# Exceptions 1

CMSC 202

## Warmup

```
class A
{
public:
     virtual void Foo( )
     { cout << "A in Foo!" << endl; }

     void Bar( )
     { cout << "A in Bar!" << endl; }
protected:
     int val;
};

class B : public A
{
public:
     void Bar( )
     { cout << "B in Bar!" << endl; }
};
```

```
int main ( )
{
     A *a1 = new B;
     a1->Foo();
                              // A in Foo!
     B *b1 = new A;
     b1->Foo();
                              // Error!
     A *a2 = new B;
     a2->Bar();               // A in Bar!

     B b2;
     cout << b2.val;          // Error!

     B *b3 = new B;           // B in Bar!
     b3->Bar();
                              // B in Bar!
     B b4;
     b4.Bar();

     return 0;
}
```

## Common Errors (Runtime)

Memory allocation error when using **new**

File open error

Out of bounds array subscript

Division by zero

Function PreConditions not met

## Error Handling Techniques

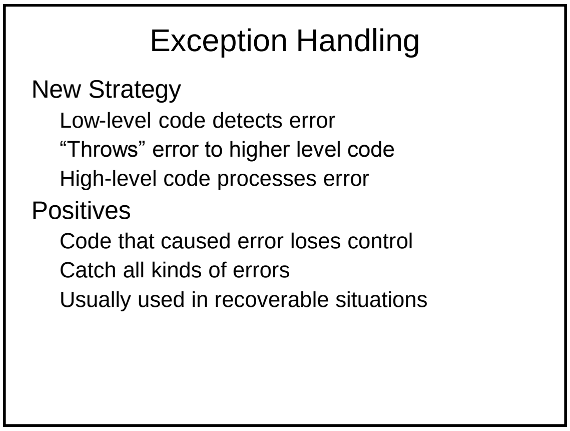
- assert (condition)
  - if the condition is false, the program terminates
- Ignore the error or try to handle the error internally
  - devastating for real products, but maybe okay for your own software
- Set an indicator for other code to detect (e.g., return a flag)
- Issue an error message and exit

## 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

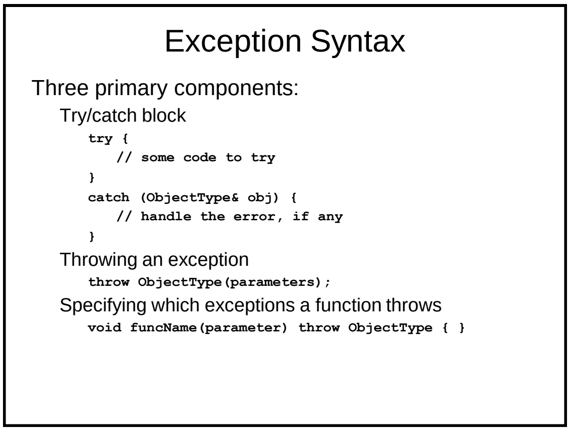
- Class user may handle error in any way
  - Exit program
  - Output message & continue
  - Retry function
  - Ask user what to do
  - …
- Class implementer can't know which the user of class wants



## 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

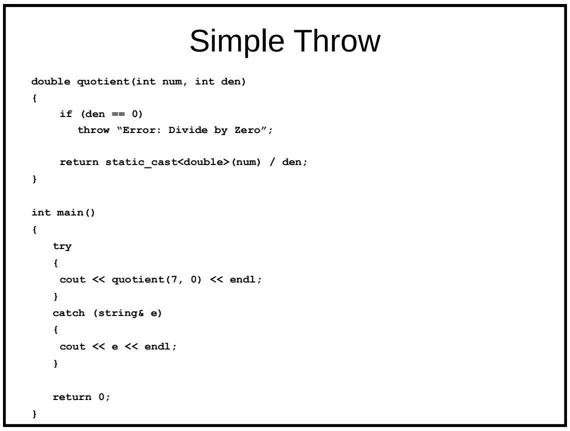
```
try {
    // some code to try
}
catch (ObjectType& obj) {
    // handle the error, if any
}
```

- 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

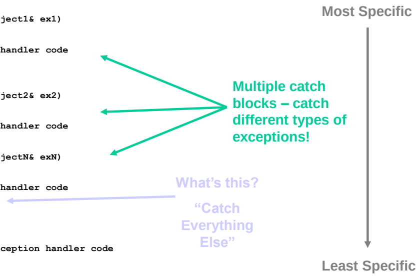
```
class DivByZeroEx
{
public:

    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 (DivByZeroEx& ex) {      // exception handler
        cerr << "Exception occurred: " << ex.what() << endl;
    }

    // code continues here

   return 0;
}
```

# 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
void function2( )
{
     cout << "function2" << endl;
     throw int(42);
}

// function1 calls function2,
// but with no try/catch
void function1( )
{
     function2( );
     cout << "function1" << endl;
}
```

```
// main calls function1,
// with try/catch
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/can’t handle error?

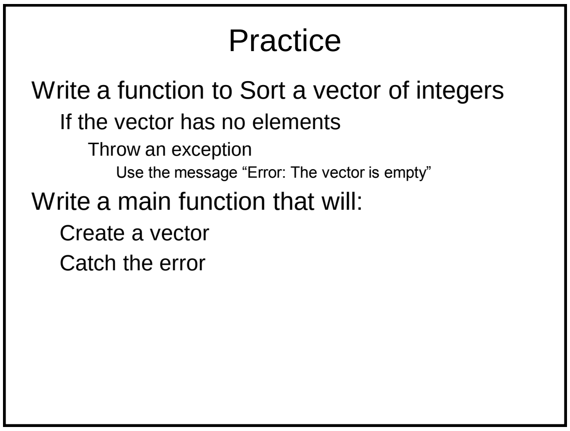
- Re-throw error to next scope up the stack

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

## Rethrow Example

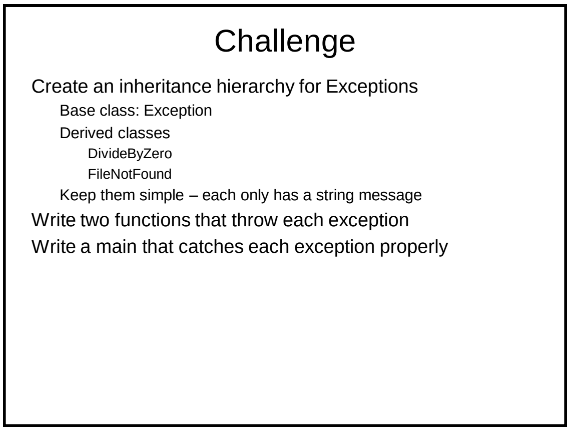
Application program
- // handles exception if full
- Add item to inventory
  - // rethrows exception if full
  - Insert item in list
    - // rethrows exception if full
    - Is list full?
      - // throws exception if full

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



## 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



## Challenge

- Create an inheritance hierarchy for Exceptions
  - Base class: Exception
  - Derived classes
    - DivideByZero
    - FileNotFound
  - Keep them simple – each only has a string message
- Write two functions that throw each exception
- Write a main that catches each exception properly