CS 247: Software Engineering Principles

Exceptions, Exception Safety

Reading: Eckel, Vol. 2
    Ch. 1   Exception Handling

Robust Code can be Messy

```
ifstream infile; // ifstream used to read from file

infile.open("data");  // opens file "data" on disk
if ( !infile.good() ) {
   cout << "Error opening input file data." << endl;
   exit(1);
}

while ( !infile.eof() ) {
   String name;
   infile >> name;
   if ( infile.bad() ) {
      // react to failure (stream is likely damaged)
   }
   else if ( infile.fail() ) {
      // react to failure (likely a format error)
   }
   else {
      // process data
   }
}
```

Robust Code can be Messy

```
ifstream infile; // ifstream used to read from file

infile.open("data");  // opens file "data" on disk
if (!infile) {
   cout << "Error opening input file data." << endl;
   exit(1);
}

while (!infile.eof()) { true
   String name;
   infile >> name;
   if (infile.fail()) {
      // react to read failure (likely format error)
   }
   else if (infile.bad()) {
      // react to read failure (likely some data lost)
   }
   else {
      // process data
   }
}
```

Exceptions

Exception - an unusual event or situation that precludes a function from completing normally

- i.e., keeps a function from satisfying its postcondition
- not a programmer error

C++ Exception Handling

- Separates normal code from error-handing code
- Separates risk-free code from risky code
- Allows different parts of a program to detect vs. recover from an exception
- Errors cannot be ignored (cf. error codes)

Example: RationalExceptional.cpp

Throwing an Exception

`throw` an exception if the function cannot complete its normal execution (i.e., satisfy its postcondition).

- Creates the exception object
- Transfers control to a handler (determined by the type of the exception object)
- Exits any functions along the call chain to the matching handler
- Calls destructors for local variables of exited functions

Catching an Exception

Exception is handled by the "nearest" handler whose argument matches the type of the exception object.

Local exception: exception is handled in the same routine as it is thrown

e.g., through an alternative computation or return value.

Otherwise, the exception is caught by the nearest (dynamically) matching catch-clause whose try-block encloses the throw.

If there is no matching handler, then the program aborts.

Stack Unwinding

As control transfers to an exception handler, it must pop off the call stack all of the scope blocks that reside on top of the matching handler.

- Objects in the stack are deleted for us automatically, invoking the appropriate destructors.
- Partially constructed stack-based objects (e.g., objects from failed constructors) are properly destroyed.
- If, during stack unwinding, a destructor throws an exception that it does not handle locally, the program terminates.

What about heap-based objects?

Exception Specifications

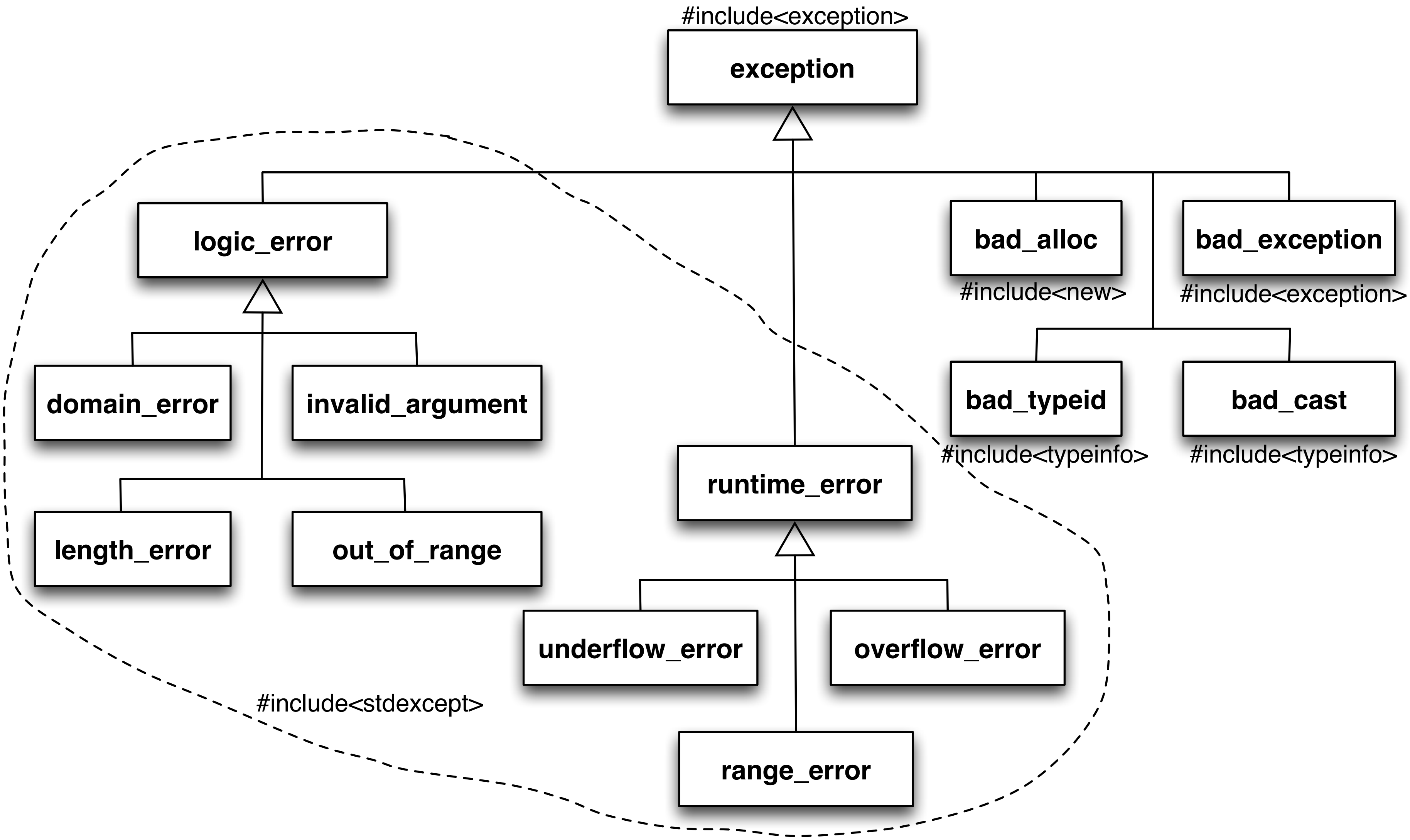
Functions may declare the set of exceptions that it might throw back to the caller (called a throw list).

```
double safeDivide (int numerator, int denominator)
                           throw (DivideByZero, UnderFlow);


void f(); // no exception specification -- could throw anythin


void g() throw();  // specifies that no exceptions are
thrown
```

