Structuur van Computerprogramma’s 2

dr. Dirk Deridder
Dirk.Deridder@vub.ac.be
http://soft.vub.ac.be/
Chapter 3 - User Defined Types
Rational Example
#include <iostream>
#include "rational.h"

using namespace std;

int main() {
    Rational leftoperand;
    Rational rightoperand;
    char operation;

    while (cin) {
        cin >> leftoperand >> operation >> rightoperand;
        switch (operation) {
            case '+':
                cout << leftoperand + rightoperand << "\n";
                break;
            case '*':
                cout << leftoperand * rightoperand << "\n";
                break;
            case '-':
                cout << leftoperand - rightoperand << "\n";
                break;
            case '/':
                cout << leftoperand / rightoperand << "\n";
                break;
        }
    }
}

2/3+5/3
7/3
1/2*5/3
5/6
5/3-2/2
2/3
2/2/2/2
1
Rational Example: `rational.h` (I)

```cpp
#ifndef RATIONAL_H
#define RATIONAL_H
#include <cassert.h>
#include <iostream>
#include "gcd.h"
using namespace std;

class Rational { //ADT representing rational numbers
public:
    Rational(int num = 0, int denom = 1) :
        numerator_(num), denominator_(denom) {
        assert(denominator_ != 0);
    }
    Rational inverse() { return Rational(denom(), num()); }
    bool isnegative() { return denom() * num() < 0; }
    void simplify() {
        int g(gcd(num(), denom()));
        numerator_ /= g;
        denominator_ /= g;
    }
    int num() { return numerator_; }
    int denom() { return denominator_; }
friend istream& // reads 2/3 as well as 4, the latter is understood as 4/1
            operator>>(istream&, Rational&);
private:
    int numerator_; // must not be 0!
    int denominator_; // must not be 0!
};
```
//overloaded arithmetic operators

inline Rational operator+(Rational r1, Rational r2) {
    return Rational(r1.num() * r2.denom() + r2.num() * r1.denom(), r1.denom() * r2.denom());
}

inline Rational operator*(Rational r1, Rational r2) {
    return Rational(r1.num() * r2.num(), r1.denom() * r2.denom());
}

inline Rational operator-(Rational r) { //unary -
    return Rational(-r.num(), r.denom());
}

inline Rational operator-(Rational r1, Rational r2) {
    return Rational(r1 + (-r2));
}

inline Rational operator/(Rational r1, Rational r2) {
    return r1 * r2.inverse();
}
//overloaded relational operators: only operator< and operator== are really necessary
//the others are automatically derived by the STL using template functions

inline bool operator<(Rational r1, Rational r2) {
    return (r1 - r2).isnegative();
}
inline bool operator==(Rational r1, Rational r2) {
    return (r1 - r2).num() == 0;
}
inline bool operator>(Rational r1, Rational r2) {
    return r2 < r1;
}
inline bool operator!=(Rational r1, Rational r2) {
    return !(r1 == r2);
}
inline bool operator>=(Rational r1, Rational r2) {
    return !(r1 < r2);
}
inline bool operator<=(Rational r1, Rational r2) {
    return !(r1 > r2);
}

//output operator, simplifies first, prints 4 for 4/1
ostream& operator<<(ostream&, Rational);
```cpp
#include <stdlib.h> // for abs(int)
#include "rational.h"

ostream& operator<<(ostream& os, Rational r) {
    r.simplify();
    os << (r.isnegative() ? "-" : "") << abs(r.num());
    if (abs(r.denom()) != 1)
        os << "/" << abs(r.denom());
    return os;
}

istream& operator>>(istream& is, Rational& r) { // reads things like 2/3, 4
    char c;
    if (!is) // return if input stream is not ok (e.g. eof)
        return is;
    is >> r.numerator_;
    r.denominator_ = 1; // default
    if (!is) // end of file after 1 number, just return and r=numerator/1
        return is;
    is.get(c); // get the next char, do not skip white noise (/n/t)
    if (c != '/') {
        // oops, not a real fraction, just return numerator/1
    cin.putback(c); // but first put back the character, so it will be read again
    return is;
    }
    is >> r.denominator_;
    assert(r.denominator_ != 0);
    return is;
}
Rational Example: gcd.h  gcd.cpp

```c
#ifndef GCD_H_
define GCD_H_

int gcd(int, int);
#endif /* GCD_H_ */

#include "gcd.h"

int gcd(int u, int v) {
    // use Euclid's algorithm to compute
    // the greatest common divisor of u,v
    if (v == 0)
        return u;
    else
        return gcd(v, u % v);
}
```
#include <iostream>
#include "rational.h"

using namespace std;

int main() {
    Rational leftoperand;
    Rational rightoperand;
    char operation;

    while (cin) { // as long as there are data on the standard input stream
        cin >> leftoperand >> operation >> rightoperand;
        switch (operation) {
            case '+':
                cout << leftoperand + rightoperand << "\n";
                break;
            case '*':
                cout << leftoperand * rightoperand << "\n";
                break;
            case '-':
                cout << leftoperand - rightoperand << "\n";
                break;
            case '/':
                cout << leftoperand / rightoperand << "\n";
                break;
        }
    }
}

2/3+5/3  2/3
7/3      5/6
1/2*5/3  5/3-2/2
2/3      2/2/2/2
1
Exceptions
Overview of Concepts

- Exception handling facility
- Exception, exception types,
- Exception throwing, exception re-throwing
- Exception handling, try/catch statement
- Catch-all exception handler
C++ provides a more gentle and cleaner way to handle errors with its exception handling facility.
When an **exceptional situation** is detected:

- a function can **throw** an object (**exception**)
  - the exception object can contain data pertinent to the situation
  - immediately exits current function without producing a return value

A calling function can **catch** exception objects of certain **exception types** using a **try statement**

- handles exceptions thrown during the execution of the **try** statement
  - first handler that matches the exception type (*) is passed the exception object
  - exception object data can be analysed to parameterise handling
  - there may be many function calls between the detection and handling level

