Error Handling & Defensive Programming

Error Handling Concepts

- Murphy's Law
  - "Anything that can go wrong will go wrong"
- Error conditions will occur, and your code needs to deal with them
  - Out of memory, disk full, file missing, file corrupted, network error, ...
- Software should be tested to see how it performs under various error conditions
  - Simulate errors and see what happens
- Just because your program works on your computer doesn't mean that it will work everywhere else
- You'll be amazed at how many weird things will go wrong when your software is used out in the "wild"

Error Handling Concepts

- What should a program do when an error occurs?
- Some errors are "recoverable" - the program is able to recover and continue normal operation
- Many errors are "unrecoverable" - the program cannot continue and gracefully terminates
- Most errors are detected by low-level routines that are deeply nested in the call stack
- Low-level routines usually can't determine how the program should respond
- Information about the error must be passed up the call stack to higher-level routines that can determine the appropriate response

Propagating Error Information

- Return Codes
- Status Parameter
- Error State
- Exceptions

Return Codes

- A method uses its return value to tell the caller whether or not it succeeded
- In case of failure, the particular value returned can be used to determine the nature of the error

```
int MyClass::OpenFile(string fileName) {
…
}

MyClass obj;
int result = obj.OpenFile("index.html");
if (result < 0) {
    switch (result) {
        case -1:  … // file doesn't exist
        case -2:  … // file isn't writable
        case -3:  … // max number of files already open
    }
}
```

Return Codes

- Disadvantages of return codes
  - You have to use the return value to return error info even if you'd rather use it to return something else
  - Every time you call a method, you need to write code to check the return value for errors
    - All of the error-checking code obscures the main flow of the program
  - It's easy to write code that simply ignores errors because nothing forces you to check return values

Status Parameter

- A method has an additional parameter through which it returns status information
- In case of failure, the particular value returned through the parameter can be used to determine the nature of the error

```
void MyClass::OpenFile(string fileName, int * status) {
…
}

MyClass obj;
int result = 0;
obj.OpenFile("index.html", &result);
if (result < 0) {
   switch (result) {
      case -1:  … // file doesn't exist
      case -2:  … // file isn't writable
      case -3:  … // max number of files already open
   }
}
```

Status Parameter

- Disadvantages of status parameters
  - Every method call has an extra parameter (but you can use the return value for whatever you want)
  - Every time you call a method, you need to write code to check the status parameter's value for errors
    - All of the error-checking code obscures the main flow of the program
  - It's easy to write code that simply ignores errors because nothing forces you to check the status parameter

Error State

- Methods don't return error info
  - If something went wrong, you can't tell
- Objects store error info internally
- If you want to know if failures have occurred, you must query the object by calling a method

```
ifstream file;

file.open("index.html");

if (!file.is_open()) {
   // file could not be opened
}
```

Error State

- Disadvantages of error state
  - Every time you call a method, you need to write code to check the object's error state
    - All of the error-checking code obscures the main flow of the program
  - It's easy to write code that simply ignores errors because nothing forces you to check the error state

Exceptions

- Exceptions are an elegant mechanism for handling errors without the disadvantages of the other techniques
  - Return values aren’t tied up
  - No extra parameters
  - Error handling code isn't mixed in with the "normal" code
  - You can't ignore exceptions - if you don't handle them, your program will crash

Exceptions - throw

- The throw keyword is used to throw an exception

```
if (something went wrong) {
    throw MyException(a, b, c);
}
```

Exceptions - try, catch

```
void DoStuff() {
    A();
    B();
    C();
}
```

Exceptions - try, catch

```
void DoStuff() {
    try {
        A();
        B();
        C();
    }
    catch (ExceptionType_1 & e) {

        // handle exception type 1
    }
    catch (ExceptionType_2 & e) {

        // handle exception type 2
    }
    catch (ExceptionType_3 & e) {

        // handle exception type 3
    }
}
```

Exceptions - try, catch

```
void DoStuff() {
    try {
        A();
        B();
        C();
    }
    catch (ExceptionType_1 & e) {

        // handle exception type 1
    }
    catch (ExceptionType_2 & e) {

        // handle exception type 2
    }
    catch (ExceptionType_3 & e) {

        // handle exception type 3
    }
    catch (...) {

        // handle all other exceptions
    }
}
```

