Unary Operator Overloading in C++

Serob Tigranyan

December 15, 2025

Contents

1 Unary Operator Overloading

1.1 Basics of Unary Operators

Unary operators are operators that work on a single operand. C++ provides several built-in unary operators such as + (unary plus), - (unary minus), ++ (increment), -- (decrement), ! (logical NOT), and ~ (bitwise NOT).

When working with custom types, you can overload these operators to define their behavior for your classes, allowing your objects to behave more intuitively and naturally in expressions.

Basic usage of unary operator overloading

```
class Vector {
private:
    double x, y;
public:
    Vector(double x, double y) : x(x), y(y) {}

    // Overload unary minus
    Vector operator-() const {
        return Vector(-x, -y);
    }

    // Overload unary plus
    Vector operator+() const {
        return *this;
    }
};
```

1.2 Prefix vs Postfix Increment/Decrement

The increment (++) and decrement (--) operators are unique among unary operators because they have both *prefix* and *postfix* forms. To distinguish between them in overloading, C++ uses a dummy `int` parameter for the postfix version.

Implementing prefix and postfix operators

```
class Counter {
private:
    int value;
public:
    Counter(int v = 0) : value(v) {}

    // Prefix increment: ++counter
    Counter& operator++() {
        ++value;
        return *this;
    }

    // Postfix increment: counter++
    Counter operator++(int) {
        Counter temp = *this;
        ++value;
        return temp;
    }

    int get_value() const { return value; }
};
```

Notice the key differences:

- *Prefix* (`++x`) returns a reference to the modified object
- *Postfix* (`x++`) returns a copy of the original value before modification
- The dummy `int` parameter distinguishes postfix from prefix

Using the operators

```
#include <iostream>

int main() {
    Counter c(5);

    std::cout << (++c).get_value() << std::endl;  // Outputs: 6
    std::cout << c.get_value() << std::endl;      // Outputs: 6

    std::cout << (c++).get_value() << std::endl;  // Outputs: 6
    std::cout << c.get_value() << std::endl;      // Outputs: 7

    return 0;
}
```

1.3 Rules for Unary Operator Overloading

When overloading unary operators, there are several important rules to follow:

1. Unary operators can be overloaded as *member functions* or *non-member functions*
2. As a member function, unary operators take no parameters (except postfix ++=/−= which take a dummy `int`)
3. As a non-member function, they take one parameter (the object)
4. Prefix operators should return a reference to allow chaining: `++(++x)`
5. Postfix operators should return by value, representing the pre-modification state
6. The `const` qualifier should be used when the operation doesn't modify the object

Member vs Non-member function example

```
class Number {
private:
    int value;
public:
    Number(int v) : value(v) {}

    // Member function version
    Number operator-() const {
        return Number(-value);
    }

    friend Number operator+(const Number& n);
};

// Non-member function version (requires friend or public access)
Number operator+(const Number& n) {
    return n;  // Unary plus just returns a copy
}
```

1.4 Common Unary Operators

Here's a comprehensive example showing all unary operator overloads:

A complete example class demonstrating all unary operators

```
#include <iostream>

class Number {
private:
    int value;
public:
    Number(int v = 0) : value(v) {}

    // Unary plus: +x
    Number operator+() const {
        return Number(+value);
    }

    // Unary minus: -x (negation)
    Number operator-() const {
        return Number(-value);
    }

    // Prefix increment: ++x
    Number& operator++() {
        ++value;
        return *this;
    }

    // Postfix increment: x++
    Number operator++(int) {
        Number temp = *this;
        ++value;
        return temp;
    }

    // Prefix decrement: --x
    Number& operator--() {
        --value;
        return *this;
    }

    // Postfix decrement: x--
    Number operator--(int) {
        Number temp = *this;
        --value;
        return temp;
    }

    // Logical NOT: !x
    bool operator!() const {
        return value == 0;
    }

    // Bitwise NOT: ~x
    Number operator~() const {
        return Number(~value);
    }

    // Address-of: &x (rarely overloaded, shown for completeness)
    Number* operator&() {
        return this;
    }

    // Dereference: *x (only makes sense for pointer-like classes)
    int operator*() const {
        return value;
    }

    int get_value() const { return value; }
};

int main() {
    Number n(5);

    std::cout << (+n).get_value() << std::endl;   // Unary plus: 5
    std::cout << (-n).get_value() << std::endl;   // Negation: -5
    std::cout << (++n).get_value() << std::endl;  // Prefix inc: 6
    std::cout << (n++).get_value() << std::endl;  // Postfix inc: 6
    std::cout << n.get_value() << std::endl;      // Now: 7
    std::cout << (--n).get_value() << std::endl;  // Prefix dec: 6
    std::cout << (n--).get_value() << std::endl;  // Postfix dec: 6
    std::cout << n.get_value() << std::endl;      // Now: 5
    std::cout << !n << std::endl;                 // Logical NOT: 0 (false)
    std::cout << (~n).get_value() << std::endl;   // Bitwise NOT: -6

    return 0;
}
```

Notice how each operator serves a specific purpose:

- `operator+()` returns a copy (unary plus)
- `operator-()` creates a new Number with negated value
- `operator++()` prefix returns reference for chaining
- `operator++(int)` postfix returns original value
- `operator--()` and `operator--(int)` work similarly for decrement
- `operator!()` checks if the number is zero
- `operator~()` performs bitwise complement
- `operator&()` returns the address (rarely overloaded)
- `operator*()` dereference operator (useful for smart pointer-like classes)

2 Summary

Unary operator overloading in C++ allows you to define how operators like `-`, `++`, `--`, and `!` work with your custom classes. The key distinction is between prefix and postfix increment/decrement operators, which are differentiated by a dummy `int` parameter for postfix versions. Prefix operators return references to the modified object, enabling chaining, while postfix operators return copies of the original value. Unary operators can be implemented as member or non-member functions, with member functions taking no parameters (except the dummy int for postfix). Proper overloading makes custom types behave intuitively and naturally in expressions, improving code readability and usability.