Exception handling

- Sometimes there are complex actions that need to take place under unusual (exceptional) circumstances
- Developer may prefer not to have these actions “clutter” the normal code sequence
- Common model for handling these exceptional events: try, throw, catch
  - declare a code block in which we’ll perform exception handling
  - insert code to detect exceptional condition
  - throw an exception with supporting info: this exits the code block
  - an external ‘catch’ captures and processes the exception
catch/throw in lisp

- limited exception handling through throw and catch
- define a catch block, can throw exceptions from inside
- throw exits a specified block with a chosen return value
  
  (setf myBlockResult (catch 'myblock
     ....do regular stuff ...
     (if (somecondition) (throw 'myblock value))
     ... do more regular stuff ... )))

- myBlockResult now holds either the normal result of the block or the value that was thrown
catch/throw in C++

- try block encloses code in which we perform checking for exceptions, can have nested try blocks
- throw/catch look somewhat like a function call/definition (except no return values)
- catch parameter list identifies what “kinds” of exceptions that catch can handle
- can have multiple catches in a row, uses first one with compatible parameter list to the data thrown
- catches can re-throw for processing by later catch
// we'll use a class to represent our exception data
class exc {

  protected:
    std::string emsg; // msg to describe what went wrong

  public:
    exc(string m) { emsg = m; }
    void print() { std::cout << "Exception! " << emsg; }
};
int main() {
    try {
        ...
        // if something goes wrong construct and throw an exception
        if (thing1 goes wrong) throw exc("thing 1!");
        ...
        if (thing2 goes wrong) throw exc("something else!");
        ...
    }
    catch (exc e) { // runs if an exception is thrown
        e.print(); // prints message about the exception
    }
}
Guidelines for exception use

- The throw/catch exits the normal processing flow, can make reading/understanding code flow more complex
- Thus not intended for general/normal error handling
- Accepted use is for problems well outside regular handling, e.g. you detect your data structure has become corrupted
- Catch blocks go from most specific to most general (try to handle most detailed cases first, if that doesn’t fit then go on to try more and more general handlers)
Exception class heirarchy

• Common approach is to use a class heirarchy of exceptions, and to only throw exceptions from that heirarchy

• Base class is the generic “something went wrong” exception, and is caught at the main routine

• Derived classes focus on more specific subtypes of exception

• Catch blocks get listed from most specific to least specific, guaranteeing the most appropriate handler gets invoked (and if it can’t handle it then can always re-throw)