Object – Oriented Programming
Week 8, Spring 2009

Overloaded operators

Weng Kai
http://fm.zju.edu.cn
Wednesday, 8 Apr., 2009

Overloading Operators

- Allows user-defined types to act like built in types
- Another way to make a function call.

Overloaded operators

Unary and binary operators can be overloaded:

+ - * / % ^ & | ~

= < > += -= *= /= %=

^= &= |= << >> >>= <<= ==

!= <= >= ! && || ++ --

, ->* -> () []

operator new operator delete

operator new[] operator delete[]

Operators you can't overload

. .* :: ?:

sizeof typeid

static_cast dynamic_cast const_cast

reinterpret_cast

Restrictions

- Only existing operators can be overloaded (you can't create a ** operator for exponentiation)
- Operators must be overloaded on a class or enumeration type
- Overloaded operators must
  - Preserve number of operands
  - Preserve precedence

C++ overloaded operator

- Just a function with an operator name!
  - Use the `operator` keyword as a prefix to name

    `operator` `*` (…)
- Can be a member function
  - Implicit first argument

    ```
    const String String::operator +(const String& that);
    ```
- Can be a global (free) function
  - Both arguments explicit

    ```
    const String operator+(const String& r, const String& l);
    ```

How to overload

- As member function
  - Implicit first argument
  - No type conversion performed on receiver
  - Must have access to class definition

Operators as member functions

```
class Integer {
public:
  Integer( int n = 0 ) : i(n) {}
  const Integer operator+(const Integer& n) const{
      return Integer(i + n.i);
  }
  ...
private:
   int i;
};
```

See: OperatorOverloadingSyntax.cpp

Member Functions

```
Integer x(1), y(5), z;
x + y;     ====> x.operator+(y);
```

- Implicit first argument
- Developer must have access to class definition
- Members have full access to all data in class
- No type conversion performed on receiver

```
z = x + y;   √
z = x + 3;   √
```

~~`z = 3 + y;`~~

Member Functions...

- For binary operators (+, -, *, etc) member functions require one argument.
- For unary operators (unary -, !, etc) member functions require no arguments:

```
const Integer operator-() const {
    return Integer(-i);
}
...
z = -x;  // z.operator=(x.operator-());
```

How to overload

- As a global function
  - Explicit first argument
  - Type conversions performed on both arguments
  - Can be made a friend

Operator as a global function

```
const Integer operator+(
            const Integer& rhs,
            const Integer& lhs);
Integer x, y;
x + y     ====> operator+(x, y);
```

- Explicit first argument
- Developer does not need special access to classes
- May need to be a friend
- Type conversions performed on both arguments

Global operators (friend)

```
class Integer {
  friend const Integer operator+ (
             const Integer& lhs,
             const Integer& rhs);
   ...
}
const Integer operator+(
            const Integer& lhs,
            const Integer& rhs){
   return Integer( lhs.i + rhs.i );
}
```

Global Operators

- binary operators require two arguments
- unary operators require one argument
- conversion:

```
z = x + y;
z = x + 3;
z = 3 + y;
z = 3 + 7;
```

- If you don't have access to private data members, then the global function must use the public interface (e.g. accessors)

Tips:Members vs. Free Functions

- Unary operators should be members
- = () [] -> ->* must be members
- assignment operators should be members
- All other binary operators as non-members

Argument Passing

- if it is read-only pass it in as a const reference (except built-ins)
- make member functions const that don't change the class (boolean operators, +, -, etc)
- for global functions, if the left-hand side changes pass as a reference (assignment operators)

Return Values

- Select the return type depending on the expected meaning of the operator. For example,
  - For operator+ you need to generate a new object. Return as a const object so the result cannot be modified as an lvalue.
  - Logical operators should return bool (or int for older compilers).

The prototypes of operators

- +-*/%^&|~
  - const T operatorX(const T& l, const T& r);
- ! && || < <= == >= >
  - bool operatorX(const T& l, const T& r);
- []
  - T& T::operator[](int index);

operators ++ and --

- How to distinguish postfix from prefix?
- postfix forms take an int argument -- compiler will pass in 0 as that int

```
 class Integer {
public:
    ...
    const Integer& operator++();   //prefix++
    const Integer operator++(int); //postfix++
    const Integer& operator--();   //prefix--
    const Integer operator--(int); //postfix--
    ...
};
```

Operators ++ and --

```
const Integer& Integer::operator++() {
   *this += 1;       // increment
   return *this;     // fetch
}

// int argument not used so leave unnamed so
// won't get compiler warnings
const Integer Integer::operator++( int ){
   Integer old( *this );   // fetch
   ++(*this);              // increment
   return old;             // return
}
```

Using the overloaded ++ and --

```
// decrement operators similar to increment
    Integer x(5);
    ++x;
          // calls x.operator++();
    x++;
          // calls x.operator++(0);
    --x;
          // calls x.operator--();
    x--;
          // calls x.operator--(0);
```

- User-defined prefix is more efficient than postfix.

