207

Chapter 7

The C++ Programming Language

Operator Overloading

208

Operator Overloading

Having defined the class `Complex`,
we want to be able to add two complex numbers.

```
Complex z0(1,2), z1(3.5), z2;
z2 = z0 + z1;   // Syntax error
```

The compiler knows that it does not know how to add complex numbers

Defining a standard function would make the code harder to read.

```
  x + y       // Add two doubles
add(z0,z1)    // Add two Complexes
```

If only we had + as a function we could have overloaded it.

```
   // Yes, operator + is a function
operator+()  // The name of the function
```

209

Operator Overloading

Almost all standard **C++** operators can be overloaded for classes (and structures) by defining functions of the form `operator` OP, where OP is one of the predefined **C++** operators.

Here is the first version of `operator+()`, implemented as a member function:

```
class Complex {
  public:
     Complex(double r = 0, double i = 0)
        : real(r), imag(i) {}
     Complex operator +(const Complex&) const;
  private:
     double real,imag;
};

Complex Complex::operator +(const Complex& z) const
{ return Complex(real + z.real, imag + z.imag);}
```

operator+ does not modify its arguments

210

Using Overriden Operators

Let `z1`, `z2` and `z3` be Complex:

```
Complex z1(1,2), z2(-1,4), z3;
```

then the C++ code:

```
z3 = z1 + z2;
```

actually calls two operator functions:

```
z3.operator= (z1.operator+(z2));
```

where the function `operator=()` is predefined by the compiler.

211

Pre-Defined Operators

- The only operators defined by default for all classes are:
  - **Address of:** `operator&` **(unary)**
  - **Assignment:** `operator=` **(binary)**
    - Allows assignment of one object of the class to another.
    - Memberwise (recursive) copy assignment
  - **Sequencing:** `operator,` **(binary)**
- Even these are just a result of a *"Historical Accident"* (compatibility with C).

212

Basic Rules of Operator Overloading

- The overloaded instance of an operator must contain at least one argument of a class type.
  - **Otherwise other versions of the operator may be invoked.**
- Only the predefined operators may be overloaded.
- The predefined precedence and associativity direction of any operator cannot be changed.
- The unary/binary nature of the operator cannot be changed.
- Cannot be overloaded:

  `::` `.` `.*` `?:` **`sizeof`**

213

The Rational Number Class

```
class Rational {
  public:
     Rational(int top = 0, int bottom = 1):
        t(top), b(bottom) { normalize();}
    // ...
  private:
    int t,b;
    void normalize() // A private member function.
    {   if (b < 0) {
          b = -b;
          t = -t;
        }

        int divisor = gcd(abs(t),b);
        t /= divisor;
        b /= divisor;
    }
};
```

gcd() is defined elsewhere

214

A Unary Operator Member

```
class Rational {
//...
public:
//...
  Rational operator -() const
  { return Rational(-t,b);}
//...
};

Rational r(-1,3);
Rational s  = -r;    // Activates r.operator-()
```

The *hidden* argument is const

215

Arithmetic Operator Members

```
class Rational {
   // ....
public:
   // ....
   Rational operator +(const Rational& r) const
        { return Rational(t*r.b+r.t*b, b*r.b); }
   Rational operator -(const Rational& r) const
        { return Rational(t*r.b-r.t*b, b*r.b); }
   Rational operator *(const Rational& r) const
        { return Rational(t*r.t, b*r.b); }
   Rational operator /(const Rational& r) const
        { return Rational(t*r.b, b*r.t); }
   }
   // Note that Rational() calls Rational::normalize()
};
```

216

Type Conversion via Constructors

In our example, the `Rational` constructor provides conversion from `int` to `Rational`, thus we can write:

A Ctor with a single argument acts as a conversion operator

```
Rational r, r2(1,2),r3(1,3);
...
r = r2 + r3;   // Same as r2.operator+(r3)
...
r = r2 - 73;   // Same as r2.operator-(Rational(73))
...
r = r2 * 'a'; // Same as r2.operator*(Rational(int('a')))
...
// And even:
r = r3 * 13.1 // Same as r3.operator*(Rational(int(13.1)))
```


217

Symmetric Operators

```
Rational r, r2(1,2),r3(1,3);
...
```

We are not interested in monsters like:

```
3 += r;
```

Even an **`int`** on the left is unreasonable

```
i += r; // operator+=
```

should not be symmetric

```
...
```

But it is really required to have:

```
...
r = 5 + 3 * r2;// +, *,
```

etc. should be symmetric

```
...
```

218

Symmetric Operators

Unfortunately, the above will not work, since the object which is the receiver of the message, must appear first.

```
3 * r   // Is equivalent to  3.operator*(r)
```

Will try to activate a member function of "class" **`int`**

Since **`this`** is never converted from one type to another, we need a function that has no implicit parameter.

