School of Computing
& Communications

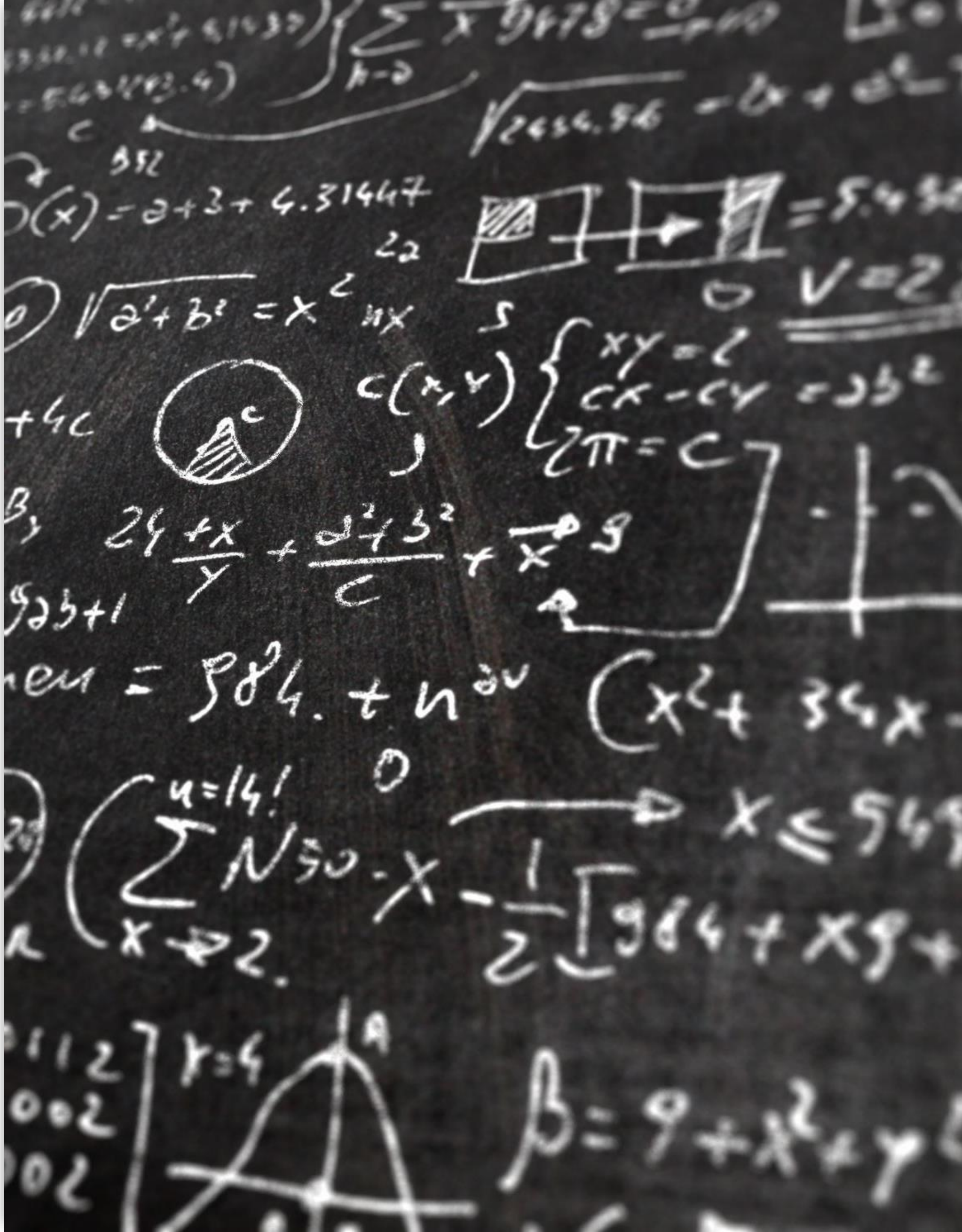
Lancaster
University

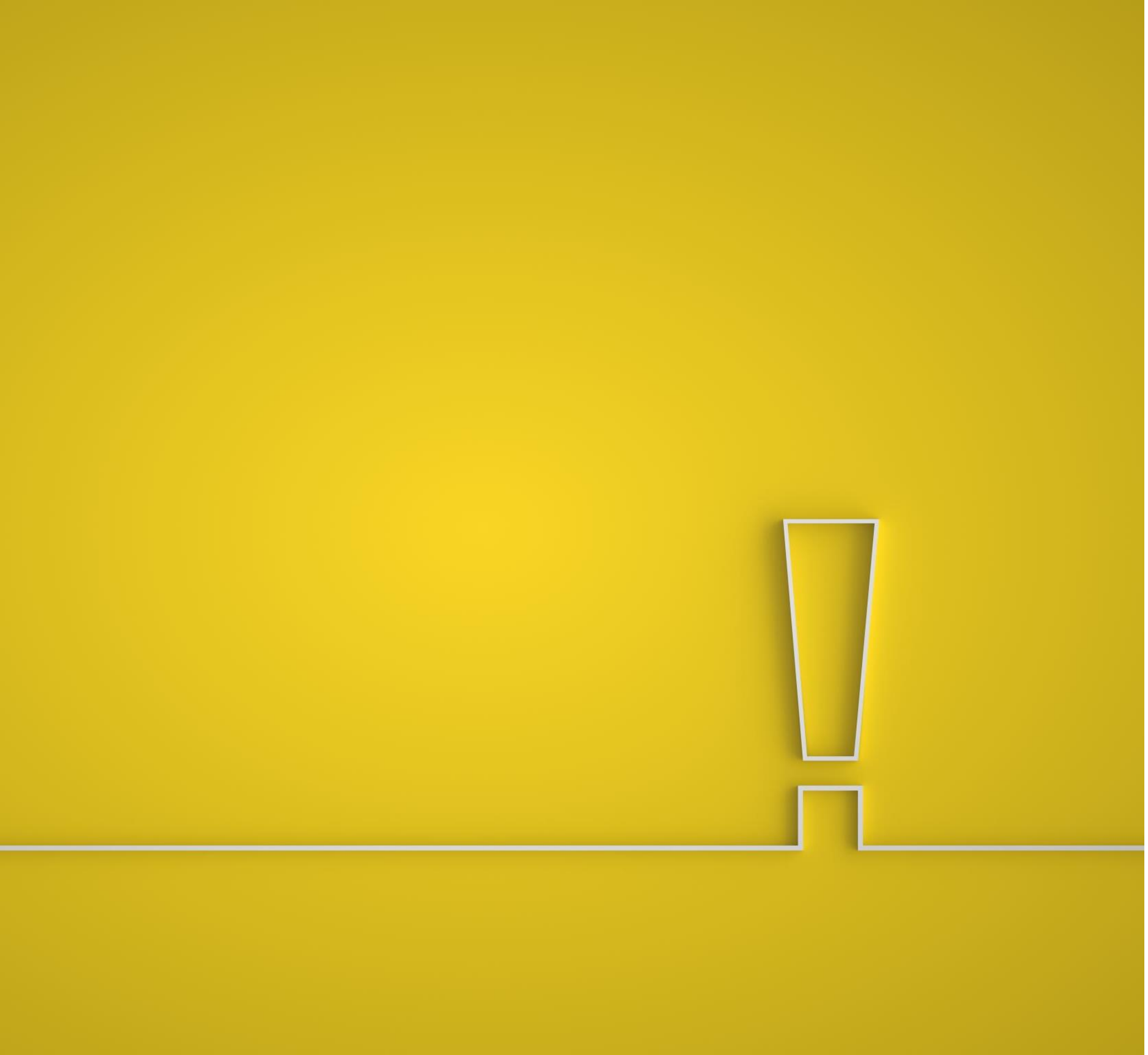
SCC.111 Software Development –Lecture 9: Debugging II

Adrian Friday, Nigel Davies, Hansi Hettiarachchi, Saad Ezzini

This lecture

- How to find logical and programming errors (debug) your code
- **Two further strategies** for isolating problems (we did ‘dry running’ last week)
- Worked examples





run time errors are bad - *you need to fix them* :)

Debugging *is* detective work

It’s frustrating to have a problem with your code you can’t find. We need to isolate it.

