Exceptions 2

CMSC 202

Review

- Purpose of Exceptions
  - Handle Errors
- Three Key Components
  - Try
  - Catch
  - Throw
- Handle Multiple Exceptions?
  - Absolutely

Exception Classes

- Name?
  - Reflects error, ***not*** code that throws error
- Data?
  - Basic information or a message
    - Parameter value
    - Name of function that detected error
    - Description of error
- Methods?
  - Constructor (one or more)
  - Accessor (one or more)

Exception Examples

```
// Good example of an Exception Class
class NegativeParameter
{
    public:
        NegativeParameter(  const string& parameter,
                int value );
        int GetValue ( ) const;
        const string& GetParameter( ) const;
    private:
        int m_value;
        string m_paramName;
};

// Trivial example of an Exception Class
class MyException { };
```

Code that catches this exception gets the parameter name and its value

Code that catches this exception gets no other information – just the "type" of exception thrown

Exception Specification

Functions/Methods can specify which exceptions they throw (or that they don't throw any)

Syntax:

```
// Throws only 1 type of exception
retType funcName( params ) throw (exception);

// Throws 2 types of exceptions (comma separated list)
retType funcName( params ) throw (exception1, exception2);

// Promises not to throw any exceptions
retType funcName( params ) throw ( );

// Can throw any exceptions [backwards compatibility]
retType funcName( params );
```

Specification Example

```
// Divide() throws only the DivideByZero exception
void Divide (int dividend, int divisor ) throw (DivideByZero);


// Throws either DivideByZero or AnotherException
void Divide (int dividend, int divisor ) throw (DivideByZero,
                        AnotherException);

// Promises not to throw any exception
void Divide (int dividend, int divisor ) throw ( );

// This function may throw any exception it wants
void Divide (int dividend, int divisor );
```

Exceptions in Constructors

Best way to handle Constructor failure
- Replaces Zombie objects!
- Any sub-objects that were successfully created are destroyed (destructor is ***not*** called!)

Example:

```
// MyClass constructor
MyClass::MyClass ( int value )
{
   m_pValue = new int(value);

   // pretend something bad happened
   throw NotConstructed( );
}
```

Exceptions in Destructors

Bad, bad idea…
- What if your object is being destroyed in response to another exception?
  - Should runtime start handling your exception or the previous one?

General Rule…
- Do not throw exceptions in destructor

Standard Library Exceptions

#include <stdexcept>
- **bad_alloc**
  - Thrown by new when a memory allocation error occurs
- **out_of_range**
  - Thrown by vector's at( ) function for an bad index parameter.
- **invalid_argument**
  - Thrown when the value of an argument (as in vector< int > intVector(-30);) is invalid

Derive from `std::exception`
- Define own classes that derive…

Exceptions and Pointers

What happens to the Fred object?

```
void myNew1stFunction( )
{
     // dynamically allocate a Fred object
     Fred *pFred = new Fred;

     try {
         // call some function of Fred
     }
     catch( ... )              // for all exceptions
     {
    throw;                // rethrow to higher level
     }

     // destroy the Fred object
     // if no exception and normal termination
     delete pFred;
}
```

One Solution…

What's not so good about this solution?

```
void myNew1stFunction( )
{
     // dynamically allocate a Fred object
     Fred *pFred = new Fred;

     try {
         // call some function of Fred
     }
     catch( ... )              // for all exceptions
     {
         delete pFred;         // delete the object
         throw;                // rethrow to higher level
     }

     // destroy the Fred object
     // if no exception and normal termination
     delete pFred;
}
```

Duplicated Code!

Better Solution

- auto_ptr
  - A kind of "smart pointer"
  - Takes ownership of dynamic memory
  - Frees memory when pointer is destroyed
    - #include <memory>
- Control can be passed…
  - Between auto_ptrs and other auto_ptrs
    - Only one auto_ptr can own an object at a time
  - Between auto_ptrs and normal pointers

Auto_ptr Example

```
#include <memory>
void my2ndFunction( )
{
     Fred* p1 = new Fred;        // p1 "owns" the Fred object

     auto_ptr<Fred> p2( p1 );    // pass ownership to an auto_ptr

     // use the auto_ptr the same way
     // we'd use a regular pointer
     *p2 = someOtherFred;        // same as "*p1 = someOtherFred;"
     p2->SomeFredFunction();     // same as "p1->SomeFredFunction();"

     // use release() to take back ownership
     Fred* p3 = p2.release();    // now p3 "owns" the Fred object

     delete p3;                  // need to manually delete the Fred

     // p2 doesn't own any Fred object, and so won't try
     // to delete one
}
```

Multiple Auto_ptrs

```
void my3rdFunction()
   {
     auto_ptr<Fred> p1( new Fred );
     auto_ptr<Fred> p2;

     p1->SomeFredFunction( ); // OK

     p2 = p1;             // p2 owns Fred object and p1 does not

     p2->SomeFredFunction( ); // OK

     // watch for this pitfall
     p1->SomeFredFunction( ); // error! p1 is a null pointer

     // when p1 and p2 go out of scope
     // p2's destructor deletes the Fred object
     // p1's destructor does nothing
   }
```

Exception-Safe Code

- Fundamentals
  - Exception-safe code
    - Leaves the object (or program) in a consistent and valid state
  - Hard to do well
  - Think through even simple code thoroughly…

Exception-UNsafe Example

```
// unsafe assignment operator implementation
FredArray& FredArray::operator=( const FredArray& rhs)
{
   if ( this != &rhs )
   {
      // free existing Freds
      delete [] m_data;                       // 1

      // now make a deep copy of the right-hand object
      m_size = rhs.m_size;                    // 2
      m_data = new Fred [ m_size ];           // 3

      for (int j = 0; j < m_size; j++ )       // 4
          m_data[ j ] = rhs.m_data [ j ];     // 5
   }
   return *this;
}
```

Exception-safe Example

```
// Better assignment operator implementation
FredArray& FredArray::operator=( const FredArray& rhs)
{
   if ( this != &rhs )
   {
      // code that may throw an exception first
      // make a local temporary deep copy
      Fred *tempArray = new Fred[rhs.m_size];
      for ( int j = 0; j < rhs.m_size; j++ )
          tempArray[ j ] = rhs.m_data [ j ];

      // now code that does not throw exceptions
      delete [] m_data;
      m_size = rhs.m_size;
      m_data = tempArray;
   }
   return *this;
}
```

Rule of Thumb:

Do any work that may throw an exception off "to the side" using local variables

THEN change the object's state using methods that DON'T throw exceptions

How could we improve this?

Exception Guarantees

Each function/method can guarantee one of the following when an exception occurs:

- **Weak Guarantee**
  - Program/Object will not be corrupt
  - No memory is leaked
  - Object is usable, destructible
- **Strong Guarantee**
  - Function has no effect
  - Program/Object state is same as before
- **NoThrow Guarantee**
  - Function ensures no exceptions will be thrown
    - Usually: destructors, delete and delete[ ]

Which should you follow?

- C++ library:
  - A function should always support the ***strictest*** guarantee that it can support without ***penalizing*** users who don't need it

Challenge

- Assume that our Board constructor throws an exception if the file is not properly formatted
  - Insufficient rows and columns
  - Extra rows or columns
  - Unknown character in Board file
  - Etc.
- Use an Exception-safe strategy for building the board