Object Oriented Programming in C++

Chapter5

Operator Overloading

Operator Overloading

- What?
  - an operator that has multiple meanings
  - varies depending on use
- Why? Ease of use is a principle of OO
- How? by defining as an operator function
  - functions that can extend meaning of built-in operators (cannot define your own new operators)
  - keyword operator is in definition, followed by the operator to be overloaded
- Used
  - method syntax or operator syntax
    - s1.operator>(s2) vs. s1 > s2
- But cannot overload
  - . .* :: ?: sizeof

Restrictions on Overloading

The operators

. .* :: ?: sizeof

may not be overloaded. All other operators may be overloaded, i.e.

+	-	*	/	%	^	&	|
~	!	=	<	>	+=	-=	*=
new		delete		etc.			

- The order of precedence cannot be changed for overloaded operators.
- Default arguments may not be used with overloaded operators
- New operators cannot be created
- Overloading must be explicit, i.e. overloading + does not imply += is overloaded

Class Rational

```
class Rational {
public:
   Rational(int = 0, int = 1); // default constructor
   Rational operator+(const Rational&) const;
   Rational operator-(const Rational&) const;
   Rational operator*(const Rational&) const;
   Rational operator/(const Rational&) const;
  void printRational(void) const;
private:
   int numerator;
   int denominator;
   void reduction(void);
};
```

Implementing Class Rational

```
Rational Rational::operator+(const Rational &a) const
{
   Rational sum;
   sum.numerator = numerator * a.denominator + denominator * a.numerator;
   sum.denominator = denominator * a.denominator;
   sum.reduction();
   return sum;
}


Rational Rational::operator-(const Rational &s) const
{
   Rational sub;

   sub.numerator = numerator * s.denominator - denominator * s.numerator;
   sub.denominator = denominator * s.denominator;
   sub.reduction();
   return sub;
}
```

The Driver

```
void main()
{
   Rational c(7,3), d(3,9), x;

   c.printRational();
   cout << " + " ;
   d.printRational();
   cout << " = ";
   x = c + d;
   x.printRational();

   cout << "\n";
   c.printRational();
   cout << " - " ;
   d.printRational();
   cout << " = ";
   x = c - d;
   x.printRational();
}
```

Use of operator is c + d and c - d

c is an implicit argument to the operator

The Output

7/3 + 1/3 = 8/3

7/3 - 1/3 = 2

Operators as Friend Functions

- An operator must be a friend function if the left-most operand must be of a different class or a C++ intrinsic type
- an operator implemented as a friend function must explicitly have argument(s) of the object(s)

Rational Example Revisited

```
class Rational {
public:
   Rational(int = 0, int = 1); // default constructor
   friend Rational operator+(const Rational &, const Rational&);
   friend Rational operator-(const Rational &, const Rational&);
   void printRational(void) const;
private:
   int numerator;
   int denominator;
   void reduction(void);
};
```

Operators are now friend functions, with two arguments, for both operands

Rational Example Revisited - 2

```
Rational operator+(const Rational &f, const Rational &a)
{
   Rational sum;

   sum.numerator = f.numerator * a.denominator + f.denominator * a.numerator;
   sum.denominator = f.denominator * a.denominator;
   sum.reduction();
   return sum;
}
```

There is no change in the driver program

Arithmetical Operators: Friends

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

Using Arithmetical Operators

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

- the compiler doing the necessary conversions to type Rational.

A Binary Member Operator

```
class Rational {
public:
        // ....
        Rational& operator += (const Rational &val) {
           numerator = numerator * val.denumerator + val. numerator * denumerator;
           b *= val.b;
           reduction();
           return *this;
       }
        // ....
};

Rational r1(1,2), r2(1,3), r3(1,4);

(r1 += r2) += r3;
```

Some More Binary Operator Members

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

    }
    Rational& operator /=(const Rational &val) {
                    return *this = *this / val;
    }

    // ....
};
```

Overloading Assignment Operators

- Overloading operators of the type OP= should be done with care:
  - Always use member functions
    - Global functions do not guarantee that first operand is an lvalue (e.g. non-reference return values)
  - The return type should be reference to the class.
    - C++ allows constructs of the form:
      - `(X += Y) += Z;`
  - The operator should return reference to `*this`.
  - The compiler may not enforce all these rules.

Addition and Subtraction as Global without using friend

