Operator Overloading

• C++ feature that allows implementer-defined classes to specify class-specific function for operators

• Benefits
  – allows classes to provide natural semantics for operators in a manner similar to built-in types

• Drawbacks
  – poor use leads to confusing code
  – many classes do not have natural overloaded semantics and, thus, overloaded operators are ambiguous
    • it is not clear that this situation is worse than with poorly named functions
    • however, it was serious enough that Java left the feature out

Running Example: Complex Numbers

• Complex numbers
  – natural class for operator overloading
  – operators against this class are
    • well-defined
    • conform with user expectation
  – thus, they are easy to use

```cpp
class complex {
    public:
        double _real;
        double _imag;
    private:
        ...}
```

Syntax

• Operators are treated like functions
  – take arguments, allow implementer to define semantics

• Operators are invoked differently than functions
  – unary operators (take one argument)
    • usage: \(-a\)
    • usage: \(a^b\)
  – binary operators
    • usage: \(a \cdot b\)
    • usage: \(a = b\)
  – mapping operators to functions
    • usage: \(a + b\)
    • matches prototypes
      • complex & complex::operator+ ( complex & other )

Syntax -- First Example

• operator+ -- the first example

• The following are equivalent
  – note that the compiler will factor out the temporary variables
  – I prefer the first actually
    • (aside) don’t be afraid to use temporary variables when they simplify code

```cpp
complex & complex::operator+ ( const complex & other ) const
{
    double real = this._real + other._real;
    double imag = this._imag + other._imag;
    return complex ( real, imag );
}
```

• almost all operators
  • arithmetic
    • +, -, *, /, %, ++, --
  • Bitwise operators
    • ^, &, >>, <<
  • Logical operators
    • &&, ||, !
  • Comparison
    • >, <, <=, >=, ==, !=
  • Array, function, comma
    • [ ], [ ]
  • Assignment and composite operators
    • =, ^=, |=, &=, +=, -=, *=, /=, %=, <<=, >>=,
What cannot be done

- Some operators cannot be overloaded
- Non-pointer access to class members
- Scope
- If/then/else operator
- New operators cannot be created, even if desired

const and

- Almost all overloaded operators (except those involving equality) should be const, i.e. they should not change the implicit argument
- Similarly, they should use const function arguments

Unary and Binary

- Both the unary and binary forms of an operator can be overloaded
- Differentiated by their arguments
- Applies to: +, -, *, &

Anonymous Objects and

- First (well motivated) use of anonymous objects
- Note it is often wrong (almost always) to modify this
- Same reason we use const

Polymorphic Functions and

- Operator overloading is done through functions, and, thus, may be polymorphic
- Actually, unary and binary forms are already polymorphic
Class Arguments and Global Scope

- One argument to a operator overloaded function must be a class argument
  - i.e. it is not possible to change the implementation of operators against built-in types
- Operators may also be of global scope
  - the following are equivalent
  - note that the second is not a class member
  - It is preferable to have overloaded operators belong to classes. Why?

```cpp
complex a ( 1.0, 1.0);  
cout << a << endl;      // prints "1.0 + 1.0i"
```

Common Overloaded Operators

- Now let’s construct some common operators for the complex class
  - operator<< and operator>>
  - operator=
  - operator<
- Interestingly, all of these common cases have interesting effects that do not conform well to the operator overloading model

```cpp
ostream & operator<< ( const complex & other );  
ostream & operator<< ( const complex & other );
```

operator>> and operator<<

- So we want to print out our imaginary numbers
  - What is the prototype of the function?
  - Is it binary or unary?
  - What are the arguments?

```cpp
complex a ( 1.0, 1.0);  
ostream & operator<< ( const complex & other );
```

operator>> and operator<<

- Two possible prototypes
  - class scope
  - global scope
- What is the problem with the first prototype?
- Can we make the operator a member of class complex?

```cpp
ostream & operator<< ( const complex & other );
```

Some properties of ostream

- Main ostream operator
  - ostream & ostream::operator<< ( const char [] )
- Also provides for all basic types
  - ostream & ostream::operator<< ( const type & other );
- Because it returns an ostream reference, the arguments may be chained
  - cout « "The value of PI is approximately " « 3.14159 « endl;
- Note that the function is not constant
  - changes the state of the ostream object

```cpp
ostream & operator<< ( const char [] );
```
**operator>> and operator<<**

- Implementation of the operator
  - must choose the operator to be global scope
  - we cannot modify class stream
- How did we do?
- All arguments for operator<< apply to operator>>
  - just for class stream

```cpp
ostream & operator<< (ostream & os, const complex & other )
{
  os << other._real << " + " << other._imag << "i";
  return os;
}
```

**operator<>**

- An obvious operator with an (usually) obvious implementation
- complex numbers are not well-ordered mathematically speaking
- Different example -- countdown timer class CDT

```cpp
bool CDT::operator< ( const CDT & other ) const
{
  return ( _seconds < other.seconds );
}
```