The simplest solution is to define a function that is not a member function (of any class).

Alas, this non-member function may need access to **`private`** members of the objects.

219

Friends

- A *function* `f` or *class* `A` may be defined to be a *friend* of class `B`.
- This allows access by `f` or by all methods of `A` to *private* fields of `B`.
- `B` declares its friends.
- Useful for efficiency.

Do not compile :-)

```
class A { . . . };
class B;
int f(double) { B b;
  printf("%d\n",b.x); }

class C {
   public:   void g(int);
 . . .
};

class B {
     friend A;
     friend int f(double);
     friend void C::g(int);
  private: int x;
  public:  . . .
};
```

220

Arithmetic Operators: Friends

Operators can be defined as member or non-member functions. Let's try the non-member version:

the functions are also `inline`

```
class Rational {
public:
   // ....
   friend Rational operator*
   (const Rational& r1, const Rational& r2)
         { return Rational(r1.t*r2.t,r1.b*r2.b); }
   friend Rational operator/
   (const Rational& r1, const Rational& r2)
         { return Rational(r1.t*r2.b,r1.b*r2.t); }
   // ....
private:
   // ....
};
```

221

Using Arithmetical Operators

Now we can write:

```
...
int i;
...
r = i / (3 * r2);
...
```

With the compiler doing the necessary conversions to type Rational.

222

A Friend Unary Operator

A member is preferred

```
class Rational {
  public:
   //...
   friend Rational operator -(const Rational& v)
    {
      return Rational(-v.t,v.b);
    }
   //...
};
...
Rational r(-1,3);
Rational s = -r;
```


223

This

Within any member function definition, `this` is a pointer to the object.

`this` is the only way to explicitly call the object.

```
class Complex {
      double real,imag;
  public:
      double real_part() const
        { return this->real; }
   // . . .
};
```

224

Overloading Assignment Operators

- **Overloading operators of the type OP= should be done with care:**
  - Always use member functions
    - »*Friends do not guarantee that left operand is an lvalue.*
  - The return type should be a reference to the class.
    - »*C++ allows constructs of the form:*

      `(X += Y) *= Z;`
  - The operator should return reference to `*this`.
  - The above is true for simple assignment: `a = b = c`
- **The compiler may not enforce all these rules.**

225

A Binary Operator Member
(First version)

```
class Rational {
  public:
    // ....
    Rational& operator +=(const Rational &val)
    { t = t * val.b + val.t * b;
      b *= val.b;
      normalize();
      return *this;   // a reference to the object
    }
    // ....
};

Rational r1(1,2), r2(1,3), r3(1,4);
r1 += 5;
(r1 += r2) += r3;
```

226

Binary Operator Members
(Second version)

```
class Rational {
  public:
        // Using previously-defined operators:
   Rational& operator +=(const Rational& val)
      { return *this = *this + val; }
   Rational& operator -=(const Rational& val)
      { return *this = *this - val; }
   Rational& operator *=(const Rational& val)
      { return *this  = *this * val; }
   Rational& operator /=(const Rational& val)
      { return *this = *this / val;
   // ....
};
```

In contrast to the above, experienced C++ programmers will define `operator OP` using operator `OP=`

227

Binary Operators Combination

```
class Rational {
  public:
    // ....
    Rational& operator*=(const Rational &val)
    { t *= val.t;
      b *= val.b;
      normalize();
      return *this;   // a reference to the object
    }
};

Rational operator*(const Rational& a, const Rational& b)
{ Rational tmp(a);
  return   tmp *= b;
}
```

Non-member, non-friend function

228

Relational Friend Operators

We can go on and define many more friend operators, such as all the relational operators.

