

Operator Overloading

Steven Zeil

November 4, 2013

Contents

1	Operators	2
1.1	Operators are Functions	2
2	I/O Operators	4
3	Comparison Operators	8
3.1	Equality Operators	14
3.2	Less-Than Operators	16

1 Operators

Lots of Operators

In C++, almost every operator in the language can be declared for our own classes.

- In most programming languages, we can write things like “x+y” or “x<y” only if x and y are integers, floating point numbers, or some other pre-defined type in the language.
- In C++, we can add these operators to any class we design, if we feel that they are appropriate.
 - These include + - * / | & < > <= >= = == != ++ - -> += -= *= /=
 - For example, the `std::string` class declares these for strings: + < > <= >= = == != +=

.....

1.1 Operators are Functions

Operators are Functions

- Each operator is actually a shorthand for a function named `operator?` where the ? is replaced by the actual operator symbol.
- The function takes the same number and type of parameters as does the operator
- The compiler translates the infix expressions into the equivalent function calls

.....

Examples: operators as functions

- If you write `a + b*(-c)`, that's actually just a shorthand for

```
operator+(a, operator*(b, operator-(c)))
```

- and if you write

```
testValue = (x <= y);
```

that's actually a shorthand for

```
operator=(testValue, operator<=(x, y);
```

.....

Declaring Operators

Understanding that shorthand, we can declare our own operators by giving the appropriate operator function name:

```
class BidCollection {  
    :  
    void addInTimeOrder (const Bid& value);  
    // Adds this bid into a position such that  
    // all bids are ordered by the time the bid was placed  
    //Pre: getSize() < getMaxSize()  
  
    void operator+=( const Bid& value)  
        {addInTimeOrder(value);}  
    :  
};
```

Then

```
bids.addInTimeOrder(b);
```

and

```
bids += b;
```

would do the same thing

.....

Keep it Natural

- It's easy to abuse operator overloading and simply make things confusing.
 - E.g., Don't use operator+ unless your class really does something "addition-like"
 - * string concatenation
-

Most Commonly Overloaded Operators

There are, however, 3 groups of operators that are very commonly overloaded

- I/O operators << >>
 - Comparison operators < ==
 - The assignment operator =
-

2 I/O Operators

I/O Operators

Perhaps the most commonly declared operators

- We have already seen that there is good reason for every class to provide an output function.
 - But the most common form for this is the operator <<
-

Providing the Output Operator

For example, our *Bid* class already looks like:

```
class Bid {
    :
public:
    Bid (std::string bidder, double amt,
        std::string item, Time placedAt);

    Bid ();

    // Access to attribute
    std::string getBidder() const {return bidderName;}
    double getAmount() const {return amount;}
    std::string getItem() const {return itemName;}
    Time getTimePlacedAt() const {return bidPlacedAt;}

    // Print a bid
    void print (std::ostream& out) const;
};
```

All we need to do is add:

```
inline
std::ostream& operator<< (std::ostream& out, const Bid& b)
{
    b.print(out);
    return out;
}
```

.....

Lots of Ops

The whole program, with output operators:

- The Bid ADT: [auction0ut/bids.h](#) and [auction0ut/bids.cpp](#)

- The Bidder ADT: `auctionOut/bidders.h` and `auctionOut/bidders.cpp`
- The Item ADT: `auctionOut/items.h` and `auctionOut/items.cpp`
- The Time ADT: `auctionOut/time.h` and `auctionOut/time.cpp`
- The BidCollection ADT: `auctionOut/bidcollection.h` and `auctionOut/bidcollection.cpp`
- The BidderCollection ADT: `auctionOut/biddercollection.h` and `auctionOut/biddercollection.cpp`
- The ItemCollection ADT: `auctionOut/itemcollection.h` and `auctionOut/itemcollection.cpp`

.....

Using the Output Op

Note how much cleaner the debugging output statements look.

Instead of

```
if (bid.getAmount() <= bidder.getBalance())
{
    highestBidSoFar = bid.getAmount();
    winningBidderSoFar = bid.getBidder();
    cerr << "Best bid so far is ";
    bid.print (cerr);
    cerr << " from ";
    bidder.print (cerr);
    cerr << endl;
}
```

.....

Debugging With Output Op

we now can write

```
if (bid.getAmount() <= bidder.getBalance())
{
    highestBidSoFar = bid.getAmount();
    winningBidderSoFar = bid.getBidder();
    cerr << "Best bid so far is "
        << bid << " from "
        << bidder << endl;
}
```

.....

Return Values

The << operator must return the stream it is applied to.

- That's why we can write things like `cout << a << b`;
- The compiler treats this as

```
(cout << a) << b;
```

- What is the << b applied to?
- To the value returned by `(cout << a)`!

