

Lecture 15

Exceptions I

Exceptional Events

- Sometimes, bad stuff happens...

```
int main()
{
    int arg1, arg2;
    cout << "Enter two numbers and I will add them..." << endl;
    cout << "Number 1: ";
    cin >> arg1;
    cout << "Number 2: ";
    cin >> arg2;
    cout << "The result is : " << arg1 + arg2 << endl;
}

Enter two numbers and I will add them...
Number 1: 4.9
Number 2: 3.76
The result is 7
```

Exceptional Events

- So, you try to deal with it...
- The following will work, but assumes a console based user interface...

```
// Read in an integer. Make SURE it's an integer!
int readInt()
{
    float floatVal;
    cin >> floatVal;
    while (floatVal != (int) floatVal) // Is this an integer?
    {
        cerr << "Input an INTEGER, please... ";
        cin >> floatVal;
    }
    return (int) floatVal;
}
```

Exceptional Events

- What if we're using a GUI system?
- It would be better to simply signal the calling function that invalid input was encountered...

```
// Read in an integer. Return false if an invalid number
// was entered
bool readInt(int &returnVal)
{
    float floatVal;
    cin >> floatVal;
    if (floatVal != (int) floatVal) // Is this an integer?
        return false;
    returnVal = (int) floatVal;
    return true;
}
```

Exceptional Events

- Well, OK, that *will* work, but it's not a general purpose solution.
- Suppose we're dealing with overloaded operators where we don't have the option of passing an additional parameter...

```
// Overload [] to allow individual array element access.
int &MyIntArray::operator[](int index)
{
    // Define a bad value to return when index is invalid
    int badValue = -1;

    // Check for index validity
    if ((index >= 0) && (index < arrayLength))
        return storagePtr[index];
    else
        return badValue; // index was bad!
}
```

Exceptional Events

- Well, that solution isn't really the greatest.
- The variable badValue is local to the overloaded operator member function, and since we are returning a reference...
- We could use a global and/or static member variable...

```
// Define a bad value to return when index is invalid
int badValue = -1;

// Overload [] to allow individual array element access.
int &MyIntArray::operator[](int index)
{
    // Check for index validity
    if ((index >= 0) && (index < arrayLength))
        return storagePtr[index];
    else
        return badValue; // index was bad!
}
```

Exceptional Events

- This solution will work, but the problem is that -1 is a valid integer, so we'd never know if the return value was legitimate or signaling an error condition.
- We could set an arbitrary boolean flag in a global variable...

```
// Define a bad value to return when index is invalid
bool badIndex = false;
int badValue = -1;
// Overload [] to allow individual array element access.
int &MyIntArray::operator[](int index)
{
    badIndex = false;
    if ((index >= 0) && (index < stringLength))
        return storagePtr[index];
    badIndex = true;
    return badValue; // still need to return something!
}
```

Exceptional Events

- OK, aside from the sheer ugliness of this solution, consider the following problem...
- What happens if the array access happens in the middle of an expression?

```
// Apply secret formula to two numbers at indices n and d
bool secretFormula(MyIntArray *array, int n, int d,
                  float &result)
{
    result = array[n] / (1 + array[d]); // SECRET FORMULA !!!
    // Check for bad index
    if (badIndex)
        return false; // Signal unsuccessful operation

    // Signal that operation was successful
    return true;
}
```

Exceptional Events

- If an invalid denominator index were passed, `MyIntArray::operator[]` would have set `badIndex` to true and returned `badval` which is -1.
- And, if you were on a machine that didn't like divide by zero, you might crash before ever getting to your validity check.
- So what we have here is a solution that is ugly and doesn't protect you from all situations!
- There must be a better way!
- Enter C++ *exceptions*.
- What is a C++ exception?

