

Exceptions
CMSC 202

Exceptions

Common runtime errors:

- Memory allocation error when using **new**
- File open error
- Out of bounds array subscript
- Division by zero
- Function Preconditions not met

Error Handling Techniques

assert (condition)
if the condition is false, the program terminates

Ignore the error or try to handle the error internally
devastating for real products, but maybe okay for your own software

Set an indicator for other code to detect (e.g., return a flag)

Issue an error message and exit

Error Handling, Currently

Commonly, error handling is interspersed

Advantage

Error processing close to error

Disadvantage

Code cluttered from error processing

Application cannot handle error as it wants to

Layering, Encapsulation

Low-level code should **not** process errors

Low-level code **should** alert high-level code

High-level code **should** handle errors

Fundamental Issue

Class user may handle error in any way

Exit program

Output message & continue

Retry function

Ask user what to do

...

Class implementer can't know which the user of class wants

Exception Handling

New Strategy

Low-level code detects error

"Throws" error to higher level code

High-level code processes error

Positives

Code that caused error loses control

Catch all kinds of errors

Usually used in recoverable situations

Exception Syntax

Three primary components:

Try/catch block

```
try {
    // some code to try
}
catch (ObjectType& obj) {
    // handle the error, if any
}
```

Throwing an exception

```
throw ObjectType(parameters);
```

Specifying which exceptions a function throws

```
void funcName(parameter) throw ObjectType { }
```

Simple Throw

```
double quotient(int num, int den) {
    if (den == 0)
        throw "Error: Divide by Zero";
    return static_cast<double>(num) / den;
}
```

```
int main() {
    try {
        cout << quotient(7, 0) << endl;
    }
    catch (string& e) {
        cout << e << endl;
    }
    return 0;
}
```

Throwing an Exception

```
class DivByZeroEx
{
public:
    DivByZeroEx ( ) : m_message ("divide by 0") { /* no code */ }

    const string& what ( ) const { return m_message; }

private:
    const string m_message;
};

double quotient(int num, int den)
{
    if (den == 0)
        throw DivByZeroEx();

    return static_cast<double>(num) / den;
}
```

Catching an Exception

```
int main()
{
    int numerator, denominator;
    double result;

    cout << "Input numerator and denominator" << endl;
    cin >> numerator >> denominator;

    try {
        result = quotient(numerator, denominator);
        cout << "The quotient is: " << result << endl;
    }
    catch (DivByZero& ex) { // exception handler
        cerr << "Exception occurred: " << ex.what() << endl;
    }

    // code continues here

    return 0;
}
```

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)

**** Obsolete in C++11 ****

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 );
```

Multiple Catch Blocks...Yes!

```
try
{
    // code that might throw an exception
}
catch (ExceptionObject1& ex1)
{
    // exception handler code
}
...
catch (ExceptionObject2& ex2)
{
    // exception handler code
}
catch (ExceptionObjectN& exN)
{
    // exception handler code
}
catch (...)
{
    // default exception handler code
}
```

Most Specific

Multiple catch blocks - catch different types of exceptions!

What's this?
Catch Everything Else

Least Specific

Nested Functions?

```

// function2 throws an exception // main calls function1,
void function2( )                // with try/catch
{
    cout << "function2" << endl;
    throw int(42);
}
// function1 calls function2,
// but with no try/catch
void function1( )
{
    function2( );
    cout << "function1" << endl;
}

int main( )
{
    try {
        function1( );
    }
    catch (int)
    {
        cout << "Exception "
            << "occurred"
            << endl;
    }
    return 0;
}

```

Stack is unwound until something catches the exception OR until unwinding passes main
 What happens then?

Rethrowing Exceptions

What if current scope shouldn't or can't handle error?
 Re-throw error to next scope up the stack

```

try {
    // code that could throw an exception
}
catch (someException &e){
    throw; // rethrow the exception to the next
} // enclosing try block

```

Rethrow Example

```

Application program
// handles exception if full

Add item to inventory
// rethrows exception if full

Insert item in list
// rethrows exception if full

Is list full?
// throws exception if full

```

How might we have used this in one of our past projects?

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

```

void myNewstFunction( )
{
  // 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;
}

```

What happens to the Fred object?

One Solution...

```

void myNewstFunction( )
{
  // 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;
}

```

What's not so good about this solution?

Duplicated Code!

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 ) // this points to current object
    {
        // 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 using local variables
THEN change the object's state using methods that DON'T throw exceptions

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

Practice

Write a function to Sort a vector of integers

 If the vector has no elements

 Throw an exception

 Use the message "Error: The vector is empty"

Write a main function that will:

 Create a vector

 Catch the error
