Chapter 6
Exception Handling and Debugging

Exception Handling

- Exceptions are a language facility for managing errors; not all languages support exceptions.
- Exceptions help to distinguish the normal flow of execution from *exceptional cases.*
- Exceptions are a language facility for managing errors.
- When your code encounters a problem that it can’t handle, it stops dead and throws up an *exception—an object representing the error.*

Operational Models of Exceptions

- **The termination model**
- Execution continues after the handler that caught the exception. This behavior is provided by C++, .NET, and Java.
- **The resumption model**

  Execution resumes where the exception was raised.

Exception Handling

- **If the Language supports Exceptions**
  - **use the full power of the language**

    e.g. in Java, making all exceptions “checked” leads to safer programs
  - **but you’ll still need to document them**
  - e.g. Java doesn’t force you to say why a method might throw an exception
- **Otherwise:**

  1) declare an enumerated type for exception names

  enum str_exceptions {okay, null_pointer, empty_string, not_null_terminated};

  2) have the procedure return an extra return value

  either: str_exceptions palindrome(char *s, boolean *result);

  or: boolean palindrome(char *s, str_exceptions *e);

  (be consistent about which pattern you use)

  3) test for exceptions each time you call the procedure

  e.g. if (palindrome(my_string, &result)==okay) { … }

  else /*handle exception*/

  4) write exception handlers

  procedures that can be called to patch things up when an error occurs.

Writing Exception Handlers

- **The calling procedure is responsible for:**
  - checking that an exception did not occur
  - handling it if it did
- **Could handle the exception by:**
  - reflecting it up to the next level
    - i.e. the caller also throws an exception (up to the next level of the program)
    - Can throw the same exception (automatic propagation)…
    - …or a different exception (more context info available!)
  - masking it
    - i.e. the caller fixes the problem and carries on (or repeats the procedure call)
  - halt the program immediately
    - equivalent to passing it all the way up to the top level

Debugging

- Debugging
  - Finding out why a program is not functioning as intended
- Testing ≠ debugging
  - test: reveals existence of problem; test suite can also increase overall confidence
  - debug: pinpoint location + cause of problem

A Bug's Life

- *defect* – mistake committed by a human as seen as a problem in the code
- *failure* – visible error: program violates its specification
- *root cause* – core issue that led to the defect
- [One set of definitions – there are others]

- Debugging starts when a failure is observed
  - During any phase of testing or in the field

Last resort: debugging

- Defects happen – people are imperfect
  - Industry average: 10 defects per 1000 lines of code ("kloc")
- Defects that are not immediately localizable happen
  - Found during integration testing
  - Or reported by user
- The cost of finding and fixing an error usually goes up by an order of magnitude for each lifecycle phase it passes through

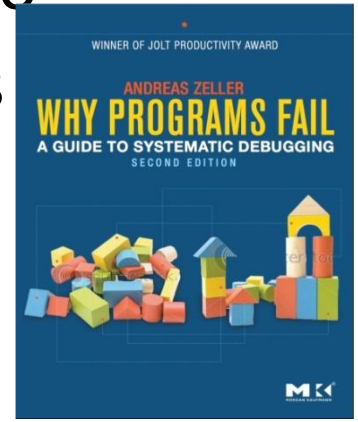
- step 1 – Clarify symptom (simplify input), create test
- step 2 – Find and understand cause, create better test
- step 3 – Fix
- step 4 – Rerun all tests

The debugging process

- step 1 – find a small, repeatable test case that produces the failure (may take effort, but helps clarify the defect, and also gives you something for regression)
  - Don't move on to next step until you have a repeatable test
- step 2 – narrow down location and proximate cause
  - Study the data / hypothesize / experiment / repeat
  - May change the code to get more information
  - Don't move on to next step until you understand the cause
- step 3 – fix the defect
  - Is it a simple typo, or design flaw? Does it occur elsewhere?
- step 4 – add test case to regression suite
  - Is this failure fixed? Are any other new failures introduced?

Debugging and the scientific method

- Debugging should be systematic
  - Carefully decide what to do – flailing can be an instance of an epic fail
  - Keep a record of everything that you do
  - Don’t get sucked into fruitless avenues
- Formulate a hypothesis
- Design an experiment
- Perform the experiment
- Adjust your hypothesis and continue



Reducing relative input size

- Sometimes it is helpful to find two almost identical test cases where one gives the correct answer and the other does not
  - Can't find "very happy" within
    - `"I am very very happy to see you all."`
  - Can find "very happy" within
    - `"I am very happy to see you all."`

General strategy: simplify

- In general: find simplest input that will provoke failure
  - Usually not the input that revealed existence of the defect
- Start with data that revealed defect
  - Keep paring it down (binary search "by you" can help)
  - Often leads directly to an understanding of the cause
- When not dealing with simple method calls
  - The "test input" is the set of steps that reliably trigger the failure
  - Same basic idea