C++ Exceptions

- A C++ exception is an abrupt transfer of control, usually resulting from an error condition.
- When an error condition is encountered, the programmer may choose to *throw* an exception.
- This initiates an *immediate* transfer of control. But to where?
- An assumption is made that if the programmer has chosen to throw an exception, he/she has also provided a place to *catch* the exception.
- Perhaps a simple example would help...

```
enum MathErr { noErr, divByZero, genericOverflow };
float divide(float numerator, float denominator)
{
    if (denominator == 0)
        throw divByZero;
    return numerator/denominator;
}
```

Somebody Catch Me!!!

- An assumption is made that the programmer has set up a place for exceptions to be caught when they occur.
- This is done with a *try block*.
- It looks something like this:


```
int main()
{
    try {
        cout << "3/2 is " << divide(3,2) << endl;
        cout << "2/0 is " << divide(2,0) << endl;
    }
    catch(MathErr x) {
        if (x == divByZero)
            cerr << "Divide by zero caught. " << endl;
        else cerr << "Other error caught. " << endl;
    }
}
```

Somebody Catch Me!!!

- The *try* statement simply defines a scope inside which any exceptions that occur *might* be caught by catch statements immediately following the *try*.
- The catch statement is a little more complicated.
- It's syntax is one of the following:
 - `catch(type variableName) { }`
 - `catch(...) { }`
- The first form is somewhat like a function declaration.
- You specify a variable declaration which will be instantiated by the value thrown *if and only if* that value matches (type wise) the type declared in the catch statement.
- Inside the scope of the catch, the variable declared in the catch statement is accessible as a local variable.

Somebody Catch Me!!!

- If the value thrown doesn't match (type wise) the catch statement(s) you supply, the exception is thrown up to the next try block.
- If there are no other try blocks present, the exception is handled by the runtime environment as an *unhandled exception*.
- This usually means a generic dialog box and/or program termination.
- In the case of CodeWarrior on the Mac, the program simply terminates with no notification from the runtime environment.
- Now that we've spelled it all out, let's go back to a simple example...



Demonstration #1

A Simple Exception

More About Catching

- For every `try` statement you have, you can have multiple catch statements each dealing with a separate type:

```
void executeSomeFunction()
{
    throw 1.4; // CodeWarrior represents this as a DOUBLE
}
int main()
{
    try {
        executeSomeFunction(); // Arbitrary function
    }
    catch(int x) { cerr << "Caught INTEGER: " << x << endl; }
    catch(float f) { cerr << "Caught FLOAT: " << f << endl; }
    catch(string s) { cerr << "Caught STRING: " << s << endl; }
    catch(...) { cerr << "Generic exception caught" << endl; }
}
```

More About Catching

- When deciding on which `catch()` to pass control to, the compiler does no implicit type conversion to force a match.
- Given the preceding try/catch block, the exception would be caught by the generic block and not the FLOAT block.
- Let's verify that...



Demonstration #2

Multiple Catches

More About Throwing

- Specifically, when an exception is thrown a temporary variable is created and the expression used to throw the exception is evaluated and stored in this temporary variable.
- You can cast the thrown value to force entry into a specific handler:

```
void executeSomeFunction()
{
    throw (float)1.4; // Force exception to be of type float
}
int main()
{
    try { executeSomeFunction(); }
    catch(int x) { cerr << "Caught INTEGER: " << x << endl; }
    catch(float f) { cerr << "Caught FLOAT: " << f << endl; }
    catch(string s) { cerr << "Caught STRING: " << s << endl; }
    catch(...) { cerr << "Generic exception caught" << endl; }
}
```

More About Throwing

- You may also throw user-defined types...
- You can “construct” new instances of classes right in the throw statement by calling a given type’s constructor...

```
class MyIndexError {
    MyIndexError(int i, char *msg):badIndex(i),theMsg(msg) {}
    int getBadIndex() { return badIndex; }
    string getMessage() { return theMsg; }
private:
    int badIndex;
    string theMsg;
};
int &MyIntArray::operator[](int index)
{
    if ((index < 0) || (index >= numElements))
        throw MyIndexError(index,“Index out of bounds”);
    return storagePtr[index];
}
```

More About Throwing

- Now, I can set up to catch this exception like this:

```
int main()
{
    MyIntArray testArray;
    try {
        cout << “Element 10 is ” << testArray[10] << endl;
    }
    catch(MyIndexError mie)
    {
        cerr << “Error, index ” << mie.getBadIndex() << “: ”
            << mie.getMessage() << endl;
    }
}
```

- // This will yield the message:
- Error, index 10: Index out of bounds

Who’s Got It?

