

## Operator Overloading

Chapter 5

## Operator Overloading

- Polymorphism for operators (e.g. +)
- Most languages provide *limited* support
  - Think of numbers
- What does *a << 1* mean ?
  - Shift an integer left by 1 bit ?
  - Send an integer to an output stream?
  - ... ?
- C++ provides very general capabilities

## You Have Seen Op Overloading

- << and >> for I/O operations
- = assignment operator
- ++Iterator (will see this in STL)
- [] for accessors/mutators
- + for strings

## Motivation for Op Overloading

- May make code easier to read/understand
  - e.g. + vs add() method
- Can write generic algorithms that work on both primitives *and* objects
- Sorting requires comparisons, so it is natural to use <, ==, etc
  - In java we need to use compareTo() for objects, but < for primitives
  - By overloading < for objects, can write once

## Overloading Problems

- Writer can get “too clever”
  - Company + Person (hiring ?)
  - Company – Person (firing ?)
  - Person + Person (couple ?)
- Does use add clarity?
- Are normal semantics maintained?
  - $a + b == b + a$  ?
  - think about strings

## What Can be Overloaded

- Assignment: =
- Mathematical + - \* / % ++ -- += -= \*= /= %=
- Bit operations: << >> <<= >>= ! ~ | &
- Logical: < <= > >= == !=
- Misc: [] -> () new delete new[] delete[]
- Dubious: || &&
- **NO**: . .\* ? sizeof

## Even if you overload

- Can't define new operators (e.g. \*\* for exponentiation)
- Can't change precedence of operator, so overloading ^ for exponentiation can fail for:  $a*b^c$  (evaluated as  $(a*b)^c$ )
- Arity, (i.e. unary or binary) can not be changed.

## How does it work ?

- Overloaded operator is simply a function with the name operatorXXX
- Compiler is responsible for translating the operator XXX to a function call to operatorXXX
- e.g.  $object1 + object2$  **may** get translated to  $object1.operator+(object2)$ ;

## Where to define function

- Member function
  - Has access to internal data
  - Seems “natural” for objects
- External function
  - May be more flexible
  - Can be a ***friend***
  - Can use access methods (perhaps inlined)

## A Simple Example

```
class Rational
{
public:
    Rational (int numerator, int denominator = 1) :
        numer(numerator), denom(denominator) { }
private:
    int numer;
    int denom;
}
```

## Add I/O Operations

- We would like to be able to “print” a value
- Member function
- But, this doesn’t chain
  - e.g. `cout << "The rational value is " << aRational`
  - Why not?
  - Solution – change the ostream class (not!)

```
class Rational {
public:
    void operator << (ostream& out = cout) const {
        out << numer << '/' << denom;
    }
};
```

## Non Member Function

- Declare as a non member function with ***two*** parameters
- Result now chains, but need access to private data members

```
ostream& operator<< (ostream out, const Rational& r) {
    out << r.??? << '/' << r.???;
    return out;
}
```

## Combined approach

- Text shows a combined approach
- Provide a member function **and** an overloaded operator

```
class Rational {
public void print (ostream& out) const {
    out << numer << '/' << denom;
}
}
ostream operator<< (ostream out, Rational r) {
    r.print(out);
    return out;
}
```

## Lets Implement \*

```
class Rational {
public:
    const Rational& operator* (const Rational& rhs) const {
        // details left to reader
    }
};
```

## Wasn't that nice!

- We can now write:
  - Rational two(2);
  - Rational threeFifths(3, 5);
  - Rational product = two \* threeFifths;
  - Rational p2 = threeFifths \* 2; // neat!
  - Rational p3 = 2 \* threeFifths; // boo!

## External Function to the Rescue

```
const Rational& operator* (const Rational& lhs,
                          const Rational& rhs) {
    // how does this solve the problem ?
}
```

## Or, If Construction is Costly

```
const Rational& operator* (int lhs, const Rational& rhs)
const Rational& operator (const Rational& lhs, int rhs)
const Rational& operator (const Rational& lhs, const Rational rhs)
```

## Why declare operator\* as returning const?

- Detect errors such as  **$a*b = c$**
- Who would ever write code like this?
- But consider ***if*  $a*b = c$**  when what was intended was ***if*  $a*b == c$**

## Interesting const aside

- The Rational << method was defined to work on a const Rational (after all, printing it shouldn't change the value, should it)
- But what if we want to print the "reduced" form of a number (i.e.  $\frac{1}{2}$  instead of 8/16) ?
- Now, print may choose to ***change*** the internal representation to the reduced form and print that. Is it still const?

## C++ const

- C++ uses bitwise const testing (i.e. if none of the bits change, you haven't changed the object – is this true?)
- To solve the "reduce" problem, C++ allows ***mutable*** keyword which allows member variables to change without violating const
- Objects with pointer can be bitwise const without being logically const!

## Operator = implementation

- Remember, if you don't write it the compiler will! (What is the default?)
- Beware of resource leaks
  - Can be a problem if reference heap
- Beware of self assignment
  - Probably won't write `a = a;`
  - But, one may write `a[i] = a[j];`
  - But, one may write `*px = *py;`

## Overloading ++ --

- These have prefix, postfix form
- Two notations used to distinguish

`operator++()` // defines the prefix form

`operator++(int)` // defines the postfix form (dummy parameter not used)

## Overloading logical operators

- Essentially equivalent to the `.equals()` method and `Comparable` interface in Java
- STL often works with only the `<` operator, since other operations are expressible in terms of `<` (i.e. `a <= b` is `!(b < a)`)
- But, just defining `<` does **not** mean compiler will generate the rest for you. If you want them, you must implement them. (True of all operators, + implies **nothing** about +=)

## == vs .equals()

- Java provides two things which seemingly do the same thing
- The typical C++ methodology is simply to overload `==`
- Having two forms differentiates the concept of equality from equivalence
- But, in many cases the differentiation may not be significant
- When might it make a difference?

## Comparing + and +=

- If + is overloaded, what does  $a = b + c$  imply?
- What does  $a = a + c$  imply ?
- Compare that to  $a += c$

## Overloading []

- Adds notational convenience to collections of data
- More convenient to write  $a[i]$  than  $a.get(i)$
- Allows convenient access such as  $grade["fritz"] = 50$ ; (i.e. non numeric index)

## Using () instead of []

- Consider a two dimensional array class
- Makes operations like  $a*b$  very natural
- But access to an individual element requires  $a[i][j]$ , thus, perhaps, exposing an internal data structure (why?)
- Can overload  $()$  – the function call operator to allow writing  $a(i, j)$  instead. What has been gained?

## Overloading type conversion

- May need to convert one type to another
- Think of expression:  $(int) someDouble$
- Would like to write:  $pi + (double) someRational$

```
class Rational {
public:
    double operator double() {
        return static_cast<double>(numer / denom);
    }
};
```

## auto\_ptr

- An *auto\_ptr* is an object that acts just like a pointer **except** it automatically deletes what it points to when it (the *auto\_ptr*) is destroyed.
- A template class that overrides pointer operators (e.g. \* and ->) to reduce programmer errors
- STL provides library of “smart” pointer to help programmer handle memory management and prevent leaks, etc

## Summary

- Define meaning of operators applied to non primitive types
- Can be member/non-member function
- Typically override = logical (especially ==), I/O operators
- Overriding [] will often have both accessor and mutator form (differ in return type)
- ++ -- can be both prefix, postfix