CS 247: Software Engineering Principles

ADT Design

Readings: Eckel, Vol. 1
- Ch. 7 Function Overloading & Default Arguments
- Ch. 12 Operator Overloading

- domain specific types
- safety (tool supported)
- evolvability
- code efficiency (wrt error checks)

Outside-in ADT design
- use/test scenarios
↓
interface
↓
data representation implementation

Abstract Data Types (ADTs)

An abstract data type (ADT) is a user-defined type that bundles together

- the range of values that variables of that type can hold
- the operations that manipulate variables of that type.

Provides compiler support for your restrictions on values and operations - turns programmer errors into type errors (checked by the compiler)

Can change value range, data representation without changing client source code

Rational ADT

- arithmedic operators (+, -, *, /, ...)
- streaming operators (<<, >>)
- formats
- reduction
- copy
- serialization into bytes
- comparison operators (==, !=, <, >, ...)

Client Code for Rational ADT

```
#include <iostream>
using namespace std;

int main () {
  Rational r, s;

  cout << "Enter rational number (a/b): ";
  cin >> r;
  cout << "Enter rational number (a/b): ";
  cin >> s;

  Rational t(r+s);

  cout << r << " + " << s << " = " << r+s << endl;
  cout << r << " * " << s << " = " << r*s << endl;
  cout << r << " == " << s << " is " <<  (r==s) << endl;

  return 0;
}
```

1. Legal Values

```
class Rational {
public:
  Rational ();  // == 0/1
  Rational (int num, int denom) throw (char const*);
  explicit Rational (int num);  // == num/1
private:
  int numerator_;
  int denominator_;
```

A constructor initializes new object to a legal value

2. Public Accessors and Mutators

```
class Rational {
public:
   int numerator() const;
   int denominator() const;
   void numeratorIs( const int );
   void denominatorIs( const int ) throw (char const*);
```

Accessors and mutators provide restricted read/update access to data members

- want some naming convention

Best practice: Mutators check that client-provided values are within ADT value range

Best practice: Whenever possible,

- pass parameters by `const` reference
- use `const` member functions

3. Function Overloading

```
class Rational {
public:
  Rational ();                        // default value == 0/1
  Rational (int num);                 // value = num/1
  Rational (int num, int denom);      // value = num/denom
```

Function overloading allows you to use the same function name for variants of the same function.

- functions must have different argument signatures
- cannot overload functions that differ only by return type

4. Default Arguments

```
class Rational {
public:
  Rational ();                          // default value == 0/1
  Rational (int num);                   // value = num/1
  Rational (int num, int denom);        // value = num/denom

  explicit Rational (int num = 0, int denom = 1);
```

Use default arguments to combine variants that vary in user-provided arguments.

- must appear only in the function declaration
- only trailing parameters may have default values
- once one default argument is used in a function call, all subsequent arguments in call must be defaults

5. Operator Overloading

```
Rational operator+ ( const Rational &r ) const {
    return Rational( numerator() * r.denominator() + denominator() * r.nominator(),
           denominator() * r.denominator() );
}

bool operator== ( const Rational &r ) const {
    Rational a( this->reduce() );
    Rational b( r->reduce() );

    return ( (a.numerator()==b.numerator()) && (a.denominator()==b.denominator() );
}
```

Design Decision: signature of the operator

- argument types, return type, const, pass-by-value/pass-by-reference

Best Practice: use operator signatures that the client programmer is used to

(e.g., `operator==` returns a `bool`)

- Cannot create new operations (e.g., operator**)
- Cannot change the number of arguments

Member vs Nonmember Function

( ) some must be member functions (=, [ ])
constructors, destructors, virtual functions, some operators
access, mutators

- " " " non-member "
- first arg not the ADT being defined
- allow compiler to perform type conversion on first arg

6. Nonmember Functions

```
class Rational {
…
};

// Arithmetic Operations
Rational operator+ (const Rational&, const Rational&);
Rational operator* (const Rational&, const Rational&);

// Comparison Operations
bool operator== (const Rational&, const Rational&);
bool operator!= (const Rational&, const Rational&);
```

A nonmember function is a critical function of the ADT that is declared outside of the class.

