CS2141 – Software Development using C/C++

Operator Overloading

A Rational Class

- Consider this class for storing rational numbers:

```
class rational {
   public:
   rational( int t = 0, int b = 1 )
      : top( t ), bottom( b ) { }
   rational( const rational & r )
      : top( r.top ), bottom( r.bottom ) { }
   int numerator( ) const { return top; }
   int denominator( ) const { return bottom; }

   private:
   int top;
   int bottom;
};
```

An add function

- To implement addition with two rationals, the following could be added to the class definition:

```
const rational add( const rational & r ) const {
   int t = top * r.bottom + bottom * r.top;
   int b = bottom * r.bottom;
   return rational( t, b );
}
```

- Now addition works:

```
rational a( 5, 6 );
rational b( 2, 3 );
rational c = a.add( b );
```

A Better add Function

- The syntax of the add function could be better. It would be nicer (and make sense) to write:

```
rational c = a + b;
```

- *Operator overloading* makes this possible:

```
const rational operator+( const rational & r ) const
{
   int t = top * r.bottom + bottom * r.top;
   int b = bottom * r.bottom;
   return rational( t, b );
}
```

Operator Overloading

- Operator overloading allows existing C++ operators to work with user-defined data types.
- There are limits to this, however:
  - At least one operand must be a user-defined type. It is impossible to change the meaning of 2 + 2.
  - Cannot create new operators.
  - Cannot change precedence and associativity.
  - Don't change the meaning of an operator - **`operator+`** should always do something similar to addition.

Overloaded Operators

```
+     -     *     /     %     ^
&     |     ~     !     &&    ||
++    --    <<    >>    ,     <
<=    ==    !=    >     >=    =
+=    -=    *=    /=    %=    &=
|=    ^=    <<=   >>=   [ ]   ( )
->    ->*   new   delete
```

Functions and Methods

- Operators can generally be overloaded as member functions or global functions.
  - Unary operators can be methods with no arguments or global functions with one argument.
  - Binary operators can be methods with one argument or global functions with two arguments.
- Operators **`[]`**, **`()`**, **`->`**, and **`=`** must be methods.
- If used as I/O operators (as they usually are), **`>>`** and **`<<`** must be global functions.

Binary Arithmetic Operators

- The result should be a new value.
- The return value should be constant so it cannot be the target of an assignment:

```
(a + b) = b;  // This should be impossible
```

- Parameters are values or constant references.
  - The operands should not be modified.
  - Methods should be declared constant:

```
const rational operator/( const rational & r ) const;
```

Binary Arithmetic Ops. cont.

- Subtraction as a method:

```
const rational operator-( const rational & r ) const
{
    int t = top * r.bottom - bottom * r.top;
    int b = bottom * r.bottom;
    return rational( t, b );
}
```

- Multiplication as a global function:

```
const rational operator*( const rational & l, const rational & r )
{
    return rational( l.numerator( ) * r.numerator( ),
    l.denominator( ) * r.denominator( ) );
}
```

Comparison Operators

- Work like the binary arithmetic operators, except these return a boolean.
- Equals and less-than as methods:

```
bool operator==( const rational & r ) const
{
   return top * r.bottom == bottom * r.top;
}

bool operator<( const rational & r ) const
{
   return top * r.bottom < bottom * r.top;
}
```

Increment and Decrement

- Can be prefix form (++i) or postfix form (i++).
  - Prefix form increments and returns the new value:

```
int a = 5;
int b = a++;     // a = 6, b = 6
```

  - Postfix form increments but returns the original value:

```
int c = a++;     // a = 7, c = 6
```

- Prefix increment for the rational as a method:

```
const rational operator++( ) {
  top = top + bottom;
  return *this;
}
```

Increment and Decrement cont.

- To distinguish postfix from prefix, the postfix version uses a dummy integer argument:

```
const rational operator++( int ) {
   rational temp = *this;
   top += bottom;
   return temp;
}

const rational operator--( int ) {
   rational temp = *this;
   top -= bottom;
   return temp;
}
```

Shift Operators

- Usually overloaded for input and output.
- When used for input and output, must be a global function, not a method.
- Output operator for the rational class:

```
ostream & operator<<( ostream & out,
                      const rational & r )
{
   out << r.numerator( ) << "/" <<
   r.denominator( );
   return out;
}
```

Assignment Operator

- The right operand is copied to the left operand.
- Should return a constant reference or a constant value to prevent a second assignment.
- Assignment operator for **`rational`** as a method:

```
const rational & operator=( const rational & r )
{
   top = r.top;
   bottom = r.bottom;
   return *this;
}
```

Assignment Operator cont.

- The assignment operator will be provided by the compiler if the programmer doesn't write it
  - The compiler version just copies the data members
  - If the class has pointers to other values that should be copied, the programmer should write the assignment
- Common mistakes:
  - Not returning a value
  - Not handling self-assignment
  - Simply copying pointers rather than making copies of the heap-resident values the object has pointers to

Address-of operator

- Can be overloaded to point at part of an object:

```
class rational {
   public:
   ...

   int * operator&( ) {  // Returns pointer to
      return &top;       // top for some
   }                     // mysterious reason.

   private:
   int top;
   int bottom;
};
```

Conversion Operators

- Allows conversion from a user-defined data type to another data type
- The return type is determined from the function or method name

```
// Global function
operator double( const Rational & r )
{
   return r.numerator( ) /
   (double)r.denominator( );
}
```

Subscript Operator

- Often defined for container classes:

```
class Storage {
   public:
   Storage( int s ) { space = new int[s]; }
   ~Storage( ) { delete [] space; }

   int & operator[]( int i ) { return
   space[i]; }

   private:
   int * space;
};
```

Parenthesis Operator

- The parenthesis is the only operator that can have any number of arguments.
- Allows an object to be used like a function:

```
class LessThan {
   public:
   LessThan( int v ) : val( v ) { }
   bool operator()( int x ) { return x < val; }
   private:
   int val;
}

LessThan tester( 6 );
if( tester( 3 ) ) ...
```