Approach 2: Testing hypotheses to isolate faults

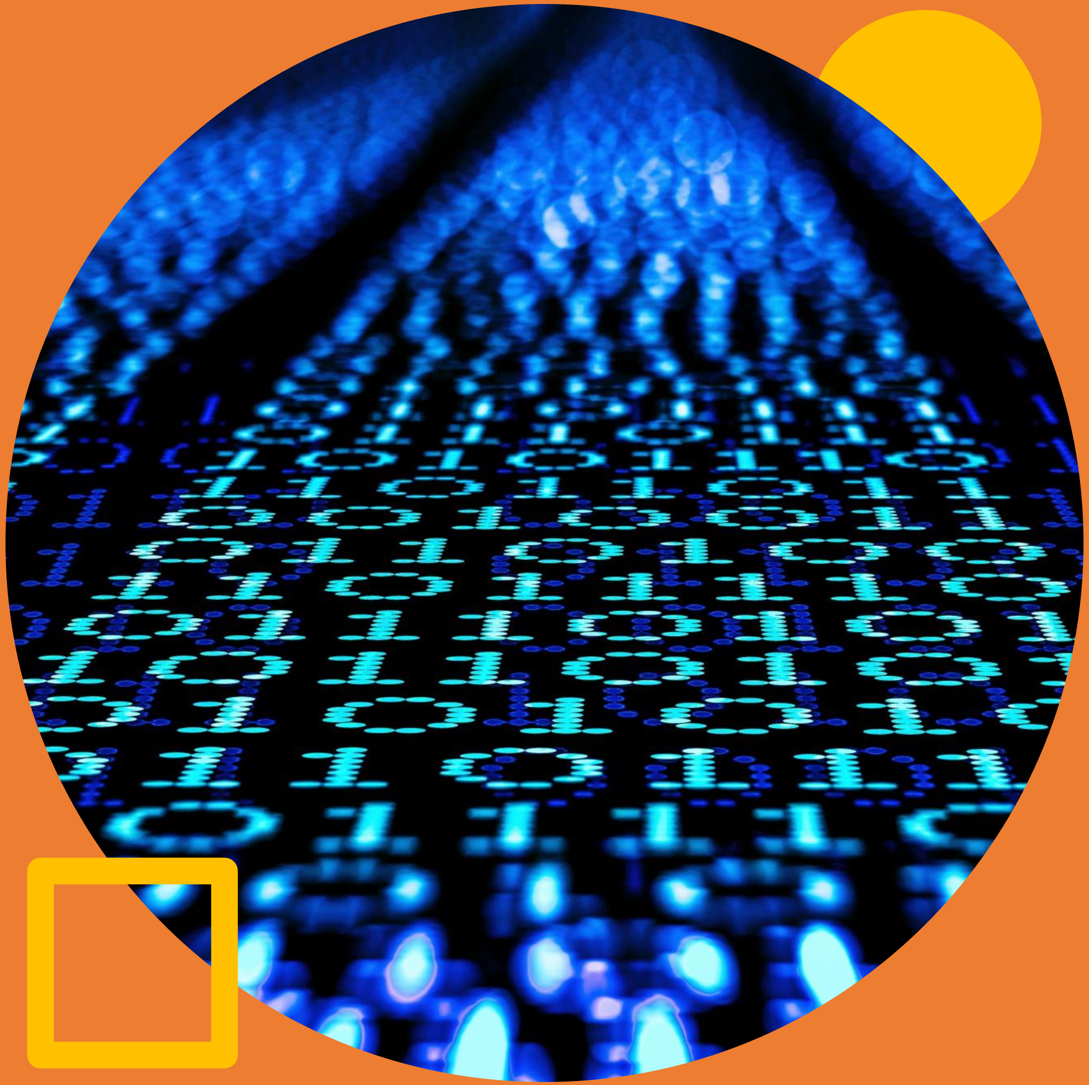
- a variable *should have* a certain value at some point in your source file
- the loop *should* exit, but doesn't
- in a given *if-then-else* statement, the *else* is the one that is executed
- that when you call a certain function, the function receives the *correct parameters*, and returns the *correct* *<u>result</u>*

Testing hypotheses using printf

Walkthrough: Testing hypotheses using printf

It’s almost quantum. Note that the act of modifying your code might mask subtle problems (e.g. memory corruption or ‘splat bugs’, timing or concurrency issues)

As code gets more complex or problems are harder to isolate, *what if we could execute, stop and restart the live code?*



Approach 3: Runtime debugging

Typically for finding more mysterious and hard to isolate faults

Typical runtime debugger functionality & terminology

01

Set ‘breakpoints’ at specific lines or on specific conditions to ‘interrupt execution’

