Abstract Data Types

Development and Implementation
OO Programming and Design

- **Goal**
  - Well-defined abstractions that allow objects to be created and used in an intuitive manner
    - User should not have to bother with unnecessary details
  - Example: programming a microwave

- **Common practice**
  - Use information hiding principle and encapsulation to support integrity of data

- **Result**
  - Abstract Data Type or ADT
Abstract Data Type

- Consider
  
  ```
  Rational a(1,2);  // a = 1/2
  Rational b(2,3);  // b = 2/3
  cout << a << " + " << b << " = " << a + b;
  Rational s;      // s = 0/1
  Rational        // t = 0/1
  cin >> s >> t;
  cout << s << " * " << t << " = " << s * t;
  ```

- Observation
  
  Natural look that is analogous to fundamental-type arithmetic objects
Rational Number Review

- Rational number
  - Ratio of two integers: $a/b$
    - Numerator over the denominator

- Standard operations
  - Addition
    $$a/b + c/d = \frac{ad + bc}{bd}$$
  - Subtraction
    $$a/b - c/d = \frac{ad - bc}{bd}$$
  - Multiplication
    $$a/b \times c/d = \frac{ac}{bd}$$
  - Division
    $$a/b \div c/d = \frac{ad}{bc}$$
Rational Number Representation

- Requirements
  - Represent a numerator and denominator
    - Implies in part a class representation with two `int` data members
      - NumeratorValue and DenominatorValue
      - Data members private to support information hiding
  - Public arithmetic behaviors (member functions)
    - Rational addition, subtraction, multiplication and division
  - Public relational behaviors
    - Equality and less than comparisons
      - Practice rule of class minimality
Rational Number Representation

- Other requirements
  - Public object behaviors
    - Construction
      - Default construction -- 0/1 -- design decision
      - Specific numerator and denominator construction
      - Copy construction (provided automatically)
    - Assignment (provided automatically)
    - Value insertion and extraction
  - Non-public object behaviors
    - Inspection and mutation of data members
      - Clients deal with a Rational object!
Rational Number Representation

- Other requirements
  - Auxiliary operations (necessarily public)
    - Arithmetic, relational, insertion, and extraction operations
  - Provides the natural form we expect
    - Class definition provides a functional form that auxiliary operators use
  - Provides commutativity consistency
    - For C++ reasons $1 + r$ and $r + 1$ would not be treated the same if addition was a member operation
Class Rational
Public interface: Add(), Subtract(),
Multiply(), Divide(), Equal(),
LessThan(), Insert(), Extract()
Data members: NumeratorValue,
DenominatorValue
Other members: GetNumerator(), GetDenominator(),
SetNumerator(), SetDenominator(),

Instantiation
Rational a(1,2);
Rational b(2,3);

Object a
Attributes:
NumeratorValue(1)
DenominatorValue(2)

Object b
Attributes:
NumeratorValue(2)
DenominatorValue(3)
Consider

#include "rational.h"

Rational r;
Rational s;
cout << "Enter two rationals (a/b): ";
cin >> r >> s;
Rational t(r);
Rational Sum = r + s;
Rational Product = r * s;
cout << r << " + " << s << " = " << Sum;
cout << r << " * " << s << " = " << Product;
Implementation Components

- Header file
  - Define class and prototype library functions
    - rational.h
- Rational class implementation
  - Define member functions
    - rational.cpp
- Auxiliary function implementations
  - Define assisting functions that provide expected but non-member capabilities
    - rational.cpp
Rational ADT Header File

- File layout
  - Class definition and library prototypes nested within preprocessor statements
    - Ensures one inclusion per translation unit
  - Class definition proceeds library prototypes

```cpp
#ifndef RATIONAL_H
#define RATIONAL_H

class Rational {
    // ...

};

// library prototypes ...
#endif
```
class Rational {   // from rational.h
public:
    // for everybody including clients
protected:
    // for Rational member functions and for
    // member functions from classes derived
    // from rational
private:
    // for Rational member functions
} ;
// auxiliary prototyping
Rational Public Section

public:

    // default constructor
    Rational();

    // specific constructor
    Rational(int numer, int denom = 1);

    // arithmetic facilitators
    Rational Add(const Rational &r) const;
    Rational Multiply(const Rational &r) const;

    // stream facilitators
    void Insert(ostream &sout) const;
    void Extract(istream &sin);
Rational Protected Section

protected:

    // inspectors
    int GetNumerator() const;
    int GetDenominator() const;

    // mutators
    void SetNumerator(int numer);
    void SetDenominator(int denom);
private:
    // data members
    int NumeratorValue;
    int DenominatorValue;
Auxiliary Operator Prototyping