Best Practice: Don't use exception specifications (but you should know what they are because you might encounter them)

Standard Exceptions

Programmers can define their own exception hierarchy, or derive new exceptions from the standard exception hierarchy.



Standard Exception Declarations

```
class exception {
public:
  exception () throw();
  exception (const exception&) throw();
  exception& operator= (const exception&) throw();
  virtual ~exception() throw();

  virtual const char* what() const throw();
}


class runtime_error : public exception {
public:
  explicit runtime_error (const string& what_arg);
};
```

```
#include <stdexcept>
#include <climits>

Rational operator* (const Rational &r, const Rational &s) {

    long numer = (long)(r.numerator() * s.numerator());
    long denom = (long)(r.denominator() * s.denominator());

   // proceed if value is within int range
   if ( numer <= INT_MAX && denom <= INT_MAX )
     return Rational( numer, denom );
   throw runtime_error("Product exceeds INT_MAX");
 }


int main() {
   Rational r(2000000000);
   Rational s(2000000000);
   try {
     Rational t(r*s);
   }
   catch ( runtime_error &e ) {
     cout << e.what() << ": " << r << " * " << s << " << endl;
   }
}
```

The RAII Programming Idiom

Resource Acquisition is Initialization (RAII) -- equates resource management with the lifetime of an object

- resource is allocated inside an object's constructor
- resource is deallocated inside an object's destructor

```
class Resource {
    resource_type *r_;
    resource_type* allocate( parms p );
    void release ( resource_type* );
public:
    Resource (parms p) : r_( allocate(p) ) { }
    ~Resource() { release(r_); }
    // copy constructor, assignment
    // accessors, mutators
};
```

Smart Pointers

A smart pointer is an ADT that simulates a pointer but uses RAII to provide automatic deallocation of referent.

Example: auto_ptr<> is a templated object with an internal pointer variable that

(1) helps to ensure one auto_ptr owner of referent
(2) allows transfer of ownership
  (Boost has smart pointers that do not support transfer of ownership)
(3) automatically deallocates object at the end of the pointer's scope.

Will eventually be replaced by

unique_ptr<> — one owner of referent, transferable ownership
shared_ptr<> — multiple ownership, object deallocated when reference count reaches zero

Example: RationalAutoPtr.cpp

auto_ptr

```
template <class X>  // partial def
class auto_ptr {
public:
    typedef X element_type; // holds smart pointer to X

    // construct/copy/destroy:
    explicit auto_ptr(X* p =0) throw();
    auto_ptr(auto_ptr&) throw();            // transfers ownership
    auto_ptr& operator=(auto_ptr&) throw(); // transfers ownership
    ~auto_ptr() throw();

    // members:
    X& operator*() const throw();    // access referent
    X* operator->() const throw();   // access member of referent
    X* get() const throw();        // returns pointer
    X* release() throw();          // unbinds auto_ptr, returns ptr
    void reset(X *p =0) throw();   // destroys old referent and
                                      binds auto_ptr to *p
};
```

Summary

When to use exceptions

- Not for programming errors (use assertions)
- Often not needed if simply terminating program
  - Can let system clean up
- Throw exceptions in constructor, mutators
- Don't throw exceptions in destructors, copy constructors, assignment, accessors, operators

How to use exceptions

- Nest exception class declarations inside class descriptions
- Use class hierarchies (for polymorphic handlers)
- Catch exceptions by reference, not by value (polymorphism)
- Use RAII, smart pointers to perform automated cleanup
- Current wisdom says to not use exception specifications

```
#include <stdexcept>
#include <climits>

Rational operator* (const Rational &r, const Rational &s) {

    long numer = (long)(r.numerator() * s.numerator());
    long denom = (long)(r.denominator() * s.denominator());

   // proceed if value is within int range
   if ( numer <= INT_MAX && denom <= INT_MAX )
     return Rational( numer, denom );
   throw runtime_error("Product exceeds INT_MAX");
 }


int main() {
   Rational r(2000000000);
   Rational s(2000000000);
   try {
     Rational t(r*s);
   }
   catch ( runtime_error &e ) {
     cout << e.what() << ": " << r << " * " << s << " " << endl;
   }
}
```

```
#include <stdexcept>
#include <climits>

Rational operator* (const Rational &r, const Rational &s) {

    long numer = (long)(r.numerator() * s.numerator());
    long denom = (long)(r.denominator() * s.denominator());

   // proceed if value is within int range
   if ( numer <= INT_MAX && denom <= INT_MAX )
     return Rational( numer, denom );
   throw runtime_error("Product exceeds INT_MAX");
 }


int main() {
   Rational r(2000000000);
   Rational s(2000000000);
   try {
     Rational t(r*s);
   }
   catch ( runtime_error &e ) {
     cout << e.what() << ": " << r << " * " << s << " << endl;
   }
}
```