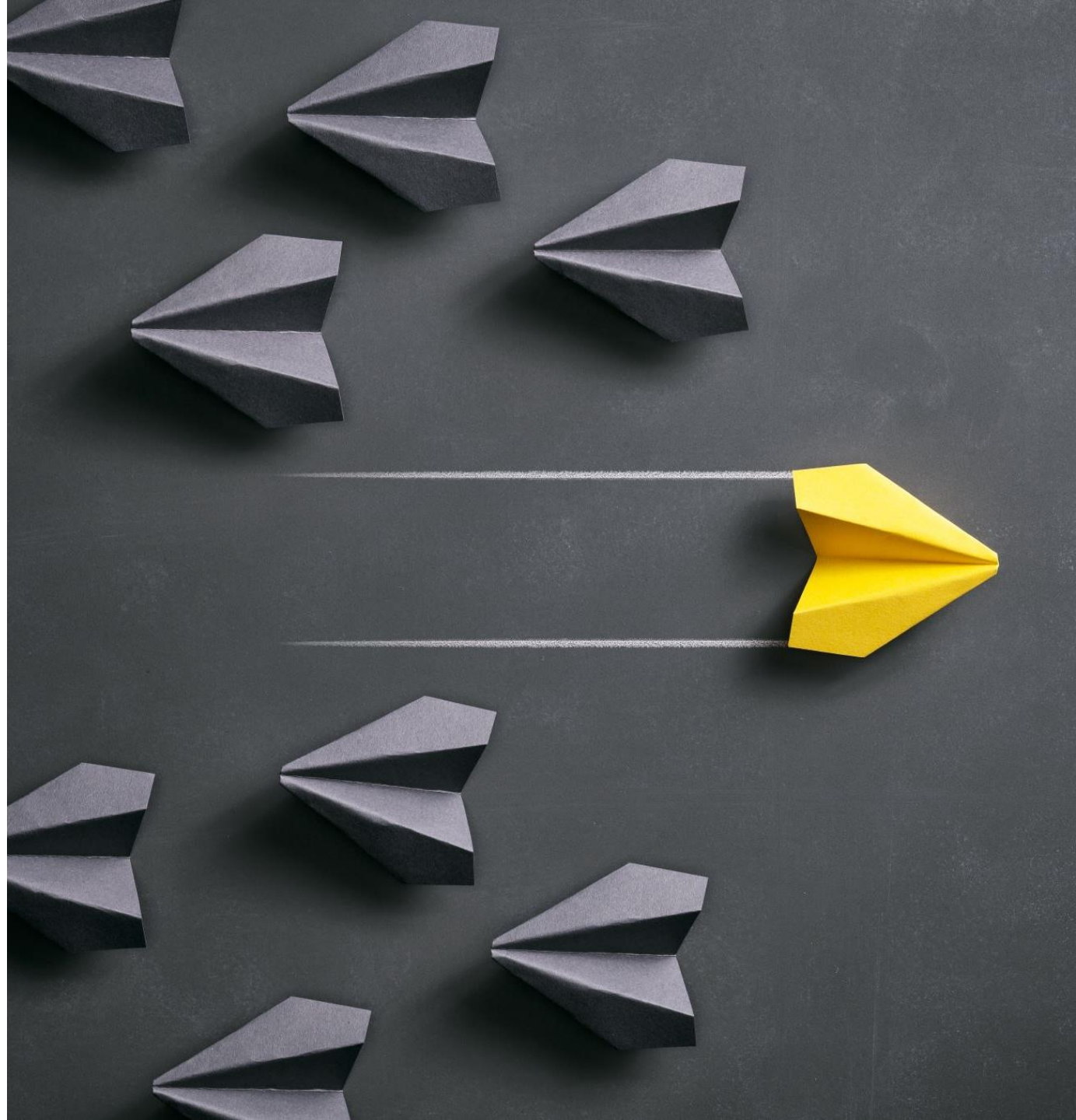


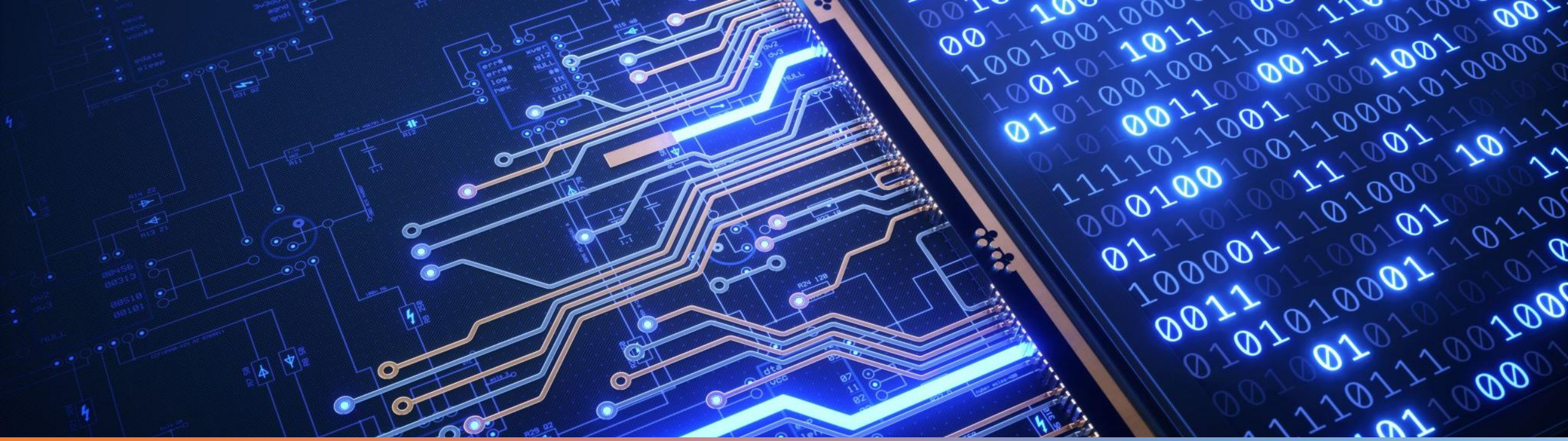
SCC.111 Software Development – Lecture 3: Control Flow

Adrian Friday, Nigel Davies, Hansi Hettiarachchi, Saad Ezzini

Today: Code in motion

- Intro, how code 'moves' (as it is executed)
- Controlling the flow of programs
- What makes this hard at first (thinking how the 'program state' changes around this flow)





Your code

- Looks like a static thing, but unlike a text document...
 - But when you execute it, “it comes alive” – there’s a thread of execution that ‘moves’ through the code
 - and we need to learn how to ‘see’ this when we read it



