

21.4 — Overloading the I/O operators

 learncpp.com/cpp-tutorial/overloading-the-io-operators/

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

    void print() const
    {
        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()
{
    const 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{};
    double m_y{};
    double 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 output done here

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

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

    std::cout << point1 << '\n';

    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;`

Consider what would happen if our `operator<<` returned `void` instead. When the compiler evaluates `std::cout << point << '\n'`, due to the precedence/associativity rules, it evaluates this expression as `(std::cout << point) << '\n'`; `std::cout << point` would call our `void`-returning overloaded `operator<<` function, which returns `void`. Then the partially evaluated expression becomes: `void << '\n'`;, 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 << '\n'`;, 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{};
    double m_y{};
    double 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';

    return 0;
}

```

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>>`:

```

#include <iostream>

class Point
{
private:
    double m_x{};
    double m_y{};
    double 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);
    friend std::istream& operator>> (std::istream& in, 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;
}

std::istream& operator>> (std::istream& in, Point& point)
{
    // Since operator>> is a friend of the Point class, we can access Point's members
    directly.
    // note that parameter point must be non-const so we can modify the class members
    with the input values
    in >> point.m_x;
    in >> point.m_y;
    in >> point.m_z;

    return in;
}

```

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

```

int main()
{
    std::cout << "Enter a point: ";

    Point point;
    std::cin >> point;

    std::cout << "You entered: " << point << '\n';

    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 * f2 is an r-value

    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>
#include <numeric> // for std::gcd

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();
    }

    void reduce()
    {
        int gcd{ std::gcd(m_numerator, m_denominator) };
        if (gcd)
        {
            m_numerator /= gcd;
            m_denominator /= gcd;
        }
    }

    friend Fraction operator*(const Fraction& f1, const Fraction& f2);
    friend Fraction operator*(const Fraction& f1, int value);
    friend Fraction operator*(int value, const Fraction& f1);

    void print() const
    {
        std::cout << m_numerator << '/' << m_denominator << '\n';
    }
};

Fraction operator*(const Fraction& f1, const Fraction& f2)
{
    return Fraction(f1.m_numerator * f2.m_numerator, f1.m_denominator *
f2.m_denominator);
}

Fraction operator*(const Fraction& f1, int value)
{
    return Fraction(f1.m_numerator * value, f1.m_denominator);
}

Fraction operator*(int value, const Fraction& f1)
{

```



```
        return Fraction(f1.m_numerator * value, f1.m_denominator);  
    }
```

If you're on a pre-C++17 compiler, you can replace `std::gcd` with this function:

```
#include <cmath>  
  
int gcd(int a, int b) {  
    return (b == 0) ? std::abs(a) : gcd(b, a % b);  
}
```

[Show Solution](#)