CS100
Introduction to Programming

Lecture 26: Exceptions

What is an exception?

- An exception or exceptional event is an event that occurs during the execution of a program that disrupts the normal flow of instructions
- The following will cause exceptions:
  - Accessing an out-of-bounds array element
  - Writing into a read-only file
  - Trying to read beyond the end of a file
  - Sending illegal arguments to a method
  - Performing illegal arithmetic (e.g divide by 0)
  - Hardware failures
  - …

Handling exceptions

- Basic idea:
  - Check for exceptional events
  - Deal with them
- Example of simple exception handling

```
T & operator()( int r, int c ) {
    if( !(r < rows()) || !(c < cols()) ) {
          std::cout << "Error: attempt to access "
          std::cout << "element out of bounds\n";
          return m_data[0][0];
    }
    // normal code
    return m_data[r][c];
}
```

«Exceptions»

- You can use a **try-catch** block to handle exceptions that are thrown

```
try {
    // code that might throw exception
}
catch ([Type of Exception] e) {
// what to do if exception is thrown
}
```

«Exceptions»

- You can use multiple catch blocks to catch exceptions

```
try {
    // code that might throw exception
}
catch ([Type of Exception 1] e) {
// what to do if exception 1 is thrown
}
catch ([Type of Exception 2] e) {
// what to do if exception 2 is thrown
}
catch ([Type of Exception 3] e) {
// what to do if exception 3 is thrown
}
```

«Exceptions»

- Example

```
template<class T>
class Matrix {
//…
T & operator()( int r, int c ) {
    try {
          if( !(r < rows()) || !(c < cols()) )
                throw "Access out of bounds";
          // normal code
          return m_data[r][c];
    }
    catch(const char * msg) {
          std::cout << msg << "\n";
          return m_dummyVal;
    }
}
//…
```

Throwing Exceptions

- Use the **throw** statement to throw an exception
  - `if(ptr.equals(null))`
    `throw "Null ptr exception";`
- The throw statement requires a single argument: a throwable object
- **Throwable** objects are

  primitive types

  instances of any subclass of **`std::exception`**

«Exceptions»

- Can use more complicated types
- Need to overload std::exception

```
class OutOfBoundsExc : public std::exception {
public:
    OutOfBoundsExc( int r, int c, int rows, int cols ) :
            m_r(r), m_c(c), m_rows(rows), m_cols(cols) {}
    void print() {
            std::cout << "Attempt to access element ("
                      << r << "," << c << ") in a matrix "
                      << "of size " << m_rows
                      << "x" << m_cols;
    }
private:
    int m_r;
    int m_c;
    int m_rows;
    int m_cols;
}
```

«Exceptions»

- Using the new type

```
T & operator()( int r, int c ) {
    try {
            if( !(r < rows()) || !(c < cols()) )
                    throw OutOfBoundsExc(r, c, rows(), cols());
            // normal code
            return m_data[r][c];
    }
    catch(OutOfBoundsExc & e) {
            e.print();
            return m_dummyVal;
    }
}
```

«Exceptions»

- Exceptions can be caught anywhere!

```
template<class T>
class Matrix {
//…
  T & operator()( int r, int c ) {
    if( !(r < rows()) || !(c < cols()) )
      throw OutOfBoundsExc(r,c,rows(),cols());
    // normal code
    return m_data[r][c];
  }
//…
}

int main() {
    Matrix<double> mat(5,5);
    double val;
    try { val = mat(3,5); }
    catch( OutOfBoundsExc & e ) { e.print(); }
    return 0;
}
```

Dummy val no longer needed!

Why handling exceptions

- Compilation cannot find all errors
- To separate error detection, reporting, and handling
  - Reporting/handling are separated from regular code
  - Increases code clarity
  - We defer how to worry about errors to somewhere else
- Group and differentiate error types
  - Write error handlers that handle very specific exceptions