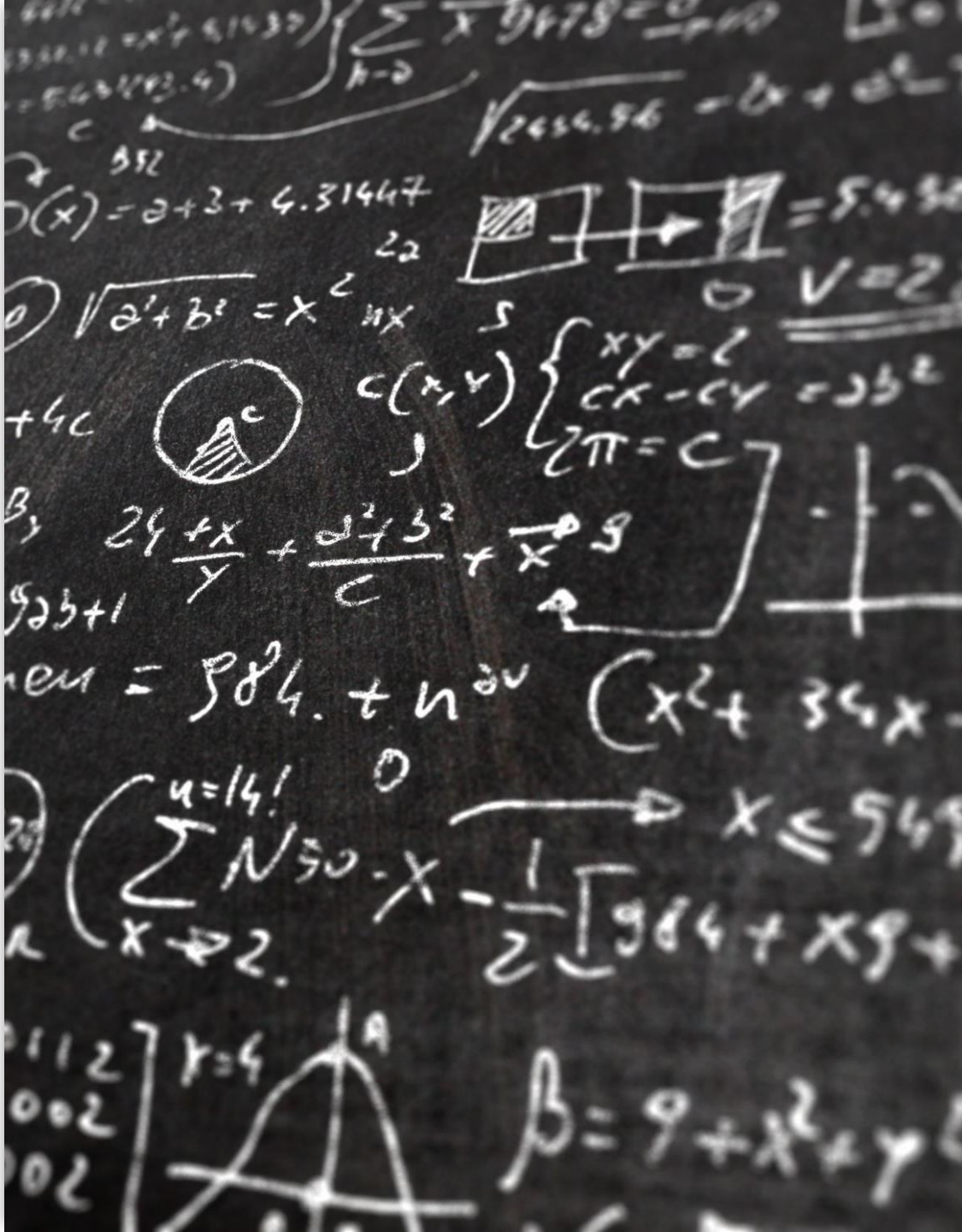
SCC.111 Software Development – Lecture 8: Debugging

