13.7 — Overloading the comparison operators

ALEX AUGUST 28, 2021

In lesson 5.6 -- Relational operators and floating point comparisons, we discussed the six comparison operators. Overloading these comparison operators is comparatively simple (see what I did there?), as they follow the same patterns as we've seen in overloading other operators.

Because the comparison operators are all binary operators that do not modify their left operands, we will make our overloaded comparison operators friend functions.

Here's an example Car class with an overloaded operator== and operator!=.

```cpp
#include <iostream>
#include <string>

class Car
{
private:
    std::string m_make;
    std::string m_model;

public:
    Car(const std::string& make, const std::string& model)
        : m_make{ make }, m_model{ model }
    {
    }

    friend bool operator== (const Car& c1, const Car& c2);
    friend bool operator!= (const Car& c1, const Car& c2);
};

bool operator== (const Car& c1, const Car& c2)
{
    return (c1.m_make == c2.m_make &&
            c1.m_model == c2.m_model);
}

bool operator!= (const Car& c1, const Car& c2)
{
    return (c1.m_make != c2.m_make ||
            c1.m_model != c2.m_model);
}

int main()
{
    Car corolla{ "Toyota", "Corolla" };
    Car camry{ "Toyota", "Camry" };

    if (corolla == camry)
        std::cout << "a Corolla and Camry are the same.\n";

    if (corolla != camry)
        std::cout << "a Corolla and Camry are not the
same.\n";

    return 0;
}
```

The code here should be straightforward.

What about operator< and operator>? What would it mean for a Car to be greater or less than another Car? We typically don't think about cars this way. Since the results of operator< and operator> would not be immediately intuitive, it may be better to leave these operators undefined.

Best practice

Only define overloaded operators that make intuitive sense for your class.

However, there is one common exception to the above recommendation. What if we wanted to sort a list of Cars? In such a case, we might want to overload the comparison operators to return the member (or members) you're most likely to want to sort on. For example, an overloaded operator< for Cars might sort based on make and model alphabetically.

Some of the container classes in the standard library (classes that hold sets of other classes) require an overloaded operator< so they can keep the elements sorted.

Here's a different example overloading all 6 logical comparison operators:

```cpp
#include <iostream>

class Cents
{
private:
    int m_cents;

public:
    Cents(int cents)
 : m_cents{ cents }
 {}

    friend bool operator== (const Cents& c1, const Cents& c2);
    friend bool operator!= (const Cents& c1, const Cents& c2);

    friend bool operator< (const Cents& c1, const Cents& c2);
    friend bool operator> (const Cents& c1, const Cents& c2);

    friend bool operator<= (const Cents& c1, const Cents& c2);
    friend bool operator>= (const Cents& c1, const Cents& c2);
};

bool operator== (const Cents& c1, const Cents& c2)
{
    return c1.m_cents == c2.m_cents;
}

bool operator!= (const Cents& c1, const Cents& c2)
{
    return c1.m_cents != c2.m_cents;
}

bool operator< (const Cents& c1, const Cents& c2)
{
    return c1.m_cents < c2.m_cents;
}

bool operator> (const Cents& c1, const Cents& c2)
{
    return c1.m_cents > c2.m_cents;
}

bool operator<= (const Cents& c1, const Cents& c2)
{
    return c1.m_cents <= c2.m_cents;
}

bool operator>= (const Cents& c1, const Cents& c2)
{
    return c1.m_cents >= c2.m_cents;
}

int main()
{
    Cents dime{ 10 };
    Cents nickel{ 5 };

    if (nickel > dime)
        std::cout << "a nickel is greater than a dime.\n";
    if (nickel >= dime)
        std::cout << "a nickel is greater than or equal to a
dime.\n";
    if (nickel < dime)
        std::cout << "a dime is greater than a nickel.\n";
    if (nickel <= dime)
        std::cout << "a dime is greater than or equal to a
nickel.\n";
    if (nickel == dime)
        std::cout << "a dime is equal to a nickel.\n";
    if (nickel != dime)
        std::cout << "a dime is not equal to a nickel.\n";

    return 0;
}
```

This is also pretty straightforward.

Minimizing comparative redundancy

In the example above, note how similar the implementation of each of the overloaded comparison operators are. Overloaded comparison operators tend to have a high degree of redundancy, and the more complex the implementation, the more redundancy there will be.

