Error Handling in C++

- Exceptions, once *thrown*, must be *caught* by specialized error-handling code.
  - If an exception goes uncaught, the program will crash to the desktop.
Error Handling

• Many of C++’s built-in classes (especially those in the std namespace) are designed to throw exceptions when errors occur.

• Examples:
  – exception
  – runtime_error
  – range_error
Throwing Exceptions

- For a class to signal an error, it can create an instance of the appropriate exception type and then throw it.
- Exceptions often may take on a message, in string form, which can be useful for debugging purposes.
A First Object

Person::Person(string name, int age)
{
    if (name.compare("") == 0)
    {
        throw invalid_argument("Name may not be null!");
    }
...
Person::Person(string name, int age)
{
...
    if(age < 0)
    {
        throw out_of_range
            ("Age must be non-negative!");
    }
...
...
A First Object

• Note that whenever the constructor throws an exception that isn’t handled internally, it cancels the construction process.
  – Constructor problem solved!
The general structure for handling an error that has occurred:

```java
try {
    throw exception();
} catch (exception e){
    //Fix the error!
}
```
Error Handling

```
try {
    throw exception();
}

catch (exception e) {
    //Fix the error!
}
```

“try” this code and see if any errors happen
Error Handling

```java
try {
    throw exception();
} catch (exception e) {
    //Fix the error!
}
```

If they do, skip the rest of this code and “catch” the error here…
Error Handling

```java
try {
    throw exception();
}

catch (exception e) {
    //Fix the error!
}
```

If they do, **skip the rest of this code** and “catch” the error here...
try{
    throw exception();
}
catch (exception e)
{
    //Fix the error!
}
Error Types

- There are actually multiple sorts of errors which can occur within a program and its code.

  - *Compile-time* errors: the interpreter / compiler can’t make sense of your code.

  - *Logical* errors: the program doesn’t crash, but it behaves differently than intended.
Error Types

- There are actually multiple sorts of errors which can occur within a program and its code.
  - *Run-time* errors: Errors which crash a program, but were not intentionally generated by programmer code.
  - Generally, user-generated issues caused by bad inputs. (GIGO, PEBKAC)
  - When a calculator program is told to divide by zero, if it doesn’t check for illegalness, a runtime error will occur.
Error Types

• There are actually multiple sorts of errors which can occur within a program and its code.
  
  – *Generated* errors: the program detects that it is malfunctioning and generates an error to signal it.
  
  – Often generated to prevent run-time errors from crashing the program. Making these gives a chance for recovery if they are caught elsewhere.
Error Types

• When a program hangs (goes unresponsive), it’s typically a logical error.
These license terms are an agreement between Sysinternals (Sysinternals) and you. By using the SYSGEAR software, you agree to the terms of this agreement. If you do not agree to these terms, do not use the software.

BY USING THE SOFTWARE, YOU AGREE TO THE FOLLOWING:

1. INSTALL THE SOFTWARE ON A SINGLE COMPUTER.
2. SCOPE OF LICENSE.

The software is provided 'AS IS' without warranty of any kind. The entire risk as to the quality and performance of the software is with you. You may not: (i) reproduce or distribute the software; (ii) reverse engineer, decompile or disassemble the software; (iii) modify or create derivative works based upon the software; (iv) publicly display or perform the software; (v) resell the software; or (vi) use the software for any purpose other than personal or non-commercial use.

If you close the program, you might lose information.

- Close the program
- Wait for the program to respond
Recovery

• Unfortunately, like in the prior example, not all errors can be detected.
  – Sometimes, an application can get stuck in an infinite loop, rendering it completely unresponsive.
  – Multithreaded applications can also become stuck due to “deadlock” and “livelock” situations.
On the Use of Exceptions

• Exceptions are extremely handy to have as a tool for an object to indicate bad inputs to its constructor or method.

• However, exceptions are quite “expensive”, computationally, to throw.
  – Remember, they interrupt everything.
On the Use of Exceptions

Objects should throw exceptions when:

– A constructor receives (bad) inputs that would result in an invalid object.
– A method receives bad input
– Out of range index or value
– Null pointer
On the Use of Exceptions

• Objects should throw exceptions when:
  – A method cannot perform the requested action.
  – Some objects may have different “modes,” where certain actions may only be possible in certain situations.
  – Example: a file must be opened to read or write from/to it.
On the Use of Exceptions

- Objects should throw their own exceptions, rather than relying on a future method to throw exceptions.
  - When debugging, it is better to know the underlying source of the erroneous error.
  - Thus, the sooner code can detect that an error will occur (even if later in the chain), the better.
On the Use of Exceptions

• Objects should throw their own exceptions, rather than relying on a future method to throw exceptions.
  – Failure to do so will make it seem as if the object is miscoded, using the future method incorrectly.
On the Use of Exceptions

- Exceptions are a useful tool for object-orientation, allowing objects to actively prevent actions that would be invalid.
On the Use of Exceptions

- Exceptions are a useful tool for object-orientation, allowing objects to actively prevent actions that would be invalid.
  - They also allow objects to report why those actions are invalid, which aids debugging.
  - Sometimes, it may even be possible to recover from the error, depending on its type.