```
Rational operator +(const Rational &r1, const Rational &r2)
    {
        Rational res(r1);
        return res += r2;
    }
Rational operator -(const Rational &r1, const Rational &r2)
    {
        return r1 + (-r2); // Far from efficient...
    }
```

Overloading the Stream Insertion and Extraction Operators

It is possible to overload the stream insertion (<<) and extraction operators (>>) to work with classes.

This has advantages, in that

- it makes programs more readable
- it makes programs more extensible
- it makes input and output more consistant

Overloading the Stream Operators : Rational Example

```
class Rational {
   friend istream& operator>> (istream &, Rational &);
   friend ostream& operator<< (ostream &, const Rational &);
   friend Rational operator+(const Rational &, const Rational&);
   friend Rational operator-(const Rational &, const Rational&);
public:
   Rational(int = 0, int = 1); // default constructor
   void printRational(void) const;
private:
   int numerator;
   int denominator;
   void reduction(void);
};
```

The Stream Insertion Operator <<

```
ostream &operator<< (ostream &output, const Rational &r)
{
   if (r.numerator == 0)              // print fraction as zero
      output << r.numerator;
   else if (r.denominator == 1)   // print fraction as integer
      output << r.numerator;
   else
      output << r.numerator << "/" << r.denominator;
   return output;
}
```

The Stream Extraction Operator >>

```
istream &operator>>(istream &input, Rational &r)
{
    input >> r.numerator;
    input.ignore(1); // skip the /
    input >> r.denominator;
    r.reduction();
   return input;
}
```

The Driver Program and Output

```
main()
{
   Rational c, d, x;

   cout << "Please input a fraction in the form a/b :";
   cin >> c;
   cout << "Please input another : ";
   cin >> d;
   cout << c <<  " + " << d << " = ";
   x = c + d;
   cout << x << "\n";
   c.printRational();
   cout << " - " ;
   d.printRational();
   cout << " = ";
   x = c - d;
  cout <<  x << "\n";
}
```

```
Please input a fraction in the form a/b :2/3
Please input another : 6/7
2/3 + 6/7 = 32/21
2/3 - 6/7 = -4/21
```

Overloading a Unary Operator as a Member Function

In the Class Declaration

```
public:
   Rational(int = 0, int = 1); // default constructor
   Rational operator-();
```

In the Implementation

```
Rational Rational::operator-()
{
        Rational n(-numerator, denominator);
        return n;
}
```

Type Casting

It is possible to overload the type cast operation, e.g. for the Rational Class, casting it to a float.

```
public:
   Rational(int = 0, int = 1); // default constructor
   Rational operator-();
   operator double() const;

Rational::operator double() const
{
        return float(numerator)/float(denominator);
}
```

The String Class

- Comparison: == != < > <= >=
  - These will be non-members, since they are symmetric
- Assignment: =
  - Must be a member function
    - Compiler imposed restriction
- Concatenation: +
  - Friend
- Assignment with Concatenation: +=
  - Member

The Class Definition

```
// String.h:  Header file for a simple string class
#include <stdio.h>
#include <math.h>
#include <string.h>  //Get the C string function declarations
class String {
        public:
        String(char *s = ""): str(strcpy(new char[strlen(s) + 1], s)) {}
        String(const String& s):str(strcpy(new char[strlen(s.str) + 1], s.str)) {}
        ~String(void) { delete[] str; }
    // ...
        String& operator =(const String&);
        friend String operator +(const String&, const String&);
    private:
        char* str;              //Points to null-terminated text
};
```

String Assignment Operator

```
String& String::operator =(const String& right)
   {
        if (this == &right)
              return *this;
        delete[] str; // free up the previous string
        str = new char[strlen(right.str) + 1];
        strcpy(str, right.str);
        return *this;
   }
```

Reminder: Assignment vs. Initialization

- Initialization: **by copy constructor**

```
Rational z1;
Rational z2 = z1;
```

- Assignment: **by assignment operator**

```
Rational z1;
Rational z2;
z1 = z2;
```

- Assignment and initialization are similar.
  - Usually overloading the assignment operator and the copy constructor are done together:
    - If memberwise copy is wrong in one case, it's probably wrong in the other case also.
  - Difference: **assignment needs to worry about the data being overwritten**

String Concatenation

```
String operator +(const String& left, const String& right)
{
    String result;
    delete[]result.str; // Free the empty string in the result.
    result.str = new char[strlen(left.str) + strlen(right.str) + 1];
    strcpy(result.str, left.str);
    strcat(result.str, right.str);
    return result;
}
```

Concatenation as a Special Case of Assignment