Analogies

An architect understands how a building is inhabited just from the plans

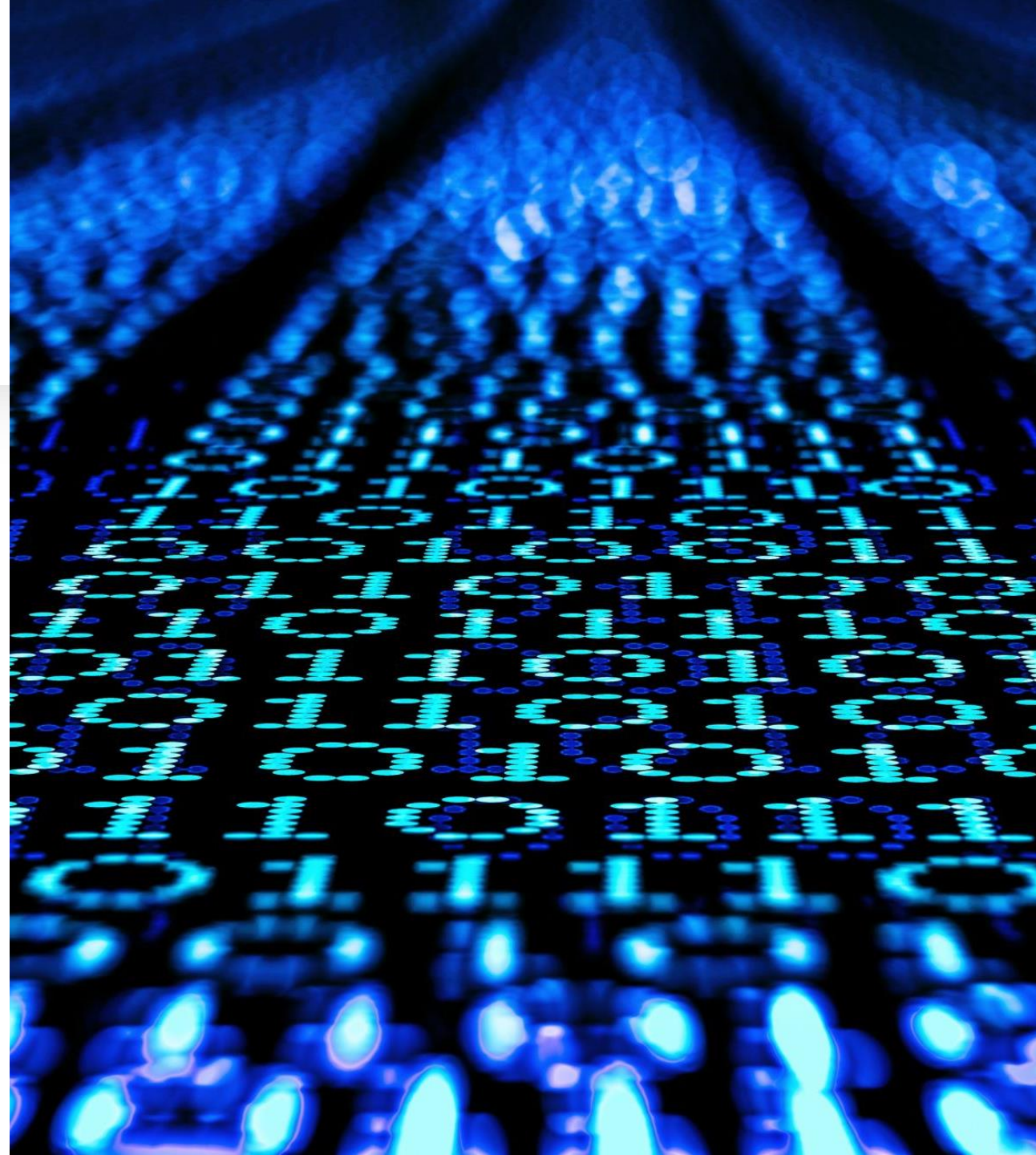


Analogies

A musician can 'hear' the performance just from reading the music

When writing our code...

- We need to see how it is executed to make sense of what it is going to do
 - ... *this means we also need to understand all those keywords, operators and function calls (like printf) and know what they do*





The code moves at different scales



Between programs... (later courses...)



Between blocks of code (later in *this* course)



Between statements of the program (*this* lecture)



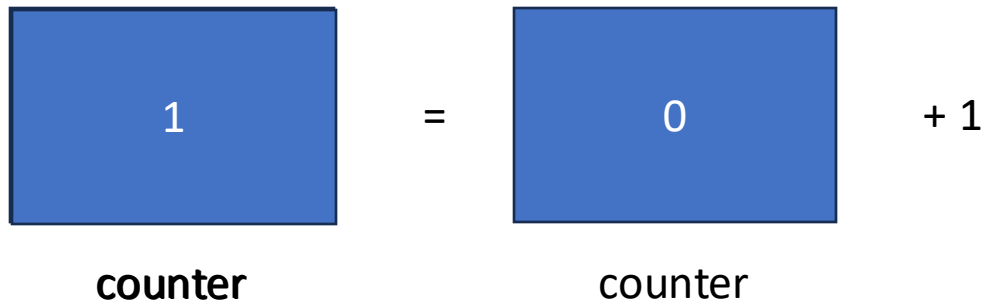
Within a line of code as expressions within the statements are evaluated (*this* lecture)

This is the 'flow' of evaluating this statement

counter = counter + 1

Evaluate the right hand side expression (RHS)

Then 'assign' it's value into the left hand side (LHS)



=1

Assign

Do (0 + 1)

Expression evaluation follows rules too

- Here, with *literals* (5 & 2), and *variables* (a, b):

