21.8 — Overloading the increment and decrement operators

Overloading the increment (`++`) and decrement (`--`) operators is pretty straightforward, with one small exception. There are actually two versions of the increment and decrement operators: a prefix increment and decrement (e.g. `++x; --y;`) and a postfix increment and decrement (e.g. `x++; y--;`).

Because the increment and decrement operators are both unary operators and they modify their operands, they're best overloaded as member functions. We'll tackle the prefix versions first because they're the most straightforward.

Overloading prefix increment and decrement

Prefix increment and decrement are overloaded exactly the same as any normal unary operator. We'll do this one by example:

```
#include <iostream>

class Digit
{
private:
    int m_digit{};
public:
    Digit(int digit=0)
        : m_digit{digit}
    {
    }

    Digit& operator++();
    Digit& operator--();

    friend std::ostream& operator<< (std::ostream& out, const Digit& d);
};

Digit& Digit::operator++()
{
    // If our number is already at 9, wrap around to 0
    if (m_digit == 9)
        m_digit = 0;
    // otherwise just increment to next number
    else
        ++m_digit;

    return *this;
}

Digit& Digit::operator--()
{
    // If our number is already at 0, wrap around to 9
    if (m_digit == 0)
        m_digit = 9;
    // otherwise just decrement to next number
    else
        --m_digit;

    return *this;
}

std::ostream& operator<< (std::ostream& out, const Digit& d)
{
        out << d.m_digit;
        return out;
}

int main()
{
    Digit digit { 8 };
```

```
    std::cout << digit;
    std::cout << ++digit;
    std::cout << ++digit;
    std::cout << --digit;
    std::cout << --digit;

    return 0;
}
```

Our Digit class holds a number between 0 and 9. We've overloaded increment and decrement so they increment/decrement the digit, wrapping around if the digit is incremented/decremented out of range.

This example prints:

```
89098
```

Note that we return *this. The overloaded increment and decrement operators return the current implicit object so multiple operators can be "chained" together.

Overloading postfix increment and decrement

Normally, functions can be overloaded when they have the same name but a different number and/or different type of parameters. However, consider the case of the prefix and postfix increment and decrement operators. Both have the same name (eg. operator++), are unary, and take one parameter of the same type. So how it is possible to differentiate the two when overloading?

The C++ language specification has a special case that provides the answer: the compiler looks to see if the overloaded operator has an int parameter. If the overloaded operator has an int parameter, the operator is a postfix overload. If the overloaded operator has no parameter, the operator is a prefix overload.

Here is the above Digit class with both prefix and postfix overloads:

```
class Digit
{
private:
    int m_digit{};
public:
    Digit(int digit=0)
        : m_digit{digit}
    {
    }

    Digit& operator++(); // prefix has no parameter
    Digit& operator--(); // prefix has no parameter

    Digit operator++(int); // postfix has an int parameter
    Digit operator--(int); // postfix has an int parameter

    friend std::ostream& operator<< (std::ostream& out, const Digit& d);
};

// No parameter means this is prefix operator++
Digit& Digit::operator++()
{
    // If our number is already at 9, wrap around to 0
    if (m_digit == 9)
        m_digit = 0;
    // otherwise just increment to next number
    else
        ++m_digit;

    return *this;
}

// No parameter means this is prefix operator--
Digit& Digit::operator--()
{
    // If our number is already at 0, wrap around to 9
    if (m_digit == 0)
        m_digit = 9;
    // otherwise just decrement to next number
    else
        --m_digit;

    return *this;
}

// int parameter means this is postfix operator++
Digit Digit::operator++(int)
{
    // Create a temporary variable with our current digit
    Digit temp{*this};

    // Use prefix operator to increment this digit
```

```
    ++(*this); // apply operator

    // return temporary result
    return temp; // return saved state
}

// int parameter means this is postfix operator--
Digit Digit::operator--(int)
{
    // Create a temporary variable with our current digit
    Digit temp{*this};

    // Use prefix operator to decrement this digit
    --(*this); // apply operator

    // return temporary result
    return temp; // return saved state
}

std::ostream& operator<< (std::ostream& out, const Digit& d)
{
        out << d.m_digit;
        return out;
}

int main()
{
    Digit digit { 5 };

    std::cout << digit;
    std::cout << ++digit; // calls Digit::operator++();
    std::cout << digit++; // calls Digit::operator++(int);
    std::cout << digit;
    std::cout << --digit; // calls Digit::operator--();
    std::cout << digit--; // calls Digit::operator--(int);
    std::cout << digit;

    return 0;
}
```

This prints

```
5667665
```

There are a few interesting things going on here. First, note that we've distinguished the prefix from the postfix operators by providing an integer dummy parameter on the postfix version. Second, because the dummy parameter is not used in the function implementation, we have not even given it a name. This tells the compiler to treat this variable as a placeholder, which means it won't warn us that we declared a variable but never used it.

Third, note that the prefix and postfix operators do the same job -- they both increment or decrement the object. The difference between the two is in the value they return. The overloaded prefix operators return the object after it has been incremented or decremented. Consequently, overloading these is fairly straightforward. We simply increment or decrement our member variables, and then return *this.

The postfix operators, on the other hand, need to return the state of the object *before* it is incremented or decremented. This leads to a bit of a conundrum -- if we increment or decrement the object, we won't be able to return the state of the object before it was incremented or decremented. On the other hand, if we return the state of the object before we increment or decrement it, the increment or decrement will never be called.

The typical way this problem is solved is to use a temporary variable that holds the value of the object before it is incremented or decremented. Then the object itself can be incremented or decremented. And finally, the temporary variable is returned to the caller. In this way, the caller receives a copy of the object before it was incremented or decremented, but the object itself is incremented or decremented. Note that this means the return value of the overloaded operator must be a non-reference, because we can't return a reference to a local variable that will be destroyed when the function exits. Also note that this means the postfix operators are typically less efficient than the prefix operators because of the added overhead of instantiating a temporary variable and returning by value instead of reference.

Finally, note that we've written the post-increment and post-decrement in such a way that it calls the pre-increment and pre-decrement to do most of the work. This cuts down on duplicate code, and makes our class easier to modify in the future.