Not all of them need to be friend

```
class Rational {
 public:
   // ....
   friend bool operator ==(const Rational&, const Rational&);
   friend bool operator !=(const Rational&, const Rational&);
   friend bool operator <=(const Rational&, const Rational&);
   friend bool operator >=(const Rational&, const Rational&);
   friend bool operator < (const Rational&, const Rational&);
   friend bool operator > (const Rational&, const Rational&);
   // ....
};
```

229

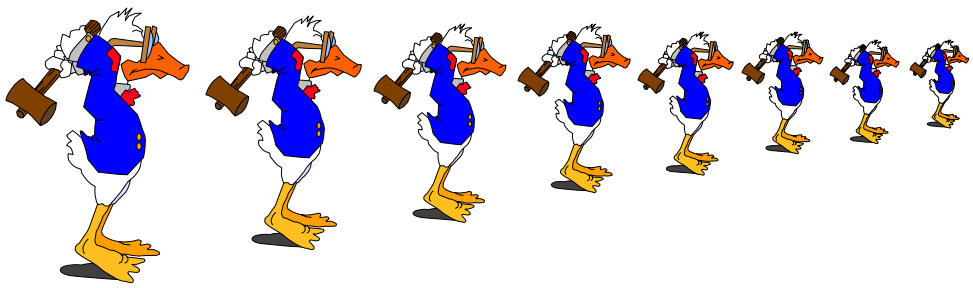
Member Operators vs. Friends

- Prefer members, if possible.
- Use non-members (sometimes friends) when you must (e.g, for symmetry).
- Use members to return a reference.

230

Making Copies of Existing Objects

The Copy Constructor
and Overloading Operator=



231

The Default Assignment Operator

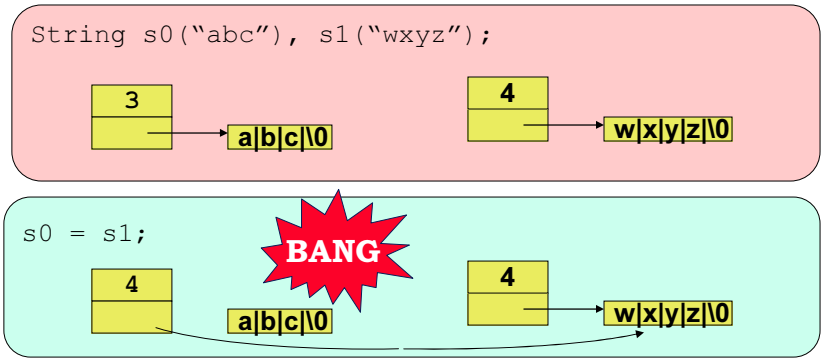
- The default assignment operator, recursively assigns each data member from the object on the right to the object on the left.
- This is exactly what we want, for objects like `Complex`, or `Rectangle`.

```
Complex c0(2), c1(-1,2.3);
Point p0(3,2), p1(5,0.1), p2(0,0);
Rectangle r0(p1,p2), r1(p1,p0);
   . . .
c0 = c1;
r0 = r1;
   . . .
```

232

The Default Assignment Operator (Cont.)

- However, consider what will happen if we use the default assignment operator for the last version we had for `String`.



233

The Default Assignment Operator (Cont.)

Here is how to correctly implement
an assignment operator for the class `String`.

```
String&
String::operator=(const String& rhs)
{ if (this == &rhs) return *this;
  len = rhs.len;
  delete[] s;
  s = new char[len + 1];
  strcpy(s, rhs.s);
  return *this;
}
```

- Return a reference
- A member function
- No self assignment

- Free old memory
- Allocate new memory
- Work with new memory
- Return the lhs object

A safer, more efficient, implementation is suggested in C++ FAQs, by M. Cline

234

The Copy Constructor

- The assignment operator copies an existing object into an existing object.
- But in many cases we have to create a new object that is an exact copy of an already existing object.
- The most visible case is when defining a variable that is initialized by an object of the same type:

```
int     i  = 3, j(5);
double  x  = i, y = x;
Complex c0 = x, c1 = c0;
String  s0("abc"), s1 = s0, s2(s0);
```


235

The Copy Constructor (Cont.)

- Less visible cases are:
  - Passing an argument, by value, into a function.
  - A function returns a value (and not a reference).
- In all these cases a constructor must construct the new object and it is named the copy Ctor:

```
class String {
  public:
     . . .
    String(const String& src): len(src.len),
        s(strcpy(new char[len+1], src.s){}
     . . .
};
```

236

The Copy Constructor (Cont.)

- How could we manage without a copy Ctor for classes like Complex or Rational ?
  - » Because the compiler creates a default copy Ctor, whenever a class is defined without one.
- How does the the default copy Ctor work?
  - » It initializes each data field of the new object by the corresponding value from the initializing object using the appropriate copy Ctor !!!