Exceptions - try, catch

```
#include <new>
using namespace std;

void DoStuff() {
   int * p = 0;
   try {
      p = new int[10000000000];

      … // use the array

      delete [] p;
   }
   catch (bad_alloc & e) {
      cout << "Insufficient memory" << endl;
   }
   catch (exception & e) {
      cout << "Error: " << e.what() << endl;
      delete [] p;
   }
}
```

Exceptions - try, catch

```
void DoSomething() {
    try {
        A();
    }
    catch (exception & e) {
        cout << "Error: " << e.what() << endl;
    }
}

void A() {
    try {
        B();
    }
    catch (bad_alloc & e) {
        // handle bad_alloc exception
    }
}

void B() {
    // some code that throws might throw exceptions
}
```

When an exception is thrown:

1. The program searches the enclosing try for an exception handler (or catch block) whose parameter matches the thrown object's type or one of its superclasses
2. The catch blocks are searched in the order they appear in the file, and the first matching one is used
3. If a matching exception handler is found, the thrown object is passed to the exception handler, and the handler is executed
4. If the code isn't in a try block, or no matching exception handler is found, the method aborts and the program searches the calling method for an appropriate exception handler
5. This process continues up the call stack until either an appropriate exception handler is found, or the search fails and the program terminates

finally – Java has it, C++ doesn't

- Finally block – code to be executed when the try block is exited, no matter what (i.e., if an exception occurred or not)

```
try {

}
catch (ExceptionType_1 e) {

}
catch (ExceptionType_2 e) {

}
catch (ExceptionType_3 e) {

}
finally {
}
```

In C++, use destructors to achieve finally-like functionality

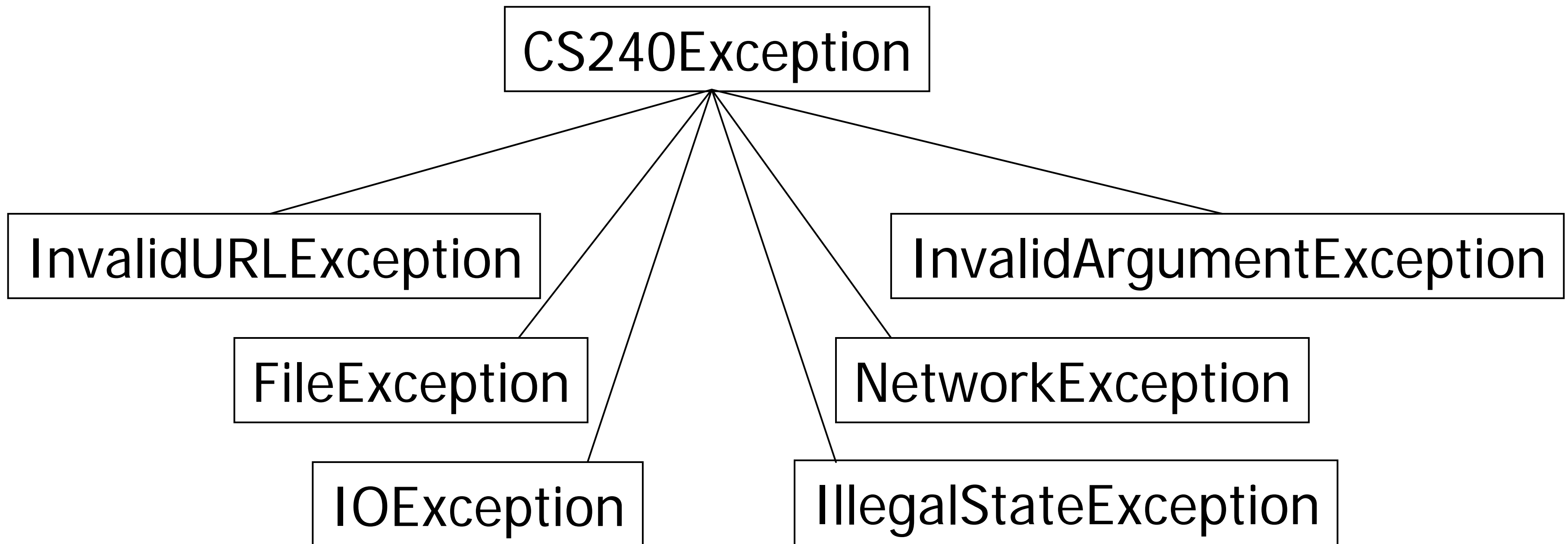
- When an exception is thrown, C++ guarantees that all objects residing on the stack will be destructed when they're popped off the stack

```
try {
   Object x; // object whose destructor contains
             // the "finally" code
}
catch (ExceptionType_1 & e) {

}
catch (ExceptionType_2 & e) {

}
catch (ExceptionType_3 & e) {

}
```

