9.6 — Overloading the comparison operators

BY ALEX ON OCTOBER 4TH, 2007 | LAST MODIFIED BY ALEX ON JANUARY 23RD, 2020

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!=.

```cpp
#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<=:

```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);
};

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";


    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.

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:

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

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

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

  for (const 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)
```

If you need a refresher on std::sort, we talk about it in lesson **6.4 -- Sorting an array using selection sort**.

Show Solution

9.7 -- Overloading the increment and decrement operators

Index

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

C++ TUTORIAL | PRINT THIS POST

108 comments to 9.6 — Overloading the comparison operators

« Older Comments 1 2

Ged
January 21, 2020 at 9:29 am · Reply

Why do we need these overloaded functions? Tested it without these ones and it worked fine.

```
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);
}
```

nascardriver
January 22, 2020 at 3:51 am · Reply

They're used in the first example of this lesson, but not needed in the quiz.

Ryan
December 10, 2019 at 10:48 am · Reply

Is it recommend or preferable to use references in the parameters for these functions, like:

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

You are not changing the values for the objects, so what's the point in making c1 and c2 references?

nascardriver
December 11, 2019 at 5:38 am · Reply

We use references for speed. We use `const` because we don't modify them. See lesson 7.3.

David
September 11, 2019 at 8:37 pm · Reply

The mini quiz was the first one iv'e had any real difficulty with and ultimately gave up and peeked at the answer. Unless I missed something the last time we touched std::sort it was essentially here is sort it sorts arrays when given a beginning and end. I don't remember going over how it actually worked even looking it up I could only really find examples on how to use it which all essentially said the same thing. Give it a starting point, and ending point and a third optional parameter that was how it would compare things.

Looking at the answer now it makes sense but I feel like if something as simple as:

//Note* std::sort by default compares two values and places whichever element it finds the smallest closer to the beginning of an array.

If that little bit of info was there it would have been a lot easier to work backwards how to get std::sort to do what you needed it to do. The second half of the mini quiz was fine though.

Alex
September 16, 2019 at 2:56 pm · Reply

I added a link to the lesson where we discuss std::sort and added a little context there.
Thanks for the feedback.

DecSco
July 31, 2019 at 2:47 am · Reply

Is there any disadvantage in defining the operator as

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

?
Readability aside, maybe.

nascardriver
July 31, 2019 at 4:15 am · Reply

It's exactly the same.

masterOfNothing
July 13, 2019 at 9:10 am · Reply

Hi, in the quiz where we had to overwrite operator< (Car, Car) for Car class, I found out this worked with std::sort function as well:

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

nascardriver
July 13, 2019 at 9:34 am · Reply

You didn't test with enough data

```
c1: Porsche Cayman
c2: Audi R8
Correct: false
You:     true
```

masterOfNothing
July 15, 2019 at 9:19 am · Reply

That I did not. Thanks for pointing it out.

polak
July 3, 2019 at 12:15 am · Reply

The two-way comparison operator expressions have the form:

lhs < rhs (1)
lhs > rhs (2)
lhs <= rhs (3)
lhs >= rhs (4)
lhs == rhs (5)
lhs != rhs (6)

1) Returns true if lhs is less than rhs, false otherwise.
2) Returns true if lhs is greater than rhs, false otherwise.
3) Returns true if lhs is less than or equal to rhs, false otherwise.
4) Returns true if lhs is greater than or equal to rhs, false otherwise.
5) Returns true if lhs is equal to rhs, false otherwise.
6) Returns true if lhs is not equal to rhs, false otherwise.
In all cases, for the built-in operators, lhs and rhs must have either
arithmetic or enumeration type
pointer type
Then there are Arithmetic comparison operators, pointer comparison operators, three-way comparison...

Just a little note for y'all :)

Anthony
May 24, 2019 at 11:34 am · Reply

Hi,

What is the correct form of the == operator where instead of:

```
MyObject obj1, obj2;
if (obj1 == obj2) { ... }
```

we have:

```
MyObject obj;
if (obj) { ... }
```

Imagine that MyObject has a pointer member variable m_ptr, and we want to test whether this pointer has been initialised. I'm thinking along the lines of:

```
class MyObject {
private:
    int m_ptr {nullptr};
public:
    ...
    bool operator== () { ... }  // <-- what would the formal parameter be?
}
```

nascardriver
May 25, 2019 at 12:45 am · Reply

You can overload typecast operators (Lesson 9.10).
They can be used to write a custom conversion to bool.