- Actually, I could have set up one of four catch statements to catch exceptions of type MyIndexError.
- They are:

```
catch(MyIndexError mie){} // Copy of object thrown in mie
catch(MyIndexError &mie){} // reference of object thrown in mie
catch(MyIndexError){} // no access to object thrown
catch(...){} // no access to object thrown
```

- We mentioned earlier that if an exception wasn’t caught by the catch statements in a given try block, the runtime environment would look for any other try blocks further up the stack and try *their* catch statements.
- That would look something like this:

Who’s Got It?

```
void func1()
{
    try {
        func2();
    } catch(ArrayIndexError aie) {
        cout << “Array Index Error: ” << aie.getMsg() << endl;
    }
}
void func2()
{
    try {
        float x = divide(globalIntArray[15334],globalIntArray[1]);
    } catch(MathErr me) {
        cout << “Math Error encountered: ” << me.getMsg() << endl;
    }
}
```

- If globalIntArray is only 50 elements big, what happens?

Who’s Got It?

- By the same token, be careful of using built in types for throwing exceptions. You might just catch something you didn’t intend on!
- Suppose we had decided to use an enum to differentiate between an index error and a divide by zero error in our previous example.
- We might cast the enum to an int when throwing and implement func2() like this:

```
void func2()
{
    try {
        float f = divide(globalIntArray[15334],globalIntArray[1]);
    } catch(int x) {
        if (x == divByZero)
            cerr << “Divide by Zero caught! ” << endl;
        else cerr << “Generic exception caught: ” << x << endl;
    }
}
```

Demonstration #3

Catching More than You Expect

Even More about Throwing

- Sometimes, when catching an exception, you can only do "so much" to fix the situation.
- Consider a routine to move a robot to a series of positions. When done, you must return the robot to its original position:

```
// Some routine to read an array of Positions from the user
int getPositionSequence(Position *arrayOfPositions)
{ ... }

// Call to move robot to a specific position. If aPos is
// invalid a BadPositionException exception is thrown
void MoveRobot(Position &aPos)
{
    if (badPos(aPos))
        throw BadPositionException(aPos);
    // Continue with move logic...
}
```

Even More about Throwing

- When we execute the code which moves the robot to each successive position, we are prepared to catch a BadPositionException.
- When we catch it, we return the robot to its original position.
- But we have no concept of GUI here, how is the user notified?

```
// Move the robot to a succession of positions
void MoveRobot(Position *positions,int numPos)
{
    Position origPos = getCurrentPosition();
    try {
        for (int i=0; i<numPos; i++)
            MoveRobot(positions[i]);
    } catch(BadPositionException bpe) {
        MoveRobot(origPos);
        throw; // What does this do?
    }
    MoveRobot(origPos);
}
```

Even More about Throwing

- throw by itself simply re-throws the current exception.
- The assumption is that someone further up the chain is ready to catch it, of course!

```
// Move a robot
void MoveTheRobot()
{
    Position *positionArray;
    int numPositions = getPositionSequence(&positionArray);
    try {
        MoveRobot(positionArray,numPositions);
    } catch(BadPositionException bpe) {
        cerr << "Error: attempt to move robot to bad " <<
            << "position " << endl << "POSITION " <<
            << bpe.getBadPosition() << endl;
    }
}
```

Lecture 15

Final Thoughts