CS 240 Exception Classes

- The CS 240 Utilities provide several exception classes
- These exceptions are thrown by the Web Access classes when errors occur, and must be handled by your code
- You may also throw them from your own methods

CS 240 Exception Classes



CS240Exception

```
class CS240Exception {
 protected:
  std::string message;

 public:
  CS240Exception() {
    message = "Unknown Error";
  }
  CS240Exception(const string & msg) {
    message = msg;
  }
  CS240Exception(const CS240Exception & e) {
    message = e.message;
  }
  ~CS240Exception() {
    return;
  }
  const string & GetMessage() {
    return message;
  }
};
```

Handling CS240Exception's

```
#include <iostream>
#include "URLConnection.h"
#include "CS240Exception.h"
using namespace std;

void main() {
   InputStream * is = 0;
   try {
      is = URLConnection::Open("http://www.cs.byu.edu/index.html");
      while (!is->IsDone()) {
         char c = is->Read();
         cout << c;
      }
      delete is;
   }
   catch (CS240Exception & e) {
      cout << "Error: " << e.GetMessage() << endl;
      delete is;
   }
   catch (...) {
      cout << "Unknown error occurred" << endl;
      delete is;
   }
}
```

Defensive Programming

- Good programming practices that protect you from your own programming mistakes, as well as those of others
  - Assertions
  - Parameter Checking

Assertions

- As we write code, we make many assumptions about the state of the program and the data it processes
  - A variable's value is in a particular range
  - A file exists, is writable, is open, etc.
  - The maximum size of the data is N (e.g., 1000)
  - The data is sorted
  - A network connection to another machine was successfully opened
  - ...
- The correctness of our program depends on the validity of our assumptions
- Faulty assumptions result in buggy, unreliable code

Assertions

```
int BinarySearch(int data[], int dataSize, int searchValue) {

    // What assumptions are we making about the parameter values?

    …
}
```

- data != 0
- dataSize >= 0
- data is sorted
- What happens if these assumptions are wrong?

Assertions

- Assertions give us a way to make our assumptions explicit in our code
- `#include <assert.h>`
- `assert(temperature > 32 && temperature < 212);`
- The parameter to `assert` is any boolean expression
- If the expression is false, assert prints an error message and aborts the program
- Assertions are usually disabled in released software
- Assertions are little test cases sprinkled throughout your code that alert you when one of your assumptions is wrong
- This is a powerful tool for avoiding and finding bugs

Assertions

```
int BinarySearch(int data[], int dataSize, int searchValue) {
   assert(data != 0);
   assert(dataSize > 0);
   assert(IsSorted(data, dataSize));

   …
}

string * SomeFunc(int y, int z) {
   assert(z != 0);
   int x = y / z;
   assert(x > 0 && x < 1024);
   return new string[x];
}
```

Exceptions vs. Assertions

- If one of my assumptions is wrong, shouldn't I throw an exception rather than use an assertion?
- Assertions are used to find and remove bugs before software is shipped
  - Assertions are turned off in the released software
- Exceptions are used to deal with errors that can occur even if the code is completely correct
  - Out of memory, disk full, file missing, file corrupted, network error, …

Parameter Checking

- Another important defensive programming technique is "parameter checking"
- A method or function should always check its input parameters to ensure that they are valid
- Two ways to check parameter values
  - assert
  - if statement that throws exception if parameter is invalid
- Which should you use, asserts or exceptions?

Parameter Checking

- Another important defensive programming technique is "parameter checking"
- A method or function should always check its input parameters to ensure that they are valid
- Two ways to check parameter values
  - assert
  - if statement that throws exception if parameter is invalid
- Which should you use, asserts or exceptions?
- If you have control over the calling code, use asserts
  - If parameter is invalid, you can fix the calling code
- If you don't have control over the calling code, throw exceptions
  - e.g., your product might be a class library

Parameter Checking

```
int BinarySearch(int data[], int dataSize, int searchValue) {
   assert(data != 0);
   assert(dataSize > 0);
   assert(IsSorted(data, dataSize));

   …
}


int BinarySearch(int data[], int dataSize, int searchValue) {
   if (data == 0 || dataSize <= 0 || !IsSorted(data, dataSize)) {
      throw InvalidArgumentException();
   }

   …
}
```