```
operator bool(void)
{
  return (this->m_ptr != nullptr);
}
```

Anthony
May 25, 2019 at 2:52 am · Reply

Thank you nascardriver :) I was, as is fairly tyoical, barking up entirely the wrong tree

Tommy
April 29, 2019 at 10:18 pm · Reply

Hello, forgive the constant confirming

Because we are only overloading the operators to manage car classes, does that mean that std::sort can compare strings and sort accordingly? Does it do this by taking the first character's ASCII code and converting it into an int?

nascardriver
April 30, 2019 at 3:31 am · Reply

@std::sort can sort everything that has an @operator<.
@std::string::operator< compares strings char by char, until the strings differ. char is an

integral type, there's no need for a conversion.

Jack
January 7, 2019 at 9:36 am · Reply

Q2. my version of overloaded operator< function.

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

nascardriver
January 8, 2019 at 2:07 am · Reply

Hi Jack!

Your code will work in most cases. Not all

c1: hond acivic
c2: honda civic

Correct answer: c1 < c2
Your answer: !(c1 < c2)

On top of that, your code constructs 2 temporary strings, which is slow.

Alex
January 3, 2019 at 5:22 pm · Reply

In case anyone is interested. I used the compare function of string and it worked.

```
bool operator< (const Car &c1, const Car &c2)
{
    int make_comp = c1.m_make.compare(c2.m_make);
    int model_comp = c1.m_model.compare(c2.m_model);
    return make_comp == 0 ? model_comp < 0 : make_comp < 0;
}
```

nascardriver
January 8, 2019 at 2:09 am · Reply

Hi Alex!

* Initialize your variables with uniform initialization. You used copy initialization.
* You're comparing the models even if @model_comp is not used.

Marcos O.
December 17, 2018 at 9:34 am · Reply

Hi!

Regarding Quiz(2), in the solution why are the == and != operators overloaded? Aren't the values they compare already resolved to pseudo-fundamental strings when operator < is used? Also wouldn't it be preferable for operator< to be a member function of Car as the left most operand is of type Car?

My solution to Q(2):

```
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 std::ostream& operator<<(std::ostream &out, const Car &car);
    bool operator<(const Car &c2);
};

std::ostream& operator<<(std::ostream &out, const Car &car)
{
    out << "(" << car.m_make << ", " << car.m_model << ")";

    return out;
}

bool Car::operator<(const Car &c2)
{
    if (m_make == c2.m_make)
        return m_model < c2.m_model;
    else
        return m_make < c2.m_make;
}

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());

    for (auto &car : v)
        std::cout << car << '\n';

    return 0;
}
```

nascardriver

December 17, 2018 at 10:01 am · Reply

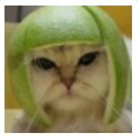
> why are the == and != operators overloaded?
They're not used in this quiz, I don't know why Alex added them. It's worth mentioning that C++ doesn't offer default operator== and operator!=. If you want to compare objects of your class, you need to define those operators manually.

> Aren't the values they compare already resolved to pseudo-fundamental strings when operator < is used?

I'm not sure what you mean. @Car::operator< doesn't make use of @Car::operator== or @Car::operator!=. It used @std::string's operators.

> Also wouldn't it be preferable for operator< to be a member function of Car as the left most operand is of type Car?
Lesson 9.4 says not to do so, because with a non-member function it'd be easier to swap the order of the parameters, which doesn't matter in this case.

> My solution
I'm skipping @main, because you copied it.
* Line 4, 5, 9: Initialize your variables with uniform initialization. @m_make and @m_model will be initialized anyway, but if you initialize everything, you won't forget to when you need it.



Alex
December 17, 2018 at 1:59 pm · Reply

The overloaded operator== and operator!= were in the original Car program that the quiz question is extending. Even though they're not used, there's no harm in having them there since it's just a copy/paste from above.

Marcos O.
December 17, 2018 at 2:21 pm · Reply

Ah k, thanks Alex!

Marcos O.
December 17, 2018 at 2:20 pm · Reply

>...C++ doesn't offer default operator== and operator!=. If you want to compare objects of your class, you need to define those operators manually.
If c++ doesnt offer default == or != then how are they used for