`b = a * 5 + 2;`

a * 5 gets done first,
then, +2, then
'assign' to b

Brackets first

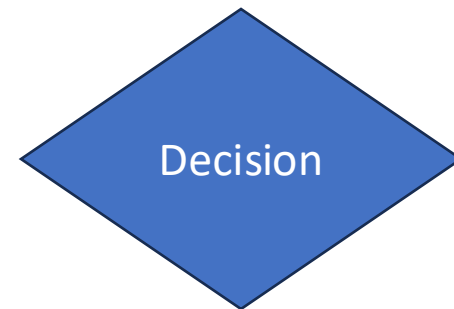
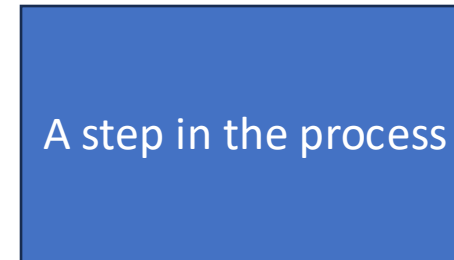
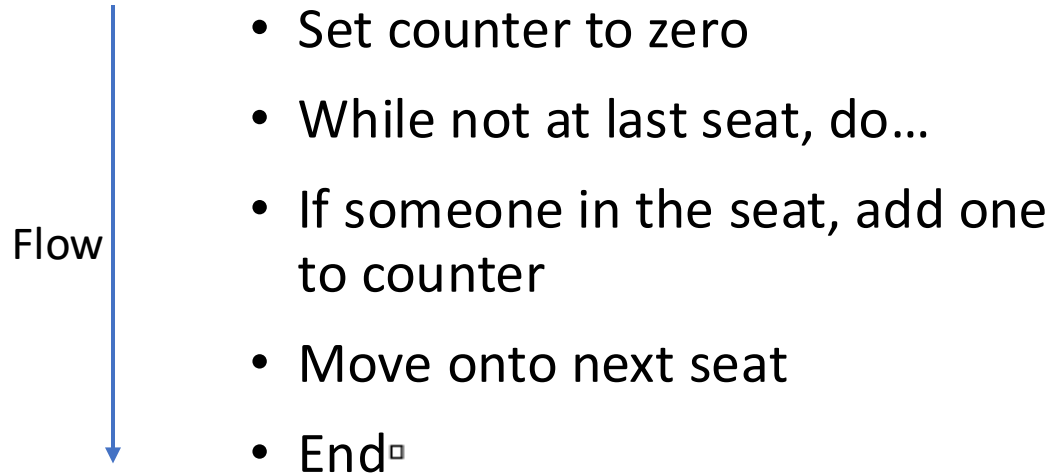
Orders (i.e. Powers and Square Roots, etc.)

Division and **M**ultiplication (left-to-right)

Addition and **S**ubtraction (left-to-right)

Programs 'flow' from statement to statement

- We decided to add one...



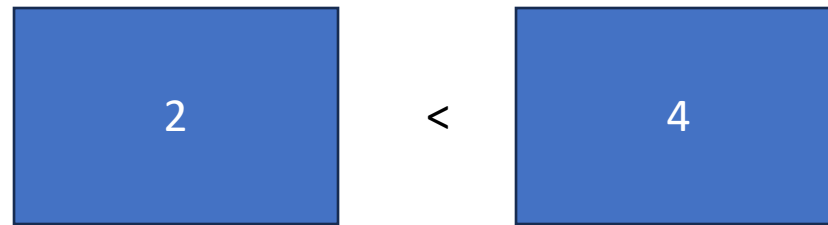
Logical operators allow us to make decisions that control flow conditionally

- < less than
- > greater than
- <= and >=
- == and !=
- ! Boolean NOT (unary, logically negates)
- && Boolean AND
- || Boolean OR

```
if (coffeesToBuy > 0) {  
    BuyACoffee();  
    coffeesToBuy--;  
}
```

```
if (coffeesToBuy > 0 &&  
    moneyLeft >= cost) {  
    BuyACoffee();  
    coffeesToBuy--;  
}
```


To control this we need to decide 'the truth'



totalBought

totalWanted

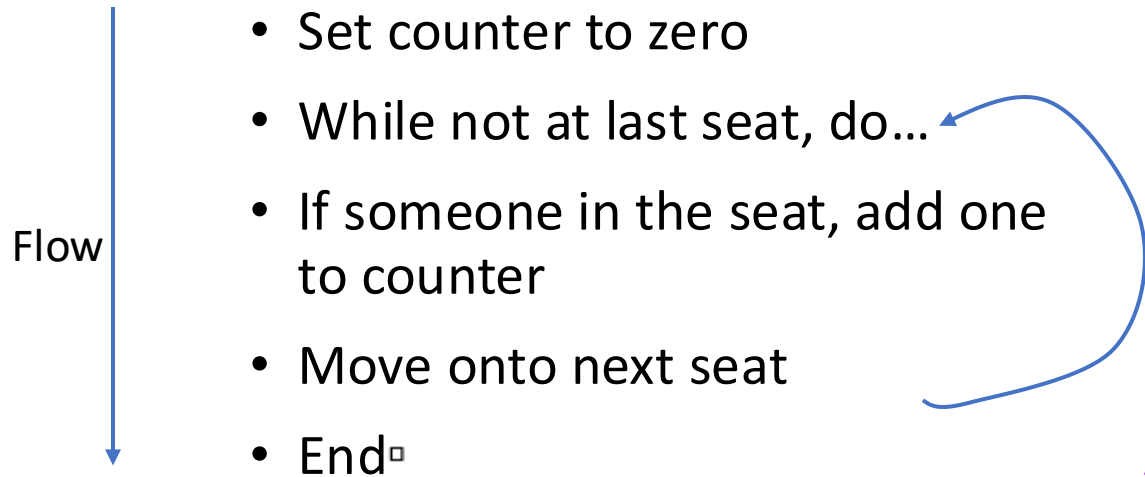
The result is logically 'TRUE', as 2 is less than 4