- reduces number of functions with direct access to private data members
- some functions have to be nonmember functions (e.g., operator>>)


```
Rational r,
r = r+2; // not error if + is member function
r = 2+r; // error if + is member function
bool t = (s == 2);
```

ostream& operator<< (ostream& sout, const Rational& r) {
  return sout << r.numerator() << "/" << r.denominator();
}

operator>>, operator<<

```
class Rational {
   friend ostream& operator<< (ostream&, const Rational&);
   friend istream& operator>> (istream&, Rational&);
   ...
};
ostream& operator<< (ostream &sout, const Rational &r);
istream& operator>> (istream &sin, Rational &s);
```

Best Practice: Streaming operators should be nonmember functions, so that first operand is reference to stream object.

```
// example client code:
cout << r << " + " << s << " = " << r+s << endl;
```

Best Practice: Return value is modified stream, so that stream operations can be chained.

7. Type Conversion of ADT objects

```
explicit Rational (int num=0, int denom=1);
```

The compiler uses constructors that have one argument to perform implicit type conversion.

Also true of constructors that have more than one argument, if rest of arguments have default values.

Can prohibit this use of constructors via keyword `explicit`:

8. Private Data Representation

```
class Rational {
public:
  Rational( int num=0, int den=1 );
  int numerator() const;
  int denominator() const;
  void numeratorIs( const int );
  void denominatorIs( const int ) throw (char const*);
private:
  int numerator_;
  int denominator_;
};
```

Best Practice: Data members should be private, always.

- ease evolution of the data representation
- client code can access data via public accessors, mutators
- derived classes can manipulate data via public or protected methods
- nonmember functions can manipulate data via public methods or by being a friend

Friends

```
// Alternate implementation
class Rational {
    friend istream& operator>> (istream&, Rational&);
    ...
};

istream& operator>> (istream &sin, Rational &r) {
    char slash;
    sin >> r.numerator_ >> slash >> r.denominator_;
    return sin;
}
```

Sometimes we want the default access to be private, but to grant access to select code (e.g., class-related nonmember functions).

- Friends are easier to track than family (when code changes)

9. Helper Functions

```
class Rational {
…
private:
    void reduce();
    static int gcd( int, int );
};
```

Best Practice: Hide helper functions as private methods or within a namespace

- helper functions modularize code that is common among multiple class-related functions
- but should not pollute the global namespace

10. Override / Final

```
class A : B {
   virtual int foo ( ) const final;
   void bar ( ) const override;
};

class Rational final {
…
};
```

Applying `override` to a method signifies that the method overrides a virtual function in the base class.

Applying `final` to a virtual function prevents the function from being overridden in derived classes.

`final` may also be applied to a class, in which case the class is prohibited from being used as a base class.

Rational

```
class Rational final {
public:
   Rational( int num=0, int den=1 );
   int numerator() const;
   int denominator() const;
   void numeratorIs( const int );
   void denominatorIs( const int );
private:
   int numerator_;
   int denominator_;
   void reduce();
   static int gcd ( int, int );
};

bool operator== ( const Rational&, const Rational&);
bool operator<> ( const Rational&, const Rational&);
bool operator<  ( const Rational&, const Rational&);
bool operator<= ( const Rational&, const Rational&);
bool operator>  ( const Rational&, const Rational&);
bool operator>= ( const Rational&, const Rational&);
Rational operator+ ( const Rational&, const Rational&);
Rational operator* ( const Rational&, const Rational&);
istream& operator>> ( istream&, Rational& );
ostream& operator<< ( ostream&, Rational& );
```

Summary of Attribute-based ADT Design

Range of legal values

- Default initial value?
- Construct object from client-provided values?
- `explicit` constructors or allow type conversion?
- Throw exception if constructor or mutator is given an illegal value.

Attributes (virtual data members)

- Accessors, mutators; consistent naming scheme

Nonmember functions

- Overloaded functions; default arguments
- Overloaded operators

Final

- Should the client programmer be able to create derived classes?