CS 247: Software Engineering Principles

ADT Design

Readings: Eckel, Vol. 1

Ch. 7 Function Overloading & Default Arguments

Ch. 12 Operator Overloading

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

Improves code efficiency -- can limit range checks to constructors, mutators

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 {
private:
  // numerator / denominator
public:
  Rational ();
  Rational (int num, int denom) throw (char const*);
```

A constructor initializes new object to a legal value

- constructor throws an exception if initial value is illegal

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) throw (char const*);
```

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

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

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

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

6. Nonmember Functions

```
// 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.

- better encapsulation of private data members
- more flexible packaging of data+functions via namespaces (future topic)
  - functions can be distributed among several files
  - client can import subset of functions
  - client can easily extend class's functionality by adding functions to namespace
- some functions have to be nonmember functions (e.g., operator>>)

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.

```
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;
protected:
  void numeratorIs( const int );
  void denominatorIs( const int ) throw (char const*);
private:
  int numerator_;
  int denominator_;
};
```

Best Practice: Data members should be private, always.

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

```
private:
    void reduce();
}
int gcd( int, int );
```

Helper functions modularize code that is common among multiple class-related functions

- protect within a namespace, or declare as private methods

Summary of Design Decisions

- Range of legal values
- Default initial value (if any)?
- Construct object from client-provided values?
- `explicit` constructors or allow type conversion?
- Use compiler-generated default constructor, destructor
- - eventually, also consider compiler-generated copy constructor, assignment
- Attributes, accessors, mutators; (names of member functions)
- What if constructor is given, or mutator generates, an illegal value?
- Overloaded functions; default values
- Overloaded operators
- Member vs nonmember functions
- Access specifiers for member functions; friends