02

Single step ‘into’ or ‘over’ functions

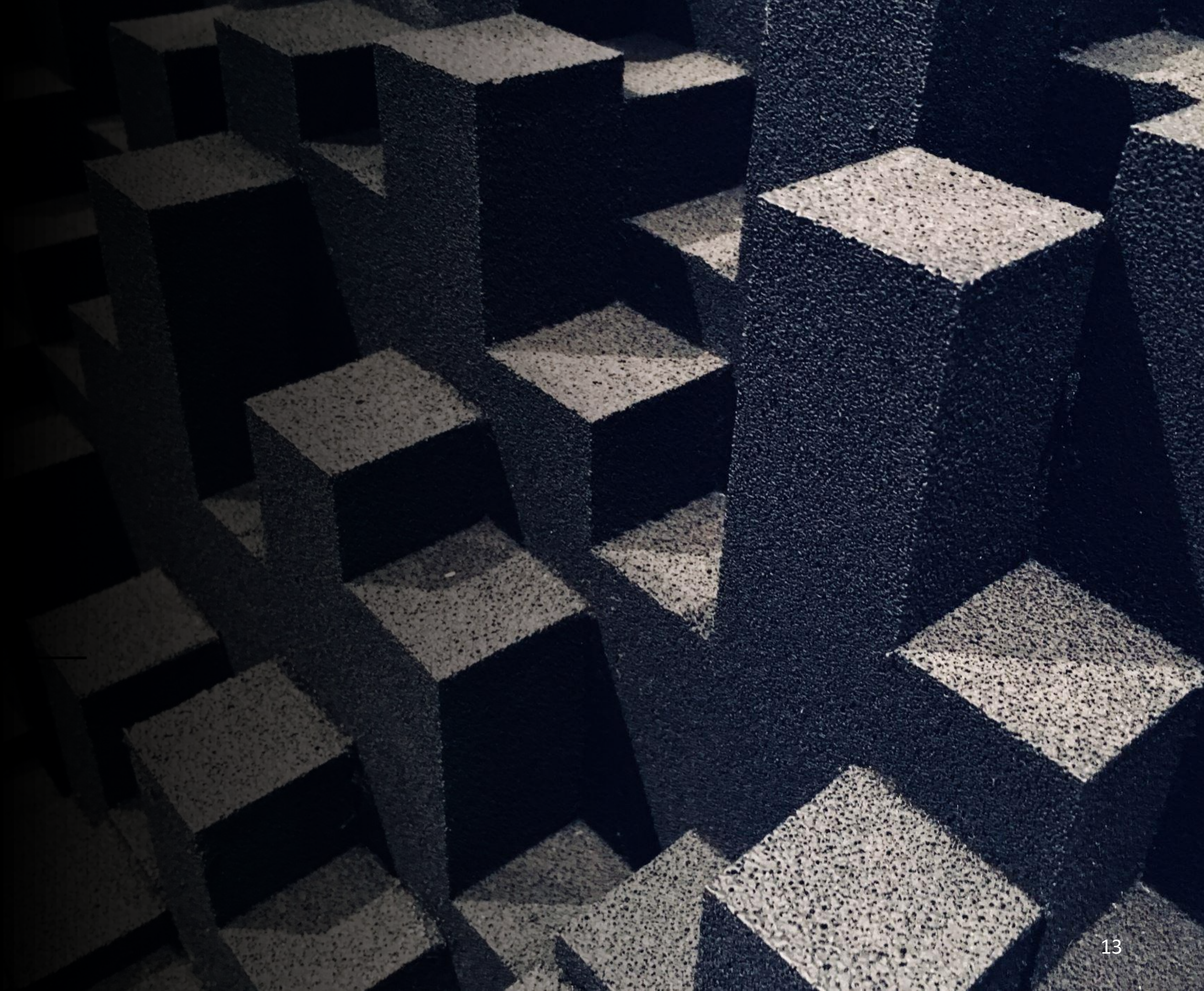
03

Inspect and sometimes change variable values

04

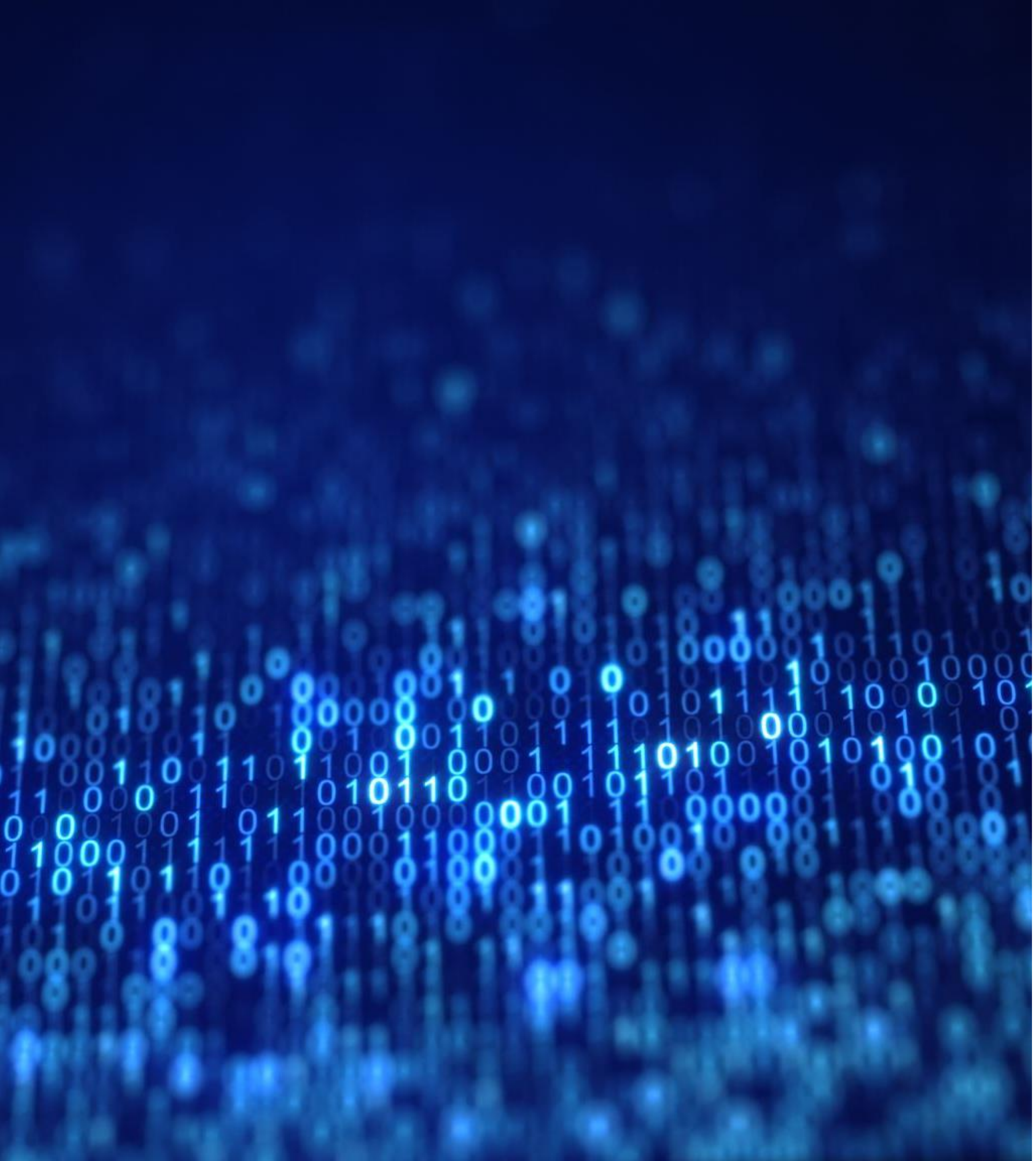
Set ‘watchpoints’ to look for changes to the state (variable values)

Walkthrough: Fault finding using the runtime debugger



When to use which approach

- Normally dry running is enough to ‘see’ the problem for manageable size code units
  - This should just become *a habit* whenever you read code
  - *another good reason for relatively specific and modest sized functions!*
- Using ‘printf’ to get your code to ‘speak to you’
  - So called ‘debug printfs’ are essential for fault finding in the small, and error logs are common for tracking behaviour of production systems
- Debuggers are useful for finding confusing or unexpected runtime errors and post ‘crash dump’ analysis
  - E.g. memory errors (‘splat bugs’), concurrency, data dependent faults



Summary

- You should know what *debugging* is
- How to formulate & test hypotheses about how your code executes
- How to use debugging statements (e.g. printf) to isolate problems
- A brief intro to software debuggers, typically for hard to find errors or code forensics