

Implementing an ADT

- Have defined public interface
- Now it's time to implement operations
- Use rational number ADT as example



Rational Number Constructor

```
Rational::Rational (int n, int d)
{
  num = n;
  den = d;
}
```

Note that default values do not appear in the function definition, only in its declaration (within the class definition).

Rational Number Addition

- **Addition algorithm**
 - Get a common denominator
 - Add numerators
 - Reduce fraction
- **Implementation**
 - "Add" member function
- **Subtraction almost the same**

Addition Examples

$$\frac{1}{2} + \frac{2}{3} = \frac{3}{6} + \frac{4}{6} = \frac{7}{6}$$

$$\frac{1}{3} + \frac{1}{6} = \frac{2}{6} + \frac{1}{6} = \frac{3}{6} = \frac{1}{2}$$

$$\frac{n_1}{d_1} + \frac{n_2}{d_2} = \frac{d_2 n_1}{d_1 d_2} + \frac{d_1 n_2}{d_1 d_2} = \frac{d_2 n_1 + d_1 n_2}{d_1 d_2}$$

Rational Number Addition

```
class Rational
{
public:
    Rational Add
    (const Rational& right) const;
private:
    void Reduce();
};
```

Annotations: "Addition member function" points to the Add function; "Reduce fraction; make sure denominator is positive" points to the Reduce function.

Implementing Addition

```
Rational Rational::Add
(const Rational& right) const
{
    Rational res;
    res.num = right.den*num
    + right.num*den;
    res.den = den * right.den;
    res.Reduce();
    return res;
}
```

Annotations: "Temporary object for return value" points to the res variable; "Should we use accessors/mutators?" is in a box pointing to the Reduce() call.

Implementing Comparisons

```

bool Rational::LessThan
    (const Rational& right) const
{
    Rational temp;
    temp = Subtract(right);
    return (temp.num < 0);
}

```

Temporary object for testing result of subtraction

"Equal" is similar. Is there a better way?

Implementing Insertion

```

void Rational::Insert
    (ostream& os) const
{
    os << num;
    os << '/';
    os << den;
}

```

Output in format "3/7".

Implementing Extraction

```

void Rational::Extract
    (istream& is)
{
    char slash;
    is >> num;
    is >> slash;
    is >> den;
}

```

"Skip over" input slash.

How about error checking?

Accessor/Mutator Implementation

```

int Rational::GetNumerator() const
{
    return num;
}
void Rational::SetNumerator(int n)
{
    num = n;
    Reduce(); //??
}

```

Same method for denominator?
What if value is zero?

Back to Addition

```

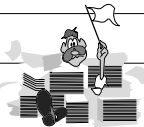
Rational a (1,3); // 1/3
Rational b (1,6); // 1/6
Rational c;      // 0/1

```

c = a.Add (b); ← OK, but awkward

c = a + b; ← Wouldn't this be nicer?

Operator Overloading



- **Can define operators**
 - For programmer-defined types
- **Implemented as functions**
 - Operator operand(s)
 - As function argument(s)
 - Object itself may be "left" operand
 - Special function names
 - Member or "outside" functions

Addition Operator Declaration

```

Rational operator+
(const Rational& left,
const Rational& right);

```

Special function name

Arguments are operands

In this case, NOT a member function!

Horizontal lines for notes

Addition Operator Definition

```

Rational operator+
(const Rational& left,
const Rational& right)
{
return left.Add(right);
}

```

Not much to do, since we already have "Add" function.

Horizontal lines for notes

What About This?

```

Rational r;
Rational s(5,7);
int t(3);

```

Define Rational and int data objects

```
r = s + t;
```

Will this work??

Compiler tries to convert s and t to the same type to apply the "+" operator. Does it know how to do this?

Horizontal lines for notes

Automatic Type Conversion

- **Compiler tries to convert**
 - E.g., int converted to double
- **What about class types?**
 - Will use single-arg constructor!
 - If explicit keyword not present
 - If argument is of "source" type
 - Creates object of class type

So This Works!

```
Rational r;
Rational s(5,7);
int t(3);
```

```
r = s + t;
```

```
r = s + (Rational)t;
```

```
r = operator+ (s, (Rational)t);
```

"(Rational) t" better as "static_cast<Rational>(t)"

This statement ...
 is equivalent
 to this ...
 or to this.

Insertion Operator Declaration

```
Rational x(1/3);
```

```
cout << x << endl;
```

Result of insertion must be "cout".

```
ostream& Rational operator<<
```

```
(ostream& os,
 const Rational& right);
```

Why return stream reference?

Insertion Operator Definition

```

ostream& operator<<
  (ostream& os,
   const Rational& right)
{
  right.Insert(os);
  return os;
}

```

Again, most of the work is done by the "Insert" function. Extraction is similar.

Equality Operator Definitions

```

bool operator==
  (const Rational& left
   const Rational& right)
{
  return left.Equal(right);
}
bool operator!= (...)
{
  return !(left==right);
}

```

Can use our "=="

Relational Operator Definitions

```

bool operator<
  (const Rational& left
   const Rational& right)
{
  return left.LessThan(right);
}
bool operator>= (...)
{
  return !(left<right);
}

```

What about ">", "<="?