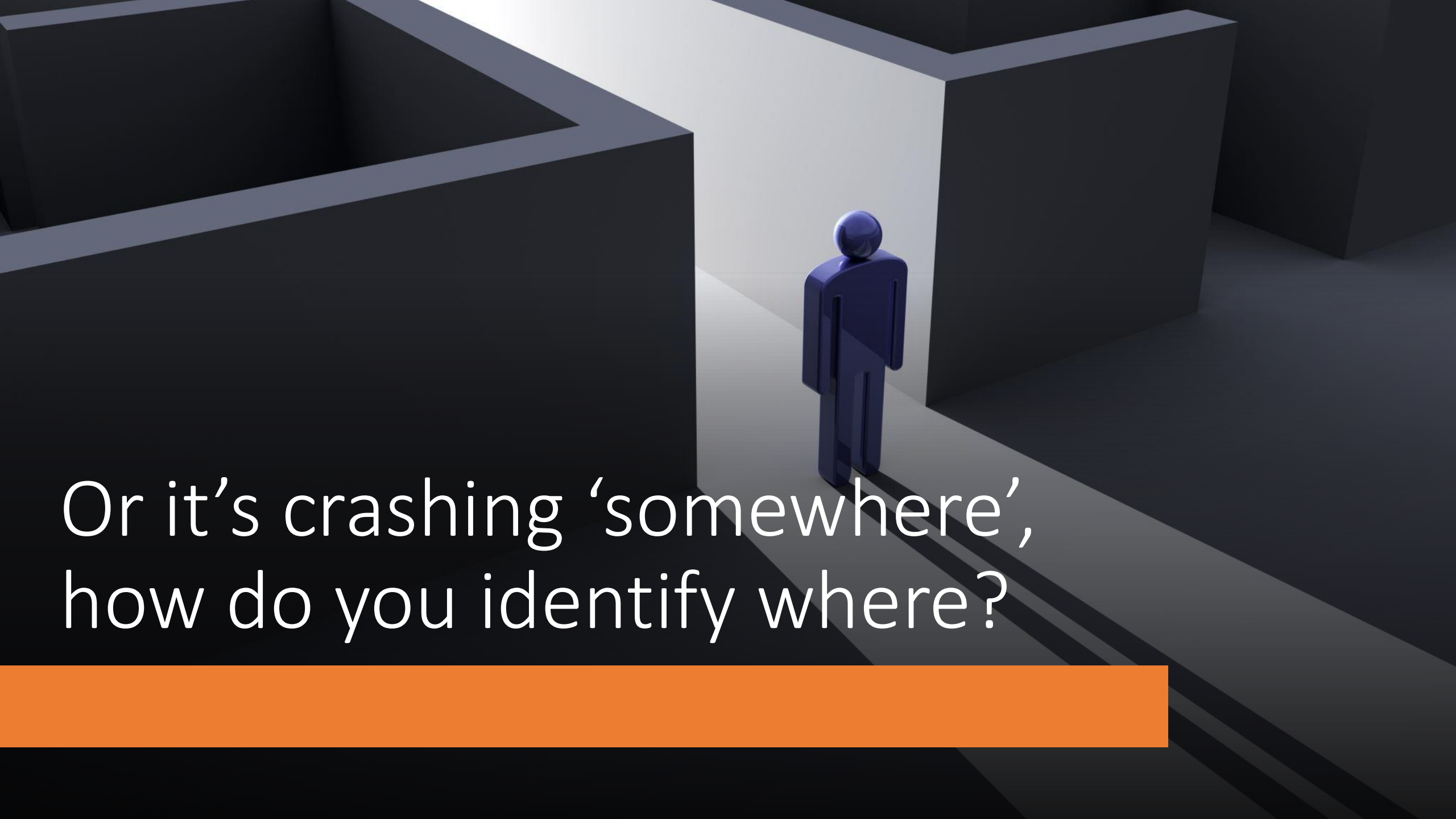
Adrian Friday, Nigel Davies, Hansi Hettiarachchi, Saad Ezzini

This lecture

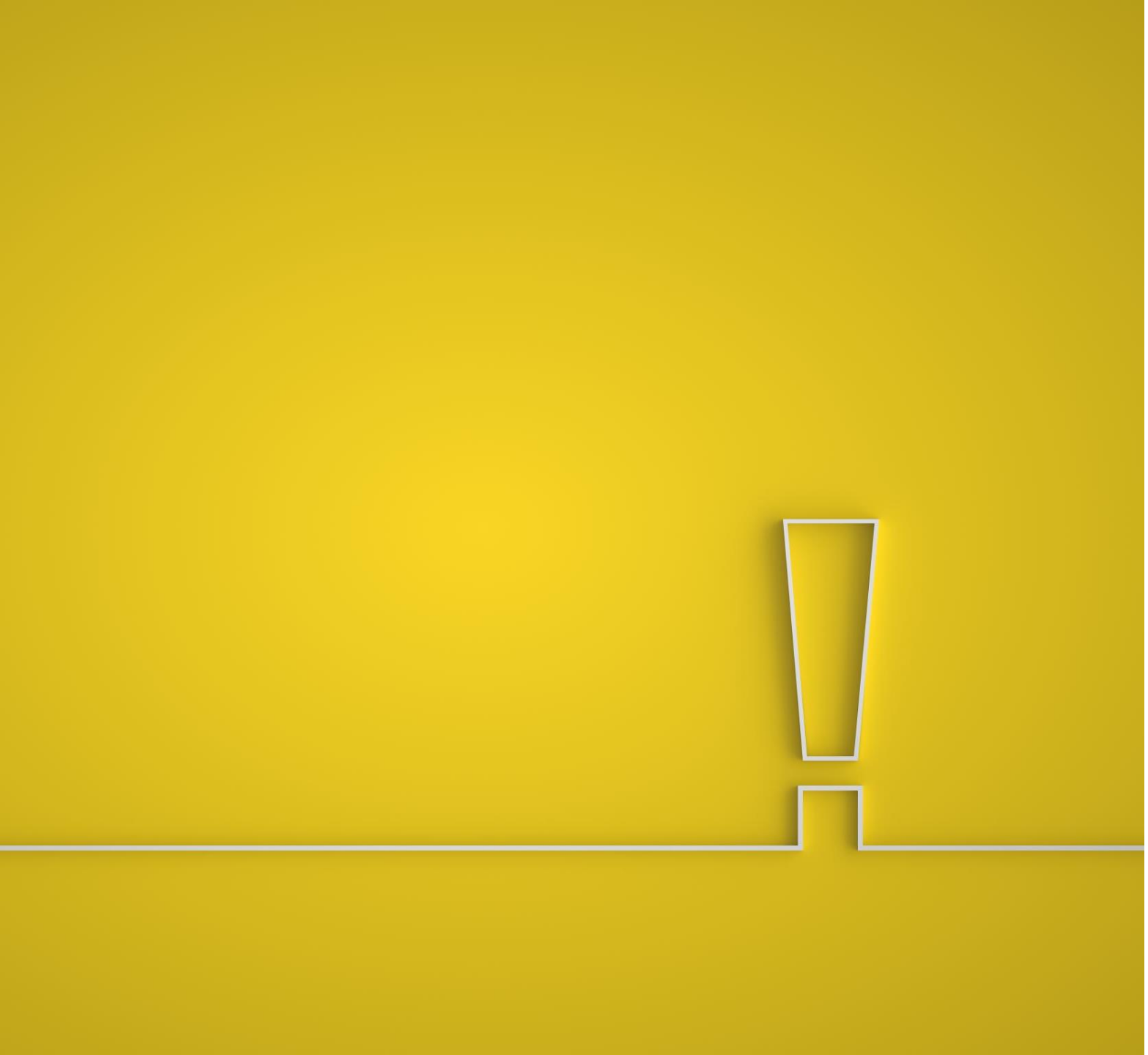
- How to find logical and programming errors (debug) your code
- Three strategies for isolating and rectifying problems
- With worked examples



let’s say a program compiles, but it doesn’t “work” to the spec



Or it’s crashing ‘somewhere’, how do you identify where?



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

“Debugging is the cornerstone of being a programmer. [...]
meaning [...] *to see into the execution of a program by examining it*.
A programmer that cannot debug effectively is blind.’’
--Robert L. Read,
author: How to be a Programmer

This means, you need to…

- understand every line, step by step
- as the computer executes it (detail, not big picture)
- lines are executed *in order*
- follow the loops, follow the branches, track the variables and their values, write them down if you have to! - *this is how we fix your programs in the lab!*
- *Yes, you may need to look stuff up!*

Approach 1: Finding mistakes using ‘dry running’ your code & ‘Rubber Duck’ debugging

How to dry run

- This may sound obvious, but to many it seems not to be ☺
  1. Start at the start (main)
  2. Follow the execution (control flow) of the program statement by statement
  3. Write down the value of any variables created (pay attention to their scope)
  4. Follow the logical path based on the variable state (around ifs, while loops etc.)

Notes: you will need to understand how statements and expressions are evaluated *in detail. Lookup any functions or operators you do not understand!*

Dry run example

Adding up a number sequence

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* result

Testing hypotheses using **printf**

```c
/* Function to turn a numeric grade from
   1 to 10 into an outcome */

void print_outcome_from_grade(int grade)
{
  // 1-4 fail

  if (grade >= 1 && grade < 5)
    printf("fail\n");
  else
    // 5-7 pass

    if (grade >= 5 && grade < 8)
      printf("pass\n");
    else
      // 8-10 distinction

      if (grade >= 8 && grade < 10)
        printf("distinction\n");
}
```

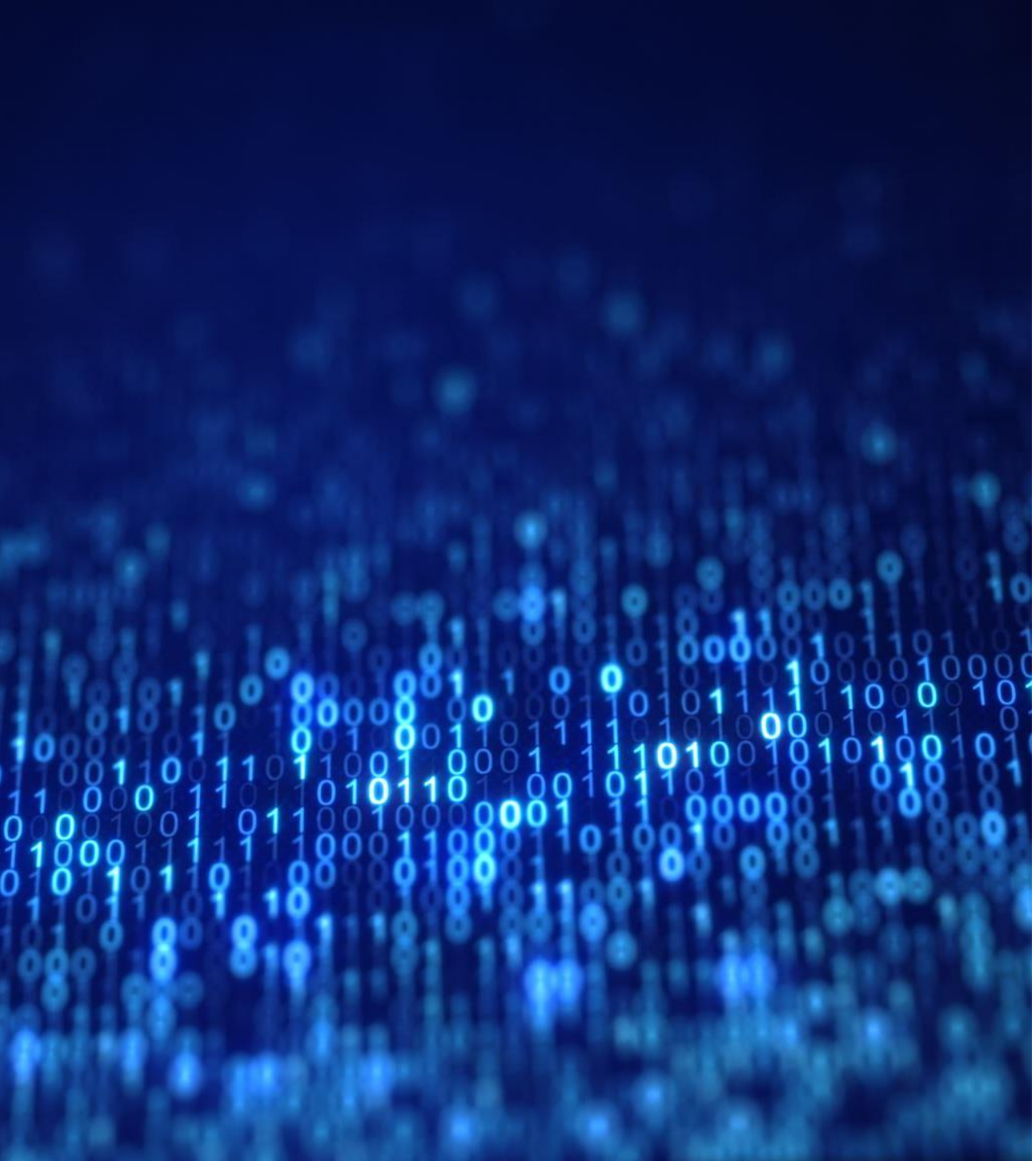
Walkthrough: Testing hypotheses using printf

Approach 3: Runtime debugging

Typically for more mysterious and hard to isolate faults – *note code in a debugger does not run identically to normal operation*

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 post ‘crash dump’ analysis, and finding confusing or unexpected runtime errors
  - E.g. memory errors (‘splat bugs’), concurrency, data dependent faults



Summary

- You should know what *debugging* is
- How to '*dry run*' your code in your head or on paper
- 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 (e.g. gdb), typically for hard to find errors or code forensics