9.6 — Overloading the comparison operators

BY ALEX ON OCTOBER 4TH, 2007 | LAST MODIFIED BY ALEX ON MAY 20TH, 2018

Overloading the 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!=.

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

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

public:
    Car(std::string make, 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== c2);
}

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. Because the result of operator!= is the opposite of operator==, we define operator!= in terms of operator==, which helps keep things simpler, more error free, and reduces the amount of code we have to write.

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.

Recommendation: Don't define overloaded operators that don't make 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 with an overloaded operator>, operator<, operator>=, and operator<=:

```
#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);
};

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 nickle(5);

    if (nickle > dime)
        std::cout << "a nickle is greater than a dime.\n";
    if (nickle >= dime)
        std::cout << "a nickle is greater than or equal to a dime.\n";
    if (nickle < dime)
        std::cout << "a dime is greater than a nickle.\n";
    if (nickle <= dime)
        std::cout << "a dime is greater than or equal to a nickle.\n";


    return 0;
}
```

This is also pretty straightforward.

Note that there is some redundancy here as well. operator> and operator<= are logical opposites, so one could be defined in terms of the other. operator< and operator>= are also logical opposites, and one could be defined in terms of the other. In this case, I chose not to do so because the function definitions are so simple, and the comparison operator in the function name line up nicely with the comparison operator in the return statement.

Quiz time

1) For the Cents example above, rewrite operators < and <= in terms of other overloaded operators.

Hide Solution

```
#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);
};

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;
}

// The logical opposite of < is >=, so we can do >= and invert the result
bool operator< (const Cents &c1, const Cents &c2)
{
    return !(c1 >= c2);
}

// The logical opposite of <= is >, so we can do > and invert the result
bool operator<= (const Cents &c1, const Cents &c2)
{
    return !(c1 > c2);
}

int main()
{
    Cents dime(10);
    Cents nickle(5);

    if (nickle > dime)
        std::cout << "a nickle is greater than a dime.\n";
    if (nickle >= dime)
        std::cout << "a nickle is greater than or equal to a dime.\n";
    if (nickle < dime)
        std::cout << "a dime is greater than a nickle.\n";
    if (nickle <= dime)
        std::cout << "a dime is greater than or equal to a nickle.\n";


    return 0;
}
```

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

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

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

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

    for (auto &car : v)
        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)
```

Hide Solution

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

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

public:
    Car(std::string make, 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);
    friend std::ostream& operator<< (std::ostream& out, const Car & c)
    {
        out << '(' << c.m_make << ", " << c.m_model << ')';
        return out;
    }

    // h/t to reader Olivier for this version of the function
    friend bool operator<(const Car &c1, const Car &c2)
    {
        if (c1.m_make == c2.m_make) // If the car is the same make...
            return c1.m_model < c2.m_model; // then compare the model
        else
            return c1.m_make < c2.m_make; // otherwise compare the makes
    }
};
```

```
35
36  bool operator== (const Car &c1, const Car &c2)
37  {
38      return (c1.m_make == c2.m_make &&
39          c1.m_model == c2.m_model);
40  }
41
42  bool operator!= (const Car &c1, const Car &c2)
43  {
44      return !(c1 == c2);
45  }
46
47  int main()
48  {
49      std::vector<Car> v;
50      v.push_back(Car("Toyota", "Corolla"));
51      v.push_back(Car("Honda", "Accord"));
52      v.push_back(Car("Toyota", "Camry"));
53      v.push_back(Car("Honda", "Civic"));
54
55      std::sort(v.begin(), v.end()); // requires an overloaded Car::operator<
56
57      for (auto &car : v)
58          std::cout << car << '\n'; // requires an overloaded Car::operator<<
59
60      return 0;
61  }
```



9.7 -- Overloading the increment and decrement operators


Index


9.5 -- Overloading unary operators +, -, and !

Share this: