions

C++ TUTORIAL | PRINT THIS POST

241 comments to 9.3 — Overloading the I/O operators

« Older Comments 1 2 3 4

Oliver
February 6, 2020 at 3:06 pm · Reply

Hi,
Would it be logical to make the operator overloaded function for << (output stream) a CONST function, as it is virtually the same as a 'void print()' function which we have been dubbing with 'void print() const'?
For example, is it wrong to write,
friend std::ostream& operator<<(std::ostream& out, const Fraction f) const;
?

Thanks

nascardriver
February 7, 2020 at 8:46 am · Reply

`operator<<` isn't a member function, it can't be `const`.

Ryan
December 10, 2019 at 6:10 am · Reply

Can you make the function "friend std::ostream& operator<<(std::ostream &out, const Fraction &f1)" a static.

Since the operator<< function above does not use the hidden this pointer. However static are member functions but friend functions are non-member functions, so would it be possible? If so, how could it be implemented. Just putting static infront of the word friend?

nascardriver
December 11, 2019 at 4:56 am · Reply

`static` combined with `friend` doesn't make sense. `static` members already have access to private data.
`operator<<` can't be a (`static`) member. When you use `<<`, the operator is searched for in the global namespace and the namespace of the argument, but not inside of classes.

Samira Ferdi
November 25, 2019 at 7:24 pm · Reply

Hi, Alex and Nascardriver!

Why frac is const Fraction?

```
1 friend std::ostream& operator<<(std::ostream &output, const Fraction &frac);
```

If, I remove the const, I have compile errors.

Alex
November 25, 2019 at 10:54 pm · Reply

Mainly because it's a good idea to pass objects by const reference instead of by reference to ensure the function doesn't inadvertently modify the argument.

But the reason it fails in this specific case is because f1 * f2 is an r-value, and r-values can only bind to const references.

Kevin
November 16, 2019 at 4:19 pm · Reply

Hi. I am working on an assignment for a class, and we are being asked to make a dog class with the specifications below. I have tried to compile this in many different ways but I keep getting one error that says that the operator<< function must only take one argument, which I don't understand since it obviously requires 2 (the out stream object and the user defined object). I've included my class definition and the operator<< function implementation, as well as the error message. Please help!

```
1  #include <string>
2  #include <iostream>
3
4  class Dog;
5
6  std::ostream& operator<< (std::ostream&, const Dog&);
7
8  class Dog
9  {
10     private:
11         std::string name;
12         float weight;
13         int age;
14     public:
15         Dog(std::string, float, int);
16         ~Dog();
17         void displayDog();
18         bool operator>= (const int&);
19         bool operator< (const Dog&);
20         bool operator== (const Dog&);
21         friend std::ostream& operator<< (std::ostream&, const Dog&);
22 };
23
24 std::ostream& Dog::operator<< (std::ostream& s, const Dog& rhs)
```

```
{
    s << "NAME: " << rhs.name << std::endl << "WEIGHT: " << rhs.weight << " pounds" << std:

    return s;
}
```

In file included from main.cpp:4:0:
Dog.h:82:63: error: 'std::ostream& Dog::operator<<(std::ostream&, const Dog&)' must take exactly one argument
std::ostream& Dog::operator<< (std::ostream& s, const Dog& rhs)

Kevin
November 16, 2019 at 5:45 pm · Reply

Nevermind, I figured it out. I had to declare the operator<< function outside of the class, then declare it as friend inside the class. THEN (the important part), don't include the Dog:: part in the function implementation, because the operator<< function isn't part of the Dog class. Like so:

```
class Dog;

std::ostream& operator<< (std::ostream&, const Dog&);

class Dog
{
    private:
        std::string name;
        float weight;
        int age;
    public:
        Dog(std::string, float, int);
        ~Dog();
        void displayDog();
        bool operator>= (const int&);
        bool operator< (const Dog&);
        bool operator== (const Dog&);
        friend std::ostream& operator<< (std::ostream&, const Dog&);
};

std::ostream& operator<< (std::ostream& s, const Dog& rhs)
{
    s << "NAME: " << rhs.name << std::endl << "WEIGHT: " << rhs.weight << " pounds" <

    return s;
}
```

nascardriver
November 17, 2019 at 2:01 am · Reply

Exactly. If you declare it as a member function, you're overloading the shift operator with you class implicitly on the left.

```
dog << "hello"; // member function
stream << dog; // non-member function with std::ostream
```

hellmet
November 16, 2019 at 4:58 am · Reply

Teehehehe

I think this would be a fun quiz question!

```
...
// what happens now :D
out << "Point(" << point << ", " << point.m_y << ", " << point.m_z << ")";
...
(Point example, line 19)
```

hersel99
November 11, 2019 at 11:34 am · Reply

Quiz solution: line 66. Unused variable `char c`.
When I was figuring out how to get rid of the slash `/` from `2/3`, I know it should be done with `in.ignore`, but because I was unsure, I also use a temporal `char`.

