

Debugging

CPSC 315 – Programming Studio

Bugs

- Term has been around a long time
 - Edison
 - Mark I – moth in machine
- Mistake made by programmers
- Also (and maybe better) called:
 - Errors
 - Defects
 - Faults

Sources of Bugs

- Bad Design
 - Wrong/incorrect solution to problem
 - From system-level to statement-level
- Insufficient Isolation
 - Changes in one area affect another
- Typos
 - Entered wrong text, chose wrong variable
- Later changes/fixes that aren't complete
 - A change in one area affects another

Debugging in Software Engineering

- Programmer speed has high correlation to debugging speed
 - Best debuggers can go up to 10 times faster
- Faster in finding bugs
- Find more bugs
- Introduce fewer new bugs

Ways **NOT** to Debug

- Guess at what's causing it
- Don't try to understand what's causing it
- Fix the symptom instead of the cause
 - Special case code
- Blame it on someone else's code
 - Only after extensive testing/proof
- Blame it on the compiler/computer
 - Yes, it happens, but almost never is this the real cause

An Approach to Debugging

Stabilize the error

Locate the source

Fix the defect

Test the fix

Look for similar errors

- Goal: Figure out *why* it occurs and fix it completely

1. Stabilize the Error

- Find a simple test case to reliably produce the error
 - Narrow it to as *simple* a case as possible
- Some errors resist this
 - Failure to initialize
 - Pointer problems
 - Timing issues

1. Stabilizing the Error

- Converge on the actual (limited) error
 - Bad: “It crashes when I enter data”
 - Better: “It crashes when I enter data in non-sorted order”
 - Best: “It crashes when I enter something that needs to be first in sorted order”
- Create hypothesis for cause
 - Then test hypothesis to see if it's accurate

2. Locate the Source

- This is where good code design helps
- Again, hypothesize where things are going wrong in code itself
 - Then, test to see if there are errors coming in there
 - Simple test cases make it easier to check

When it's Tough to Find Source

- Create multiple test cases that cause same error
 - But, from different “directions”
- Refine existing test cases to simpler ones
- Try to find source that encompasses all errors
 - Could be multiple ones, but less likely
- Brainstorm for sources, and keep list to check
- Talk to others
- Take a break

Finding Error Locations

- Process of elimination
 - Identify cases that work/failed hypotheses
 - Narrow the regions of code you need to check
 - Use unit tests to verify smaller sections
- Process of expansion:
 - Be suspicious of:
 - areas that previously had errors
 - code that changed recently
 - Expand from suspicious areas of code

Alternative to Finding Specific Source

- Brute Force Debugging
 - “Guaranteed” to find bug
 - Examples:
 - Rewrite code from scratch
 - Automated test suite
 - Full design/code review
 - Fully output step-by-step status
- Don't spend more time trying to do a “quick” debug than it would take to brute-force it.

3. Fix the Defect

- Make sure you understand the *problem*
 - Don't fix only the symptom
- Understand what's happening in the program, not just the place the error occurred
 - Understand interactions and dependencies
- Save the original code
 - Be able to “back out” of change

Fixing the Code

- Change only code that you have a good reason to change
 - Don't just try things till they work
- Make one change at a time

4. Check Your Fix

- After making the change, check that it works on test cases that caused errors
- Then, make sure it still works on other cases
 - Regression test
 - Add the error case to the test suite

5. Look for Similar Errors

- There's a good chance similar errors occurred in other parts of program
- *Before* moving on, think about rest of program
 - Similar routines, functions, copied code
 - Fix those areas immediately

Preventing Bugs Or Finding Difficult Ones

- Good Design
- Self-Checking code
- Output options
 - Print statements can be your friend...

Debugging Tools

- Debuggers
 - Often integrated
 - Can examine state in great detail
- Don't use debuggers to do "blind probing"
 - Can be far less productive than thinking harder and adding output statements
 - Use as "last resort" to identify sources, if you can't understand another way

Non-traditional Debugging Tools

- Source code comparators (diff)
- Compiler *warning* messages
- Extended syntax/logic checkers
- Profilers
- Test frameworks