```
String& String::operator +=(const String& right)
{
    // ... real logical goes here...
}

inline String operator+(const String& left, const String& right)
{
    String result = left;
    result += right;
    return result;
}
```

Induced Semantics?

- C++ makes no assumptions about the semantics of operators, e.g.:

  `operator==` need not be the inverse of `operator!=`

  `operator+` need not be commutative or associative

  `a += b` need not be the same as `a = a + b`

- If you want these semantics you must program them explicitly

Relational Operators

```
inline int operator <= (const String& left, const String& right)
{
        return strcmp(left.str, right.str) <= 0;
}

class String {
        // ...
    friend int operator <=(const String& left, const String& right);
};
```

Relational Operators

```
inline int operator >=(const String& left, const String& right)
{
    return  right <= left;
}

inline int operator ==(const String& left, const String& right)
{
    return  left <= right && left >= right;
}

inline int operator !=(const String& left,const String& right)
{
    return  ! (left == right);
}
```

Relational Operators

```
inline int operator < (const String& left, const String& right)
{
        return left <= right && left != right;
}

inline int operator > (const String& left, const String& right)
{
    return right < left;
}
```

Conversion of String to Other Types

```
class String {
        // ...
public:
    operator const char *(void)
    {
            return str;
    }
    operator double (void);
        // ...
};

inline String::operator double()
{
    return atof(str);
}
```

Streams Output Operator

```
class String {
        ...
    friend ostream& operator <<(ostream &, const String &);
        ...
};
inline ostream& operator << (ostream& stream, const String& string)
{
        return stream << string.str;
}
```

String Input Operator

```
inline istream& operator>>(istream& stream, const String& string)
{
    char buff[256];
    stream >> buff; //Unsafe, might overflow (later...)
    string = buff;  //This will call
   // String.operator=(String(buff))
    return stream;
}
```

Overloading Special Operators

- Array Reference
- Function Call
- Increment and Decrement

Overloading Array Subscript Operator

```
class String {
        StringReference *p;
public:
        ...
        char& operator[](int i)
        {
                if (i < 0 || i >= strlen(p->str))
                        error("Index to String::operator[] is out of range");
                return p->str[i];
        }
};
```

Overloading the Function Call Operator

```
class String {
    ...
public:
    ...
    String operator() (int pos,int len);
};
```

Body of Function Call Operator

```
String String::operator()(int pos, int len)
{
    int n = strlen(p->str);

    pos = max(pos, 0);
    pos = min(pos, n);
    len = max(len, 0);
    len = min(len, n - pos);

    char *s = new char[len + 1];
    strncpy(s, p->str + pos, len);
    s[len] = '\0';

    String st(s);
    delete[] s;
    return st;
}
```

Using a Function Call Operator

```
String s("This is the time for all good men");

String s1 = s(25,4);      // s1 = "good"
// Note how s is used as a function name
```

- Clearly, we can define many different function call operators with different number or type of arguments
- Remember that such usages may be confusing

Overloading the Increment and Decrement Operators

The problem with overloading the increment (++) and decrement operators (--) is that there are two forms (pre and post), e.g. i++ and ++i.

Overloading the pre forms

This is done in exactly the same way(s) as we did for unary negation.

Overloading the post forms

A dummy argument of integer type is added to the argument list, e.g.

```
Retional operator++(int) // member function form
```

Overloading ++

```
class Counter {
private:
    unsigned int count;
public:
    Counter()        { count = 0; }            // constructor
    int get_count() const  { return count; }
    Counter operator ++ ()   { // increment (prefix)
            ++count;
            return *this;
    }
    Counter operator ++ (int){   //increment(postfix)
            Counter tmp=*this;
            ++count; return tmp;
     }
  };
```

The driver

```
void main()
{
   Counter c1, c2,c3;              // define and initialize

   ++c1;                           // increment c1
   ++c2;                           // increment c2
   c3=c2++;                        // increment c2

   cout << "\nc1=" << c1.get_count();  // display
   cout << "\nc2=" << c2.get_count();
   cout << "\nc3=" << c3.get_count();
}
```

Operator Overloading Advice

- Mimic conventional use
- Use const ref for large objects. Return large objects by reference.
- Prefer the default copy operator. If not appropriate, redefine or hide it.
- Prefer member functions if access to private data is needed. Prefer non member functions otherwise.