**operator<=**

- The catch, when one defines operator<
- The compiler also creates
  - operator> ( use a < b, but switch arguments b > a )
  - operator< ( ! ( a < b ) implies a <= b )
  - operator>= ( ! ( b < a ) implies a >= b )

- What is the danger here?
  - if operator< does not provide a total ordering, then the other operators will not be well defined
  - one good example is that of line segments
  - operator< resolves the segment based on length
  - any two segments of the same length will be declared equal

**operator=**

- The C++ compiler always provides an overloaded operator=
  - even when not requested
  - provides a byte-wise copy of the object
  - deep copy of built in types
  - shallow copy of pointers
- Every class should provide its own operator=
  - avoids confusing/unwanted default behavior
  - frequently a simple function
  - often identical to the default
- We will discuss shallow and deep copying tomorrow

```cpp
complex & operator= ( const complex & other )
{
  this->_real = other._real; 
  this->_imag = other._imag; 
  return *this;
}
```

**Type Conversions**

- Constructors are the preferred method of converting the type of an object
  - e.g. the following converts a double to a complex
- let’s write it

```cpp
complex::complex ( double real )
{
  _real = real;
  _imag = 0;
}
```
Type Conversions

• Constructors are the preferred method of converting the type of an object
  - *e.g.* the following converts a double to a complex
  - how did we do?

\[
\text{complex::complex ( double real ) : _real ( real ), _imag ( 0 )}
\]

Type Conversion with Operators

• Two circumstances under which we cannot perform type conversion via constructors
  - they are?

Type Conversion with Operators

• Any constructor that accepts a single argument doubles as a type conversion routine

Type Conversion with Operators

• Some observations
  - we cannot do this by providing a new constructor in string
    - this would be preferred, but we cannot change string
  - operator string() does not have strict function semantics
    - does not specify a return type

\[
\text{complex::operator string()}\text{ const} \\
\text{ char buffer [31];}
\text{ sprintf ( buffer, "\%d + \%d i", _real, _imag );}
\text{ return string ( buffer );}
\]
operator[]

- Frequently overloaded in container classes
  - C++ standard class vector overloads to provide array-like behavior
- Simple example -- look at the contents of a stack of items
  - class stack
    
    ```cpp
    public:
        item & operator[](int which) const;
    private:
        vector <item> v;
    
    item & operator[](int which) const
    { }
    ```

Questions

- Why can one overload the operators that access class members through pointers (->, ->*), while it is illegal to overload standard class access members (.,.*)?

Why/When/What to not Overload

- confusing precedence
- expected equivalence

Precedence

- Benefit of operators is operator-like syntax, rather than functional syntax
  - convenient, intuitive and concise, ... right?
  - but, why are there RPN calculators?
- Operator evaluation relies on evaluation ordering rules
  - C++ is not the same as mathematics
    - for example, `^` is the C++ bitwise exclusive-or (not exponentiation)
    - overloading for exponentiation does not work
- a ^ 2 + 1 evaluates to `operator^(2+1)`
  - math would prefer `operator^(2) + 1`
- Functions do not have precedence problems
  - functional notation (parentheses) ensure this

Equivalence Operations

- We have an expectation that the following are the same:
  - `a = a + 1;`
  - `a += 1;`
- However, they invoke four entirely different functions
- It is the responsibility of the implementer to provide a complete and logically consistent set of operators
  - frequently a tedious (and often incomplete in my experience) task

Why no operator overloading in Java

- Book gives a lot of guidelines on operator overloading
  - merits and demerits
  - dangers of usage
- Why would Java leave this (well-established) feature out of a very similar OO language?
- Do you agree with the decision?
- What key C++ feature relies on operator overloading?
  - interestingly, Java is weak in this department
How bad does it get?

- Frequently, for error testing, it is desirable to overload `class::operator bool()`
- Used to detect errors
- `ostream::operator bool()` returns true if an error on the stream, false otherwise

```cpp
cout << x << y << z << endl;
if (cout)
{
// process error
}
```

How do we solve this?

- Overloading `class::operator void*`() solves the problem
  - `void*` cannot be shifted or manipulated by other operators
  - Stupid idiom to address a stupid problem

```cpp
cout >> x; // this now causes an error as cout cannot
if (cout)
{
// process error
}
```

Beware of Compiler Coercion

- If a compiler cannot find an exact match for a function prototype that you have provided, it will find a near match by coercion
  - Our example, converting bool to int
- This is dangerous because often it is unintended
  - Running the compiler with `-Wall` will let the implementor know when all such implicit coercions occur
  - This is a very good idea

```cpp
int x = 1;
int y = -7;
complex a(x, y); // invokes complex(double, double)

double x = 54.2;
CoundownTimer cdt(x); // invokes CoundownTimer(int)
```

```cpp
int x = 1;
int y = -7;
complex a (x, y); // invokes complex(double, double)

double x = 54.2;
CoundownTimer cdt(x); // invokes CoundownTimer(int)
```