```
m_make == c2.m_make
```

where the operator isn't defined or a simpler case

```
if (int x == int y)
```

Int in the above example is just to indicate type of x and y, not be correct code.
And what do you define as "default", is the std library and string library "default"? Is it because we arent comparing the objects of defined class (which would require overloading operator) instead the string objects within?

>It used @std::string's operators
Thank you for confirming, thats exactly what I meant, I just wanted to make sure.

>Lesson 9.4 says not to do so, because with a non-member function it'd be easier to swap the order of the parameters, which doesn't matter in this case.
Isnt this case specifically to compare an object of the same type with an object of our defined type as reference? Isn't a case where your defined object is the reference one of the specific times to use a member function?

>* Line 4, 5, 9: Initialize your variables with uniform initialization
Line 4, 5: I thought we werent supposed to initialize a variable both at definition and construction. Is that only if initialized with a value?

Line 9: VS2013 doesn't seem to like uniform initialization and often throws me errors when I try. Do you have any tips or suggestions around these errors?

Thank you for your help. I would have further appreciated if you left your assumption about me copying @main out.

nascardriver
December 18, 2018 at 7:47 am · Reply

> And what do you define as "default"
Types you write yourself. int, double and other native types have operators. @std::string is a class, it has those operators, whoever wrote the @std::string class that's being used on your system wrote those operators.

> Isnt this case specifically to compare an object of the same type [...]
Sorry, I'm having a hard time trying to understand what you're asking.

> I thought we werent supposed to initialize a variable both at definition and construction
I don't know what you're referring to. Mind sharing the lesson?
Initialize all variables. If possible, to a specific (0) value. @std::string has a default constructor, so just use empty curly brackets.

> VS2013 doesn't seem to like uniform initialization and often throws me errors when I try
If this happens sometimes, but not always, you're doing something wrong (Or there's a compiler bug, which is unlikely). If it never allows uniform initialization, upgrade your compiler.

> I would have further appreciated if you left your assumption about me copying @main
Initialize @v instead of manually pushing all elements. Also, uniform initialization.

```
std::vector<Car> v{
  { "Toyota", "Corolla" },
  { "Honda", "Accord" },
  { "Toyota", "Camry" },
  { "Honda", "Civic" },
};
```

Khang
December 11, 2018 at 6:45 am · Reply

I think the right spelling is "nickel" not "nickle".

Alex
December 11, 2018 at 12:31 pm · Reply

Typos fixed. Thanks for pointing that out.

Michael Stef
September 24, 2018 at 11:58 am · Reply

i don't understand in second quiz some answers make the program crash such as doing this:
return (c1.m_make < c2.m_make) || (c1.m_model < c2.m_model)
if one or the other is alphabetically out of order sort it , even if it won't do exactly what is required.
does this have to do something with the iterators ?

nascardriver
September 24, 2018 at 11:22 pm · Reply

Hi Michael!

Your @operator< is invalid, because it allows car A to be smaller than car B, but at the same time B is smaller than A.

```
Car a{ "A", "B" };
Car b{ "B", "A" };

std::cout << (a < b) << std::endl; // true
std::cout << (b < a) << std::endl; // true
```

Depending on the implementation of @std::sort, this could cause infinite loops, exceptions or crashes.

« Older Comments 1 2