Usually, you want to share code between `operator=` and `copy Ctor`, using `private` functions

237

The Big Three

Marshal Cline, the author of the book C++ FAQs, has noted the following:

In almost all cases of class definitions, whenever you need to define one of

A copy Ctor
A Dtor
An assignment operator

YOU NEED THEM ALL

The following "equation" is the "reason": *operator=* = *Dtor* + *copyCtor*

238

Operator Overloading Test Case

Input/Output Operators

by

Ayal Itzkovitz

239

Input/Output with C++

C

```
#include <stdio.h>
int    i;
double f;
. . .
scanf("%d %lf", &i, &f);
printf("%d %f\n", i, f);
fprintf(stderr, "%f", f);
```

++

```
#include <iostream.h>
int    i;
double f;
. . .
cin  >> i >> f;
cout << i << f << endl;
cerr << f;
```

cin, cout, cerr are objects which represent the stdin, stdout, stderr respectively.

Using operator >> and operator << objects can perform input/output.

operations are performed according to the types of the objects - no formats.

240

Using I/O operators rather than function calls

- **Easier to program.**
  - No need to specify type (%d, %f, etc)
- **Object read/write themselves.** Information hiding !
  - We ask the object to write itself.
  - Object asks to get the appropriate data for its internal data structures.
  - Try printf("???", complex);
- **Robust code.**
  - Avoid bugs like scanf("%f", &i); (i is int)
- **Future extensions.**
  - Future classes should just inplement operators >> and << for themselves.

241

istream - member functions (partial list)

Construction/Destruction — Public Members

•istream Constructs an istream object.

•~istream Destroys an istream object.

Input Functions — Public Members

•get Extracts characters from the stream.

•peek Returns a character without extracting it from the stream.

•getline Extracts characters from the stream (extracts and discards delimiters).

Other Functions — Public Members

•putback Puts characters back to the stream.

Operators — Public Members

•operator >> Extraction operator for various types.

242

Using istream member functions

Reading a line from cin

```
const int max_line = 70;
char line[max_line];
cin.getline(line, max_line);
```

Reading a single char from cin

```
int c;
c = cin.get();
```

Reading a single char from cin without removing it from queue

```
int c;
c = cin.peek();
if (c == EOF) return;
```

Putting a read character back

```
int c = cin.get();
if (c == 'l') {
  load_db();
  cin.putback(c);
}
```


243

ostream - member functions (partial list)

Construction/Destruction — Public Members

•ostream Constructs an ostream object.

•~ostream Destroys an ostream object.

Flag and Format Access Functions — Public Members

•precision Sets or reads the stream's floating-point format display precision.

•width Sets or reads the stream's output field width.

Operators — Public Members

•operator << Insertion operator for various types.

Manipulators

•endl Inserts a newline sequence and flushes the buffer.

•ends Inserts a null character to terminate a string.

244

Using ostream member functions

```
cout.width(8);
cout.precision(2);
cout.operator<<(12.3456);   // or  cout << 12.3456;
    >  will print:    12.35
cout << '[' << 12.3456 << ']' << endl;
    > will print: [   12.35]
    >
```


245

Adding I/O to the Rational Number Class

```
class Rational {
private:
   int t,b;
   void normalize(); // A private member function.
public:
     // constructor
     Rational(int top = 0, int bottom = 1):
      t(top), b(bottom) { normalize();}

     // i/o operators
     friend istream&
               operator>>(istream& in,        Rational& r);
     friend ostream&
               operator<<(ostream& out, const Rational& r);

     // other operators
        . . .
};
```

246

Adding I/O to the Rational Number Class (cont.)

```
istream&
operator>>(istream& in,        Rational& r)
{  in >> r.t >> r.b;
   return in;
};

ostream&
operator<<(ostream& out, const Rational& r)
{  out << r.t << '/' << r.b;
   return out;
};
```

```
Rational r1(1,6), r2(1,3);

cout << r1 << '+' << r2 << '=' << r1+r2 << endl;
```

> will print: 1/6 + 1/3 = 1/2

247

A different way of overloading `operator<<`

```
class Rational {
   . . .
  ostream& print(ostream& os) const
  { return os << t << '/' << b; }
   . . .
};
```

The advantages of this will be clear in the future

```
// Non-friend, if Rational::print() is public
inline ostream&
operator<<(ostream& out, const Rational& r)
{ return r.print(out); }
```

ymk

248

Special Operators