// after the class definition in rational.h

Rational operator+(const Rational &r, const Rational &s);
Rational operator*(const Rational &r, const Rational &s);
ostream& operator<<(ostream &sout, const Rational &s);
istream& operator>>(istream &sin, Rational &r);
Auxiliary Operator Importance

Rational r;
Rational s;
r.Extract(cin);
s.Extract(cin);
Rational t = r.Add(s);
t.Insert(cout);

Rational r;
Rational s;
cin >> r;
cin >> s;
Rational t = r + s;
cout << t;

● Natural look

● Should << be a member?
  ■ Consider
    r << cout;
const Rational OneHalf(1,2);
cout << OneHalf; // legal
cin >> OneHalf; // illegal
Rational ADT Implementation

```
#include <iostream>       // Start of rational.cpp
#include <string>
using namespace std;
#include "rational.h"

// default constructor
Rational::Rational() {
    SetNumerator(0);
    SetDenominator(1);
}

● Example
    Rational r;          // r = 0/1
```

Is this necessary?
Which objects are being referenced?
Remember

- Every class object
  - Has its own data members
  - Has its own member functions
    - When a member function accesses a data member
      - By default the function accesses the data member of the object to which it belongs!
        - No special notation needed
- Auxiliary functions
  - Are not class members
  - To access a public member of an object, an auxiliary function must use the dot operator on the desired object: `object.member`
Specific Constructor

// (numer, denom) constructor
Rational::Rational(int numer, int denom) {
    SetNumerator(numer);
    SetDenominator(denom);
}

- Example

    Rational u(2);  // u = 2/1 (why?)
    Rational t(2,3); // t = 2/3
    // we’ll be using t in
    // future examples
Inspectors

int Rational::GetNumerator() const {
    return NumeratorValue;
}

int Rational::GetDenominator() const {
    return DenominatorValue;
}

● Where are the following legal?
    int a = GetNumerator();
    int b = t.GetNumerator();

Which object is being referenced?

Why the const?
Numerator Mutator

```
void Rational::SetNumerator(int numer) {
    NumeratorValue = numer;
}
```

- Where are the following legal?
  ```
  SetNumerator(1);
  t.SetNumerator(2);
  ```

 Why no const?
void Rational::SetDenominator(int denom) {
    if (denom != 0) {
        DenominatorValue = denom;
    } else {
        cerr << "Illegal denominator: " << denom
             << " using 1" << endl;
        DenominatorValue = 1;
    }
}

● Example
    SetDenominator(5);
Addition Facilitator

Rational Rational::Add(const Rational &r) const {
    int a = GetNumerator();
    int b = GetDenominator();
    int c = r.GetNumerator();
    int d = r.GetDenominator();
    return Rational(a*d + b*c, b*d);
}

● Example
    cout << t.Add(u);
**Multiplication Facilitator**

```cpp
class Rational {
public:
    static Rational Multiply(const Rational &r) const {
        int a = GetNumerator();
        int b = GetDenominator();
        int c = r.GetNumerator();
        int d = r.GetDenominator();
        return Rational(a * c, b * d);
    }

private:
    int numerator;
    int denominator;
}

// Example
Rational t;
Rational u;
t.Multiply(u);
```
Insertion Facilitator

void Rational::Insert(ostream &sout) const {
    sout << GetNumerator() << '/' << GetDenominator();
    return;
}

- Example
  t.Insert(cout);

- Why is sout a reference parameter?
Basic Extraction Facilitator

```cpp
void Rational::Extract(istream &sin) {
    int numer;
    int denom;
    char slash;
    sin >> numer >> slash >> denom;
    assert(slash == '/');
    SetNumerator(numer);
    SetDenominator(denom);
    return;
}

● Example
    t.Extract(cin);
```
Auxiliary Arithmetic Operators

Rational operator+(const Rational &r, const Rational &s) {
    return r.Add(s);
}

Rational operator*(const Rational &r, const Rational &s) {
    return r.Multiply(s);
}

- Example
  cout << (t + t) * t;
Auxiliary Insertion Operators

```cpp
ostream& operator<<(ostream &sout, const Rational &r) {
    r.Insert(sout);
    return sout;
}
```

- Why a reference return?
- Note we can do either
  
  ```cpp
t.Insert(cout); cout << endl; // unnatural
cout << t << endl; // natural
  ```
Auxiliary Extraction Operator

// extracting a Rational
istream& operator>>(istream &sin, Rational &r) {
    r.Extract(sin);
    return sin;
}

● Why a reference return?

● We can do either

    t.Extract(cin);
    cin >> t; // unnatural

    // natural