- You can also view this as

```
operator<<(operator<<(cout, a), b)
```

.....

Fine Points

- These are not member functions, so they cannot access private data directly
 - You can get around this by using friend declarations described in the text.

- Programming input operators is less common, but very similar to output operators.

.....

3 Comparison Operators

Comparison Operators

Almost every class should provide == and < operators.

- Many data structures and functions in the standard library assume these are available.
- So do some of the ones we have developed, e.g., in [auctionComp/arrayUtils.h](#)

.....

Comparison Operators for Time

- We declare the comparison ops the same way we do other operator functions
- Time is a simple case, since it has only one data member

```
#ifndef TIMES_H
#define TIMES_H

#include <iostream>

/**
 * Times in this program are represented by three integers: H, M, & S, representing
 * the hours, minutes, and seconds, respectively.
 */
```



```
class Time {
public:
    // Create time objects
    Time(); // midnight
    Time (int h, int m, int s);

    // Access to attributes
    int getHours() const;
    int getMinutes() const;
    int getSeconds() const;

    // Calculations with time
    void add (Time delta);
    Time difference (Time fromTime);

    /**
     * Read a time (in the format HH:MM:SS) after skipping any
     * prior whitespace.
     */
    void read (std::istream& in);

    /**
     * Print a time in the format HH:MM:SS (two digits each)
     */
    void print (std::ostream& out) const;

    // Comparison operators
```



```
bool operator== (const Time&) const;
bool operator< (const Time&) const;

private:
    // From here on is hidden
    int secondsSinceMidnight;
};

inline
std::ostream& operator<< (std::ostream& out, const Time& t)
{
    t.print(out);
    return out;
}

inline
bool operator!= (const Time& t1, const Time& t2)
{
    return !(t1 == t2);
}

inline
bool operator> (const Time& t1, const Time& t2)
{
    return t2 < t1;
}

inline
```



```
bool operator<= (const Time& t1, const Time& t2)
{
    return !(t1 > t2);
}
```

```
inline
```

```
bool operator>= (const Time& t1, const Time& t2)
{
    return !(t1 < t2);
}
```

```
#endif // TIMES_H
```

```
#include "time.h"
```

```
#include <iomanip>
```

```
using namespace std;
```

```
/**
```

```
 * Times in this program are represented by three integers: H, M, & S, representing
 * the hours, minutes, and seconds, respectively.
```

```
*/
```

```
    // Create time objects
```

```
Time::Time() // midnight
```

```
{
    secondsSinceMidnight = 0;
}
```



```
Time::Time (int h, int m, int s)
{
    secondsSinceMidnight = s + 60 * m + 3600*h;
}

// Access to attributes
int Time::getHours() const
{
    return secondsSinceMidnight / 3600;
}

int Time::getMinutes() const
{
    return (secondsSinceMidnight % 3600) / 60;
}

int Time::getSeconds() const
{
    return secondsSinceMidnight % 60;
}

// Calculations with time
void Time::add (Time delta)
{
    secondsSinceMidnight += delta.secondsSinceMidnight;
}

Time Time::difference (Time fromTime)
```



```
{
    Time diff;
    diff.secondsSinceMidnight =
        secondsSinceMidnight - fromTime.secondsSinceMidnight;
}

/**
 * Read a time (in the format HH:MM:SS) after skipping any
 * prior whitespace.
 */
void Time::read (std::istream& in)
{
    char c;
    int hours, minutes, seconds;
    in >> hours >> c >> minutes >> c >> seconds;
    Time t (hours, minutes, seconds);
    secondsSinceMidnight = t.secondsSinceMidnight;
}

/**
 * Print a time in the format HH:MM:SS (two digits each)
 */
void Time::print (std::ostream& out) const
{
    out << setfill('0') << setw(2) << getHours() << ':'
        << setfill('0') << setw(2) << getMinutes() << ':'
        << setfill('0') << setw(2) << getSeconds();
}
```



```
// Comparison operators
bool Time::operator< (const Time& time2) const
{
    return secondsSinceMidnight < time2.secondsSinceMidnight;
}

bool Time::operator==(const Time& time2) const
{
    return secondsSinceMidnight == time2.secondsSinceMidnight;
}
```

.....

3.1 Equality Operators

Equality Operators for Compound Data

- You have to think about what you want == to *mean* for your ADT.
 - May depend on the application

.....

Common: Compare Every Member

For ADTs with multiple data members, we often expect every "significant" data member to match. E.g.,

```
class Bid {
    std::string bidderName;
```

```
double amount;  
std::string itemName;  
Time bidPlacedAt;  
:
```

can be compared like this

```
bool Bid::operator== (const Bid& b) const  
{  
    return bidderName == b.bidderName  
        && amount == b.amount  
        && itemName == b.itemName  
        && bidPlacedAt == b.bidPlacedAt;  
}
```

.....

