Operator Overloading and Friends

CSC 340

3/2/2016

Overview

- **Operator Overloading**
- Rules for Operator Overloading
- Friend Functions and Classes

Operator Overloading

- The ability to define a new meaning for the existing operators when applied to a new class or data type
- Makes objects appear to be as similar as possible to primitives
- Useful and convenient for numerical class types
  - Real numbers, complex numbers, etc.
- Not a Java feature

Example

- The vector ADT is designed to store a mathematical vector (i.e. a one row/column matrix)
- Common operations defined over vectors include
  - multiplication with another vector
    v1 * v2
  - multiplication with a scalar constant
    v1 * a
  - addition
    v1 + v2
  - substraction
    v1 - v2
  - negation
    -v1

Example

- We can overload these operators
  - As a member function of our vector class
  - As a standalone function outside of the vector class
  - As a friend, non-member function of the vector class

What's a member function?

Basics

- Operator overloading is similar to function overloading - the name of the function is the `operator` keyword followed by the operator
  operator <symbol of operator>
  operator +(…)
- Most operators can be overloaded, including arithmetic and comparison operators, and the "get/put" operators for I/O
- Not every operator can be overloaded (for example, ::)
- Not every operator can be overloaded as a member function
- Not every operator is recommended to be overloaded

Examples

- Binary operators
  - v1 * v2, v1 - v2
- Unary operators
  - -v1, v1++, ++v1
- Insertion (“put”) and extraction (“get”) operators
  - <<, >>

Overloading << and >>

- The “put” operator is a binary operator
  - The first operand is the output stream
  - The second operand is the value following the <<

`cout`	`<<`	`“Hi, World”;`
Operand 1	Operator	Operand 2

What Does << Return?

- Because << is a binary operator
  ```
  cout << “v1=“ << v1 << endl;
  ```
- We discussed this before - “chaining”
  ```
  (( cout << “v1” ) << v1 ) << endl;
  ```
- To effect this behavior, we must return cout

Overloaded << Declaration

- The format for an overloaded operator (similar to an overloaded function)
  `<return_type> operator <operator> (<parameter_list>)`
- What should the declaration look like for overloading the output of a vector object?
  - What is the return type?
  - What is the operator?
  - What are the parameters (operands)?

10

return type is whatever the type of cout is (it's ostream& - a reference to an output stream object)
operator is <<
operands are ostream&, const vector& (why const?)

Overloaded << Declaration

- All together, looks something like this (we'll talk about friends in a minute)

```
class Vector {
  public:
    friend ostream& operator << (ostream& outs, const vector& v);
  // etc…
}
```

- The & means a reference is returned
  - So far we've only returned values
- The value of a stream object is not so simple - could be an entire file, the keyboard, or the screen.
- We want to return the stream itself, not the value of the stream (thus the & - a reference to the stream, not its value)

11

Overloading >>

- Similar to <<

```
istream& operator >>(istream& ins, vector& v1)
{
  cout << "type in the size of the vector";
  ins >> v1.size;
}
```

- Why isn't this constant?
- Note that this is not recommended implementation!! (Why not?)

12

Because it will mutate most likely - we're reading something into the vector…
Because now you will always output when you are attempting to read something into a vector with this operation

Vector with Operator Overloading

- Let's implement a vector real quick:
  - Vector is an ordered pair of points - we'll just create a Vector class that begins at origin, end at Point p(a, b)
  - Scalar product kv = (ka, kb)
  - Vector addition u + v = (ua + va, ub + vb)
  - Vector subtraction u - v = (ua - va, ub - vb)

Sample code (not in separate compilation units):

```
#include <iostream>

using namespace std;

struct Point {
  int x, y;
};

class Vector {
  public:
    Vector( int x, int y ) {
      point.x = x;
      point.y = y;
```

Overview

- Operator Overloading
- **Rules for Operator Overloading**
- Friend Functions and Classes

Rules for Operator Overloading

- At least one argument of an overloaded operator must be of a class type
- An overloaded operator can be a friend of a class (more on this in a minute)
- New operators can not be created
- The number of arguments for an operator can not be changed
- The precedence and associativity of an operator can not be changed
- Logic operators such as && and `||` are not recommended to be overloaded
- The following operators can not be overloaded:
  `., ::, *, ?:, sizeof()`

Why does Java not support?

- Avoid programmers from abusing a good feature
- Operator overloading should never be used in place of named member functions when the operators do not provide intuitive semantics

Overview

- Operator Overloading
- Rules for Operator Overloading
- **Friend Functions and Classes**

Friend Functions

- Friend functions are not members of a class, but can access private member variables of the class
- Declared using the keyword `friend` in the class definition
  - Not a member function
  - Ordinary function with extraordinary access to data members of a class
- In previous examples, why were << and >> operators friends?

So they could get access to private members of the vector class for output and input

Friend Declaration Syntax

- Note the use of the friend keyword (followed by a normal function declaration: return type, name, parameter list)

```
class class_name
{
  public:
    friend function_one_declaration;
    // etc.
}
```


Using Friends

- A friend function is declared as a friend in the class definition
- A friend function is defined as a nonmember function in the implementation file without using the :: operator
- A friend function is called without using the . operator

Are Friends Needed?

- Friend functions can be written as non-friend functions using the normal accessor and mutator functions that should be part of a class
- The code of a friend function is simpler and more efficient

Choosing Friends

- How do you know when a function should be a friend or a member function?
  - In general, use a member function if the task performed by the function involves only one object
  - In general, use a nonmember function if the task performed by the function involves more than one object