Operator Overloading

Basics

- Is a form of function overloading (i.e., they are functions named operator where is an overloadable operator)
- Does not change the meaning of any operator for an intrinsic data type
- Cannot alter the precedence or associativity of an operator
- Cannot change the number of arguments
- Cannot create a new operator (e.g., **)
- Overloaded operators should be used intuitively (e.g., in a way similar to the original meaning)

Number of Arguments

<table>
<thead>
<tr>
<th>Member vs. non-member</th>
<th>Unary</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Member*</td>
<td>0</td>
<td>1**</td>
</tr>
<tr>
<td>Non-member</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

* Member functions, including overloaded operators, always have one implied argument: the object through which the function was invoked

** The left hand operand (object) invokes the operator; the right hand operand (object) becomes the argument to the operator

Overloaded Operator Example

class complex {
private:
    double real;
    double imag;

public:
    complex(double r=0, double i=0);
    complex operator+(complex);
    complex operator+(double);

friend complex operator+(double, complex);
friend complex operator+(complex, complex);
};

complex c1(3,4);
c2 = c1 + c2;

c1 = 3.5;
c2 = c1 + d;

Overloaded Operator Example

complex complex::operator+(complex b) {
    double r = real + b.real;
    double i = imag + b.imag;
    return complex(r, i);
}

complex complex::operator+(double b) {
    double r = real + b;
    double i = imag;
    return complex(r, i);
}

Overloaded Operator Example

complex operator+(double a, complex b) {
    double r = a + b.real;
    double i = b.imag;
    return complex(r, i);
}

complex operator+(complex a, complex b) {
    double r = a.real + b.real;
    double i = a.imag + b.imag;
    return complex(r, i);
}

Does all 3 operations – Assumes a double to complex conversion ctor
Return Efficiency

Minimizing constructor and destructor calls

- temp is created, including a constructor call
- Copy constructor copies the contents to the outside return location (usually a variable – another constructor call)
- The destructor is called for temp
- 3 total constructor/destructor calls

```cpp
complex complex::operator+(complex b)
{
    complex temp;
    temp.real = real + b.real;
    temp.imag = imag + b.imag;
    return temp;
}
```

CS 1220 Chapter 8 Slide 7 of 16

Return Efficiency Continued

A special case

- Builds the return object directly into the location of the outside object (one constructor call)
- c3 = c1 + c2
- The result of c1 + c2 is constructed at the memory location of c3
- No destructor call
- 1 total constructor/destructor calls

```cpp
complex complex::operator+(complex b)
{
    return complex{real + b.real, imag + b.imag};
}
```

CS 1220 Chapter 8 Slide 8 of 16

Stack Example

Other operators

```cpp
class Stack {
    int& operator[](int index);
};

int& Stack::operator[](int index)
{
    return stackmem[index];
}

Stack S;
S[5] = 10;  // can be an lvalue
S[0] = S[5];  // can be an rvalue
```

CS 1220 Chapter 8 Slide 9 of 16

Overloading Increment and Decrement

Operators with special problems

- Originally, pre- and post- versions were not distinguished
- Preincrement and predecrement
  ```cpp
  ++a  \rightarrow \text{operator++(a)}  \quad \text{// nonmember (i.e., friend)}
  ++a  \rightarrow \text{operator++( )}  \quad \text{// member}
  --a  \rightarrow \text{operator--(a)}  \quad \text{// nonmember (i.e., friend)}
  --a  \rightarrow \text{operator--( )}  \quad \text{// member}
  ```
- Postincrement and postdecrement (distinguished by a dummy integer parameter)
  ```cpp
  a++  \rightarrow \text{operator++(a, int)}  \quad \text{// nonmember (i.e., friend)}
  a++  \rightarrow \text{operator++(int)}  \quad \text{// member}
  a--  \rightarrow \text{operator--(a, int)}  \quad \text{// nonmember (i.e., friend)}
  a--  \rightarrow \text{operator--(int)}  \quad \text{// member}
  ```

CS 1220 Chapter 8 Slide 10 of 16

Overloading operator<<

Making << work for user-created types

- friend function
- Returns ostream&
- Takes two references
  - ostream&
  - The user-created type (class) reference
- Example
  ```cpp
  friend ostream& operator<<(ostream& out, Fraction& frac);
  ostream& operator<<(ostream& out, Fraction& frac)
  {
    out << frac.numerator << " / " << frac.denominator << endl;
    return out;
  }
  ```

CS 1220 Chapter 8 Slide 11 of 16

Operator<<

Continued

```cpp
#include <iostream>

class Stack {
public:
    friend ostream& operator<<(ostream& out, Stack& stk);
};

ostream& operator<<(ostream& os, Stack& stk)
{
    os << "size: " << stk.stackptr << endl;
    for (int i = stk.stackptr-1; i >= 0; i--)
        os << stk.memory[i] << endl;
    return os;
}

cout << S << endl;
```

CS 1220 Chapter 8 Slide 12 of 16
Operator Overloading Summary

Some special cases

- Unusual Operators
  - operator [] // only one parameter
  - operator,
  - operator () // doesn't like default parameters
  - operator new
  - operator delete
  - operator->*
  - operator->
  - operator?: // used for what?

- Operators That Cannot Be Overloaded (compare w/p. 360)
  - operator.
  - operator.*
  - operator::

Overloading += (and Similar)

Version 1

```cpp
void operator+=(time t2)
{
    int i1 = hours * 3600 + minutes * 60 + seconds;
    int i2 = t2.hours * 3600 + t2.minutes * 60 + t2.seconds;
    int s = i1 + i2;
    hours = s / 3600;
    s = s % 3600;
    minutes = s / 60;
    seconds = s % 60;
}
```

What about

time t3 = t1 += t2;

Overloading += (and Similar)

Version 2

```cpp
time& time::operator+=(time t2)
{
    int i1 = hours * 3600 + minutes * 60 + seconds;
    int i2 = t2.hours * 3600 + t2.minutes * 60 + t2.seconds;
    int s = i1 + i2;
    hours = s / 3600;
    s = s % 3600;
    minutes = s / 60;
    seconds = s % 60;
    return *this;
}
```

Overloading =

Similar to the copy constructor

- The parameter and the return type must be references
- Consider a = b = c

```cpp
time& time::operator=(time& t)
{
    hours = t.hours;
    minutes = t.minutes;
    seconds = t.seconds;
    return *this;
}
```