```cpp
throw Expression;
```

**VS**

```cpp
try
    CompoundStatement
    catch ( ParameterDeclaration )
        CompoundStatement
    ...
```

- typically resulting in a class object
- multiple catch handlers allowed
class RationalZeroDenom { // an exception class
public:
    RationalZeroDenom(int num): num_(num) {} 

    int num() { return num_; }

private:
    int num_; 
};

Define a type for the category of errors you want to handle
Rational Example: Throwing Exceptions

Rational::Rational(int num = 0, int denom = 1) :
    numerator_(num), denominator_(denom) {
    if (denominator_ == 0)
        throw RationalZeroDenom(num);
}

std::istream&
operator>>(std::istream& is, Rational& r)
    throw (RationalZeroDenom, std::exception) {
    is >> r.denominator_; // Stuff omitted: see book p. 78
    if (r.denominator_ == 0)
        throw RationalZeroDenom(r.numerator_);
    return is;
}

Make a copy of the thrown object and exit the function, its caller, etc. up to a call in a try block with a catch clause matching the type of the exception
#include <iostream>
#include "rational.h"
using namespace std;

int main() {
    Rational leftoperand, rightoperand;
    char operation;
    while (cin) { // as long as there are data on the standard input stream
        try { // start try block
            cin >> leftoperand >> operation >> rightoperand;
            switch (operation) {
            case '+':
                cout << leftoperand + rightoperand << "\n"; break;
            case '*':
                cout << leftoperand * rightoperand << "\n"; break;
            case '-':
                cout << leftoperand - rightoperand << "\n"; break;
            case '/':
                cout << leftoperand / rightoperand << "\n"; break;
            }
        } // end try block
        catch (RationalZeroDenom ex) { // catch n/0 exceptions
            cerr << "Bad input \"" << ex.num() << "/0\": denominator must be non-zero."
                 << endl
                 << "Try again.\" << endl;
        }
        catch (...) { // catch all other exceptions
            cerr << "Unknown exception" << endl;
        }
    }
    "catch all" handler

1/2+3/0
Bad input '3/0': denominator must be non-zero.
Try again.
1/2+3/10
4/5

The order of the handlers is important!
Run-time Behaviour: Throwing an Object E

1. **A copy of E is made**, as if E were a call-by-value parameter of a function call (cctor is used)

2. The **current function call exits immediately** (without returning a value)
   - unless the throw statement occurs inside a try block

3. The **call stack is unwound** by popping frames from active function calls (including the call that generated E)
   - Each time a frame is popped: all destructors for local objects in the frame are executed
   - Unwinding stops when the top of the stack contains a frame for an active function call which is executing a statement in a try block

4. **Control is transferred** out of the try block to one of the following **handlers**
   - Each handler can be seen as an overloaded unary function
   - The first handler that is a match for the type of E is executed with E as parameter
     - If no handlers match, the frame is popped and stack unwinding continues
     - If all frames of the stack are popped, the program exits abnormally calling `std::terminate()`

5. After executing the handler, the **try statement is finished** and the execution proceeds as normal
   - You can rethrow an exception using the statement `throw;`
Catching Exceptions: \texttt{catch(arg)} is like a function call

while (std::cin) {
    try {
        // catch any exceptions thrown in (functions called from within) this block

        Rational r;
        std::cout << "input? " << std::endl;
        std::cin >> r;
        // ...
    }
    catch (RationalZeroDenom& e) {    // Complain and continue
        std::cerr << e << ", try again" << std::endl;
    }
    catch (std::exception& e) {    // Complain and throw it again
        std::cerr << e.what() << std::endl;
        throw;  // Re-throw e.
    }
    catch (...) {    // Complain and throw it again
        std::cerr << "A weird exception was thrown" << std::endl;
        throw;
    }
}
class RationalZeroDenom {
public:
    RationalZeroDenom(int n) : num_(n) { }

    friend std::ostream& operator<<(std::ostream& os, const RationalZeroDenom& e) {
        return os << num_ << "/0 is not a legal Rational";
    }

private:
    int num_;  
};

// ...
catch (RationalZeroDenom ex) { // catch n/0 exceptions
    cerr << ex;
}
// ...
More on Exceptions

• When unwinding the stack, local objects are destructed
  • release resources in destructors

• When throwing an exception from a constructor `C::C(...)`, the destructor `C::~C()` is not called (but the destructors of the data members are)

• Exceptions thrown from a destructor: see book p. 212-213

• Exception specifications and unexpected exceptions: see book p. 213-214