Highly Variable Data

Fields that change frequently may or may not be significant:

```
bool Bidder::operator== (const Bidder& b) const  
{  
    return name == b.name; // ignore balance  
}
```

- Two *Bidder* records describe the same person if the names match, even if their bank balances differ.
- Some would say that the name is a *key* that identifies the *Bidder*,

.....

Equality for ADTs with Arrays

```

bool BidCollection::operator== (const BidCollection& bc) const
{
    if (size == bc.size) ❶
    {
        for (int i = 0; i < size; ++i) ❷
            if (!(elements[i] == ❸bc.elements[i]))
                return false;
        return true;
    }
    else
        return false;
}

```

- ❶ Do the quick & easy check first (size)
- ❷ Then insist that each array element must match
 - ❸ Note that this uses the Bid::operator== we developed above
- I did not compare the MaxSize values because that is not, logically, part of the value of the collection.

.....

3.2 Less-Than Operators

< is not always “less than”

- What does < *mean* for compound data structures?
 - Can a bid, for example, be said to be "less than" another bid? Bidders? Items?
 - < is used mainly to put things into a sorted order
 - So think of it as meaning "comes before", not "is smaller than"

.....

Less-than on Compound Data

- For ADTs with multiple data members, we usually model operator< on *lexicographic* (alphabetic) order rules.
 1. When comparing two strings, we compare the first two characters
 2. If they are different, we immediately decide that one string is < than the other.
 3. If they are equal, we move on to the next character and repeat steps 2-3
- E.g., "abc" < "bbc", "cbc" > "bbc", "abc" < "abd", "abc" > "aba"

.....

Extending Lexicographic Order to Non-Strings

Compare members until we find two that are not equal:

```
bool Bid::operator< (const Bid& b) const
{
    if (bidPlacedAt < b.bidPlacedAt)
        return true;
    else if (b.bidPlacedAt < bidPlacedAt)
        return false;

    if (bidderName < b.bidderName)
        return true;
    else if (b.bidderName < bidderName)
        return false;

    if (itemName < b.itemName)
        return true;
    else if (b.itemName < itemName)
        return false;

    if (amount < b.amount)
```

```

    return true;
else if (b.amount < amount)
    return false;

return false;
}

```

.....

Less-than for ADTs with Arrays

We extend the same idea to arrays of data:

```

bool BidCollection::operator< (const BidCollection& bc) const
{
    if (size == bc.size)           ❶
    {
        for (int i = 0; i < size; ++i)  ❷
        {
            if (elements[i] < bc.elements[i])  ❸
                return true;
            else if (bc.elements[i] < elements[i])  ❹
                return false;
            // else keep going
        }

        // All of the elements are equal
        return false;           ❺
    }
    else
        return size < bc.size;  ❶
}

```

❶ Do the quick & easy check first (size)

- ② Then run through the arrays until we find two elements that are not equal (③)
- ④ If all the array elements are equal, then this is not less than the other (they are equal).

.....

Adding Both == and <

We often have choices in how to implement these two.

- They *must* be consistent.

For any two values x and y, *exactly one* of the following conditions must be true:

- x == y
- x < y
- y > x

- A good rule of thumb: every data member checked by == should be checked by <, and vice-versa

.....

Why Just Two?

- Why do we often just provide == and <?
- It's easy to define the others in terms of these two.
 - `auctionComp/time.h`
- in fact, the C++ std library has function templates that already do this.
 - If your file `#includes` the header `<utility>` and has the statement

```
using namespace std::rel_ops;
```

then the operators `!=`, `>`, `<=`, and `>=` will be provided based on your own `==` and `<`.

.....



Taking Advantage of the Relational Ops

If we make a habit of providing these, we can often use "canned" versions of our algorithms instead of needing to write specialized versions.

Compare:

- [auctionComp/arrayUtils.h](#)
- The customized version for ItemCollection:

```
void ItemCollection::addInTimeOrder (const Item& value)
// Adds this item into a position such that
// all items are ordered by the time the item' auction ends
//Pre: getSize() < getMaxSize()
{
    // Make room for the insertion
    int toBeMoved = size - 1;
    while (toBeMoved >= 0 &&
           value.getAuctionEndTime().noLaterThan(elements[toBeMoved].getAuctionEndTime())) {
        elements[toBeMoved+1] = elements[toBeMoved];
        --toBeMoved;
    }
    // Insert the new value
    elements[toBeMoved+1] = value;
    ++size;
}
```

- But after providing relational ops on Items:

```
void ItemCollection::addInTimeOrder (const Item& value)
// Adds this item into a position such that
// all items are ordered by the time the item' auction ends
//Pre: getSize() < getMaxSize()
{
```

```
addInOrder (elements, size, value);  
}
```

.....