CSSE2310/7231 — 5.1

Debugging + Ass2

Debugging (Challenges)

Notes

1. Programming is not a single skill.
   - Writing code
   - Debugging code
   - Documenting code
   - ...

   these skills do not necessarily develop at the same rate.
2. Most programmers will have to deal with this
   - This is not pejorative
3. Is not a list of recipes

Where people start — "Stare and hope"

My program doesn't work:

- Read code
- Think about the code
- If a solution hasn't occurred to you, repeat

This leads to four challenges.

#1 Code size vs Memory size

- Your programs will likely be growing in size.
- Even "small" programs may be too big to fit in your mind at once.

#1 Code size vs Memory size

Do the following before trying to read in detail:

1. Reproduce the problem.
   - "My code segfaults sometimes."
   - "My code segaults when I enter xyz."
   - The second one is in a much better position.
2. Narrow the search area.
   - eg segfault — Add extra (flushed) output.
   - ABCCCCCDEEEF (but no G) suggests that the crash happens between F and G.

#2 Symptom vs Causes

- Bigger and more complex code means that a symptom and its cause may be further apart.
- It can be tempting to "treat the symptom" rather than the cause.

#2 Symptom vs Causes

Symptom is a segfault, we've narrowed the problem to Line X.
Printing variables shows `p` is `null` pointer.

- Is Line X the problem?
- Is X (and the functions which call it) supposed to work with `nulls`?
  - Yes — X needs to be fixed
  - No — we have a new symptom to chase.

#2 Symptom vs Causes

. . . Symptom: variable `P` is 0 on Line X. Is it 0 because:

- P was never initialised properly?
- P initialised by a misbehaving function?
- P was initialised but was incorrectly changed in between?

Follow the next symptom.

#3 Bugs you can't see

Ever had the following happen: you call someone over to help...

- after 10 seconds they point out something that should have been obvious.
- Midway through explanation. "Actually I don't need you I've worked it out".
  - "Rubber duck debugging"

You might be seeing what you expect rather than what is actually there.

If possible take a break.

#3 Bugs you can't see — untested assumptions

- A function doesn't work the way you think (eg corner cases).
- Looking at the wrong part of the code.
- A variable has the value you expect[1]

Using your intuition is fine but before you rely too heavily on something, test it.

[1] Especially in multi-threaded code

#4 Inefficient but easy

- "Stare and hope" is a trap because while it is inefficient, it is low effort per time.
- Programmers[2] get tired/stressed and it can take non-trivial self-disipline not to default to "stare and hope".

[2] like all people

Ass2

Ass2 — why debuggers

- symbolic debuggers are useful tools
  - Don't always have access to one though.
- Can deal with simple questions like "where did the code segfault?".
- In general only as good as the questions you know how to ask.
- Over-reliance on them could lead to narrow thinking
  - "Where does symptom occur?" is not the same as "What is the cause?"
  - Torvalds on kernel debuggers

Ass2 — the Binary Bomb

1. Each student has a "bomb" program.
2. Bomb has a number of phases.
3. To solve a phase, give the correct input.
   - The fewer incorrect guesses you make on a phase, the higher your mark for that phase.
   - You can't ever lose marks by attempting the bomb.
   - How? Run it inside GDB.
4. In a new directory: `getbomb`

The bomb will only run properly on moss.

Ass2 — Practice

- bomb
  - ./bomb demo1
  - ...
  - ./bomb demo4

See the man pages for `getbomb` and `bomb`

- Working out how to solve the puzzles (and how to get the tools to do what you want) is the point of the assignment. Do not discuss this with other students.
  This is stricter than other assignments.
- There is no submission process for this assignment. All attempts will be automatically recorded.
- No we won't be removing attempts from the records.
  - Even if you really didn't mean to press Y
  - or if you managed to use gdb to jump past the 'Y' check.