Relational operators

- implement != in terms of ==
- implement >, >=, <= in terms of <

```
class Integer {
    public:
        ...
    bool operator==( const Integer& rhs ) const;
    bool operator!=( const Integer& rhs ) const;
    bool operator<( const Integer& rhs ) const;
    bool operator>( const Integer& rhs ) const;
    bool operator<=( const Integer& rhs ) const;
    bool operator>=( const Integer& rhs ) const;
}
```

Relational operators

```
bool Integer::operator==( const Integer& rhs ) const {
   return i == rhs.i;
}
// implement lhs != rhs in terms of !(lhs == rhs)
bool Integer::operator!=( const Integer& rhs ) const {
   return !(*this == rhs);
}

bool Integer::operator<( const Integer& rhs ) const {
   return i < rhs.i;
}
```

Relational Operators...

```
// implement lhs > rhs in terms of lhs < rhs
bool Integer::operator>( const Integer& rhs ) const {
   return rhs < *this;
}
// implement lhs <= rhs in terms of !(rhs < lhs)
bool Integer::operator<=( const Integer& rhs ) const {
   return !(rhs < *this);
}
// implement lhs >= rhs in terms of !(lhs < rhs)
bool Integer::operator>=( const Integer& rhs ) const {
   return !(*this < rhs);
}
```

Operator []

- Must be a member function
- Single argument
- Implies that the object it is being called for acts like an array, so it should return a reference

```
Vector v(100);   // create a vector of size 100
```

```
v[10] = 45;
```

(Note: if returned a pointer you would need to do:

```
*v[10] = 45;
```

See: vector.h, vector.cpp

Copying vs. Initialization

```
MyType b;
MyType a = b;
a = b;
```

Example: CopyingVsInitialization.cpp

Automatic operator= creation

- The compiler will automatically create a **type::operator=(type)** if you don’t make one.
- *memberwise assignment*

- Example: AutomaticOperatorEquals.cpp

Assignment Operator

- Must be a member function
- Will be generated for you if you don't provide one
  - Same behavior as automatic copy ctor -- memberwise assignment
- Check for assignment to self
- Be sure to assign to all data members
- Return a reference to *this

```
A = B = C;
// executed as
A = (B = C);
```

Skeleton assignment operator

```
T&   T::operator=( const T& rhs ) {
    // check for self assignment
    if ( this != &rhs)  {
        // perform assignment
    }
    return *this;
}

//This checks address vs. check value (*this != rhs)
```

Example: SimpleAssignment.cpp

Assignment Operator

- For classes with dynamically allocated memory declare an assignment operator (and a copy constructor)
- To prevent assignment, explicitly declare operator= as private

Defining a stream extractor

- Has to be a 2-argument free function
  - First argument is an `istream&`
  - Second argument is a *reference* to a value

```
istream&
operator>>(istream& is, T& obj) {
    // specific code to read obj
   return is;
}
```

- Return an `istream&` for chaining

```
cin >> a >> b >> c;
((cin >> a) >> b) >> c;
```

Creating a stream inserter

- Has to be a 2-argument free function
  - First argument is an ostream&
  - Second argument is any value

```
ostream&
operator<<(ostream& os, const T& obj) {
    // specific code to write obj
   return os;
}
```

- Return an `ostream&` for chaining

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

Creating manipulators

- You can define your own manipulators!

```
// skeleton for an output stream manipulator
ostream& manip(ostream& out) {
   ...
   return out;
}
ostream& tab ( ostream& out ) {
   return out << '\t';
}
cout << "Hello" << tab << "World!" << endl;
```

Value classes

- Appear to be primitive data types
- Passed to and returned from functions
- Have overloaded operators (often)
- Can be converted to and from other types
- Examples: Complex, Date, String

User-defined Type conversions

- A conversion operator can be used to convert an object of one class into
  - an object of another class
  - a built-in type
- Compilers perform implicit conversions using:
  - Single-argument constructors
  - implicit type conversion operators

Single argument constructors

```
class PathName {
   string name;
public:
  // or could be multi-argument with defaults
  PathName(const string&);
  ~ PathName();
};
...
string abc("abc");
PathName xyz(abc); // OK!
xyz = abc;         // OK abc => PathName
```

Example: AutomaticTypeConversion.cpp

Preventing implicit conversions

- New keyword: explicit

```
class PathName {
   string name;
public:
  explicit PathName(const string&);
  ~ PathName();
};
...
string abc("abc");
PathName xyz(abc); // OK!
xyz = abc;         // error!
```

Example: ExplicitKeyword.cpp

Conversion operations

- Operator conversion
  - Function will be called automatically
  - Return type is same as function name

```
class Rational {
public:
    ...
    operator double() const;  // Rational to double
}
Rational::operator double() const {
  return numerator_/(double)denominator_;
}
Rational r(1,3); double d = 1.3 * r; // r=>double
```

General form of conversion ops

- X::operator *T* ()
  - Operator name is any type descriptor
  - No explicit arguments
  - No return type
  - Compiler will use it as a type conversion from X $\Rightarrow$ *T*

C++ type conversions

- Built-in conversions
  - *Primitive*

    `char` ⇒ `short` ⇒ `int` ⇒ `float` ⇒ `double`

    ⇒ `int` ⇒ `long`

  - *Implicit (for any type T)*

`T` ⇒ `T&`	`T&` ⇒ `T`	`T*` ⇒ `void*`
`T[ ]` ⇒ `T*`	`T*` ⇒ `T[ ]`	`T` ⇒ `const T`

- User-defined T ⇒ C
  - *if* `C(T)` *is a valid constructor call for* `C`
  - *if* ***`operator`*** `C()` *is defined for* `T`
- BUT
  - See: TypeConversionAmbiguity.cpp

Do you want to use them?

- In General, no!
  - Cause lots of problems when functions are called unexpectedly.
  - See: CopyingVsInitialization2.cpp
- Use explicit conversion functions. For example, in class Rational instead of the conversion operator, declare a member function:

```
double toDouble() const;
```

Overloading and type conversion

- C++ checks each argument for a "best match"
- Best match means cheapest
  - Exact match is cost-free
  - Matches involving built-in conversions
  - User-defined type conversions

Overloading

- Just because you can overload an operator doesn't mean you should.
- Overload operators when it makes the code easier to read and maintain.
- Don't overload && || or , (the comma operator)

quiz

- Can we change the behavior of operators to the primitive types?

- See: plus.cpp