Idk why but I found this funny XD

Taksh
October 17, 2019 at 12:50 pm · Reply

Hey Alex,

Thank you for this great website. I truly appreciate your efforts.

So, I followed your instructions. But I keep getting an error when I try to print a variable:

Invalid operands to binary expression.

What do you think might be causing this?

nascardriver
October 18, 2019 at 1:30 am · Reply

Show your code and the full error message with line numbers.

Jeremiah
September 25, 2019 at 6:20 am · Reply

In the answer to your quiz, you ignore the '/' character by inserting it into a unused character "char c" . See your code below:

```
std::istream& operator>>(std::istream &in, Fraction &f1)
{
    char c;

    // Overwrite the values of f1
    in >> f1.m_numerator;
    in >> c; // ignore the '/' separator
    in >> f1.m_denominator;

    // Since we overwrite the existing f1, we need to reduce again
    f1.reduce();

    return in;
}
```

However, you can avoid creating this unnecessary variable if you use the ignore function built into the istream object:

```
std::istream& operator>>(std::istream &in, Fraction &f1)
{
     // Overwrite the values of f1
    in >> f1.m_numerator;
    in.ignore(1); // ignore the '/' separator
    in >> f1.m_denominator;

    // Since we overwrite the existing f1, we need to reduce again
    f1.reduce();

    return in;
}
```

This seems more intuitive to me. Is there any reason to use the former over latter?

Alex
September 26, 2019 at 3:01 pm · Reply

Only if you want to examine the separator after extracting it. In this case, we don't do that, so your solution is better. I've updated the lesson. Thanks!

Barne
September 21, 2019 at 7:12 am · Reply

"In this case, we return the left hand parameter as a reference. This not only prevents a copy of std::ostream from being made, it also allows us to “chain” output commands together, such as std::cout << point << std::endl;"

This instantly makes sense thanks to the previous "this pointer" lesson!

I just wish we learned a little more about how std::cout works at this point. We've been using it since "Hello World!", and it's just this mystery object we feed stuff into and it gets output to a terminal somehow.

Eru
August 23, 2019 at 1:46 am · Reply

Hello, it was stated that;

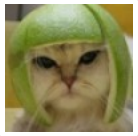
"Any time we want our overloaded binary operators to be chainable in such a manner, the left operand should be returned (by reference)."

However in one of the previous chapters, we've chained the operator+ for MinMax without returning by reference.
Does the below code(line 56) for MinMax not considered as chaining or am i missing something here?

Thanks beforehand :)

```
class MinMax
{
private:
    int m_min; // The min value seen so far
    int m_max; // The max value seen so far

public:
    MinMax(int min, int max)
    {
```

```
        m_min = min;
        m_max = max;
    }

    int getMin() { return m_min; }
    int getMax() { return m_max; }

    friend MinMax operator+(const MinMax &m1, const MinMax &m2);
    friend MinMax operator+(const MinMax &m, int value);
    friend MinMax operator+(int value, const MinMax &m);
};

MinMax operator+(const MinMax &m1, const MinMax &m2)
{
    // Get the minimum value seen in m1 and m2
    int min = m1.m_min < m2.m_min ? m1.m_min : m2.m_min;

    // Get the maximum value seen in m1 and m2
    int max = m1.m_max > m2.m_max ? m1.m_max : m2.m_max;

    return MinMax(min, max);
}

MinMax operator+(const MinMax &m, int value)
{
    // Get the minimum value seen in m and value
    int min = m.m_min < value ? m.m_min : value;

    // Get the maximum value seen in m and value
    int max = m.m_max > value ? m.m_max : value;

    return MinMax(min, max);
}

MinMax operator+(int value, const MinMax &m)
{
    // call operator+(MinMax, int)
    return m + value;
}

int main()
{
    MinMax m1(10, 15);
    MinMax m2(8, 11);
    MinMax m3(3, 12);

    MinMax mFinal = m1 + m2 + 5 + 8 + m3 + 16;

    std::cout << "Result: (" << mFinal.getMin() << ", " <<
        mFinal.getMax() << ")\n";

    return 0;
}
```



Alex
August 24, 2019 at 1:12 pm · Reply

Yeah, the "in such a manner" is actually an important qualifier here, as I'm not trying to define an absolute rule, but rather one specific to this case.

Basically, if the overloaded operator returns a temporary object, you need to return the temporary object by value, as returning it by address or reference would result in a dangling pointer/reference when the temporary object went out of scope. This is typically the case in operators overloaded as friend functions.

In the case where the implicit object is being modified, you can return the implicit object itself by reference. This is typically the case in operators overloaded as member functions.

« Older Comments 1 2 3 4