Fortunately, many of the comparison operators can be implemented using the other comparison operators:

- operator!= can be implemented as !(operator==)
- operator> can be implemented as operator< with the order of the parameters flipped
- operator>= can be implemented as !(operator<)
- operator<= can be implemented as !(operator>)

This means that we only need to implement logic for operator== and operator<, and then the other four comparison operators can be defined in terms of those two! Here's an updated Cents example illustrating this:

```cpp
#include <iostream>

class Cents
{
private:
    int m_cents;

public:
    Cents(int cents)
        : m_cents{ cents }
    {}

    friend bool operator== (const Cents& c1, const Cents& c2);
    friend bool operator!= (const Cents& c1, const Cents& c2);

    friend bool operator< (const Cents& c1, const Cents& c2);
    friend bool operator> (const Cents& c1, const Cents& c2);

    friend bool operator<= (const Cents& c1, const Cents& c2);
    friend bool operator>= (const Cents& c1, const Cents& c2);

};

bool operator== (const Cents& c1, const Cents& c2)
{
    return c1.m_cents == c2.m_cents;
}

bool operator!= (const Cents& c1, const Cents& c2)
{
    return !(operator==(c1, c2));
}

bool operator< (const Cents& c1, const Cents& c2)
{
    return c1.m_cents < c2.m_cents;
}

bool operator> (const Cents& c1, const Cents& c2)
{
    return operator<(c2, c1);
}

bool operator<= (const Cents& c1, const Cents& c2)
{
    return !(operator>(c1, c2));
}

bool operator>= (const Cents& c1, const Cents& c2)
{
    return !(operator<(c1, c2));
}

int main()
{
    Cents dime{ 10 };
    Cents nickel{ 5 };

    if (nickel > dime)
        std::cout << "a nickel is greater than a dime.\n";
    if (nickel >= dime)
        std::cout << "a nickel is greater than or equal to a
dime.\n";
    if (nickel < dime)
        std::cout << "a dime is greater than a nickel.\n";
    if (nickel <= dime)
        std::cout << "a dime is greater than or equal to a
nickel.\n";
    if (nickel == dime)
        std::cout << "a dime is equal to a nickel.\n";
    if (nickel != dime)
        std::cout << "a dime is not equal to a nickel.\n";

    return 0;
}
```

This way, if we ever need to change something, we only need to update operator== and operator< instead of all six comparison operators!

Quiz time

1. **Add the six comparison operators to the Fraction class so that the following program compiles:**

```cpp
#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 std::ostream& operator<<(std::ostream& out, const Fraction& f1);
};

std::ostream& operator<<(std::ostream& out, const Fraction& f1)
{
 out << f1.m_numerator << '/' << f1.m_denominator;
 return out;
}

int main()
{
 Fraction f1{ 3, 2 };
 Fraction f2{ 5, 8 };

 std::cout << f1 << ((f1 == f2) ? " == " : " not == ") << f2 << '\n';
 std::cout << f1 << ((f1 != f2) ? " != " : " not != ") << f2 << '\n';
 std::cout << f1 << ((f1 < f2) ? " < " : " not < ") << f2 << '\n';
 std::cout << f1 << ((f1 > f2) ? " > " : " not > ") << f2 << '\n';
 std::cout << f1 << ((f1 <= f2) ? " <= " : " not <= ") << f2 << '\n';
 std::cout << f1 << ((f1 >= f2) ? " >= " : " not >= ") << f2 << '\n';
 return 0;
}
```

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

```cpp
#include <cmath>

int gcd(int a, int b) {
    return (b == 0) ? std::abs(a) : gcd(b, a %
b);
}
```

Show Solution

2. **Add an overloaded operator<< and operator< to the Car class at the top of the lesson so that the following program compiles:**

```cpp
#include <algorithm>
#include <iostream>
#include <string>
#include <vector>

int main()
{
  std::vector<Car> cars{
    { "Toyota", "Corolla" },
    { "Honda", "Accord" },
    { "Toyota", "Camry" },
    { "Honda", "Civic" }
  };

  std::sort(cars.begin(), cars.end()); // requires an overloaded operator<

  for (const auto& car : cars)
    std::cout << car << '\n'; // requires an overloaded operator<<

  return 0;
}
```

This program should produce the following output:

```
(Honda, Accord)
(Honda, Civic)
(Toyota, Camry)
(Toyota, Corolla)
```

If you need a refresher on std::sort, we talk about it in lesson 10.4 -- Sorting an array using selection sort.

Show Solution

Next lesson
13.8 Overloading the increment and decrement operators

Back to table of contents

Previous lesson
13.6 Overloading unary operators +, -, and !

Leave a comment... Put C++ code between triple-backticks (markdown style):```Your C++ coc

Name*

Email*

Avatars from https://gravatar.com/ are connected to your provided email address.

Notify me about replies: POST COMMENT

DP N N FOUT

Newest

X

X