< is a binary logical operator which needs both a left and right hand side

2 *Is less than* 4

Making a decision to do something *again*

- Programs flow from start to finish, but we can do statements over again (repetition)



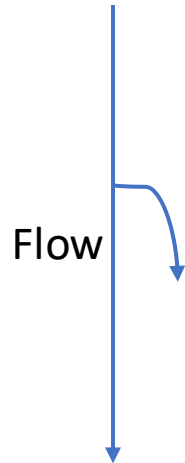
In C this is controlled with 'while', 'for', and 'do...while' 'loops', and goto (**not** used in practice):

```
#define NUMBER_OF_SEATS 280
```

```
while (seatNumber < NUMBER_OF_SEATS) {  
    seatNumber++;  
}
```

Deciding to select between paths (branching)

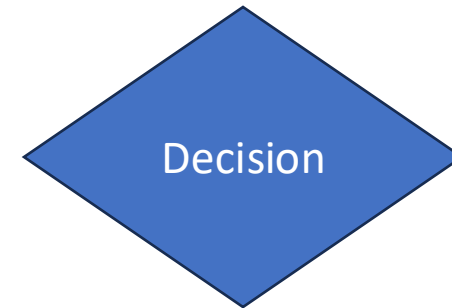
- Selection is done with 'if'



- If someone in the seat, add one to counter

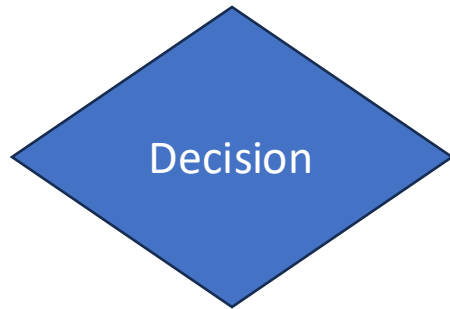
```
if (PersonInSeat(seatNumber) == 1) {  
    counter = counter + 1; // or counter++;  
}
```

We only go into the body of the 'if' if it's condition is logically TRUE



Or not...


- 'else' lets us say what to do if an expression is 'not TRUE' (paired only with 'if')



```
if (PersonInSeat(seatNumber) == 1) {  
    seatsOccupied++;  
}  
else {  
    unoccupiedSeats++;  
}
```

Tip: Also, anything non-zero is TRUE in C

```
if (1) { ... } // Always  
if (0) { ... } // Never
```



Worked
example:
program flow



Thinking about flow

- We need to start spotting these two 'shapes'
 - Where we are making decisions (control flow should 'branch') = if
 - Where we are repeating ourselves either a known or unknown number of times (control flow should 'loop') = while, for, do
 - the property to stay inside our loop is known as 'the loop invariant'

Homework:

Read the slides with more detailed explanations of forms of loop in C 😊

Flowing round the loop

```
#include <stdio.h>

int main()
{
    int greenBottles = 10;

    int bottlesFallen = 0;

    while (greenBottles > 0) {
        printf("%d green bottles, hanging on the wall, total that fell %d\n",
            greenBottles, bottlesFallen);

        greenBottles--;

        bottlesFallen++;
    }

    return 0;
}
```

Repeat code until **condition** is *false*
while (<continuation>)
 <statement>

What happens to greenBottles and
bottlesFallen each time round the
loop?

Loop invariant = ?

For several iterations

```
#include <stdio.h>

int main()
{
    int bottlesFallen = 0;

    for (int greenBottles = 10; greenBottles > 0; greenBottles--) {
        printf("%d green bottles, hanging on the wall, total that fell %d\n",
            greenBottles, bottlesFallen);

        bottlesFallen++;
    }

    return 0;
}
```

Repeat code until **condition** is *false*
for (<initialization>;
 <continuation>;
 <action>)
 <statement>

Where is the initialiser and loop counter now?


```
int x = 0;
```

```
do {  
    printf("Hi!\n");  
    x = x + 1;  
} while (x < 10);
```

```
do { <statement> }  
while (<expression>);
```

do...while loops test *after* the code block (execute body *at least once*)

You will also see 'jumps' out or round the loop in K&R

- Leaving loops early (**break**;)

```
#define NUMBER_OF_SEATS 280
```

```
while (1) {  
    // Do something  
    seatNumber++;  
    if (seatNumber == NUMBER_OF_SEATS)  
        break;  
    // Do something else  
}
```

- Jumping to the 'next iteration' early (**continue**;)

```
while (1) {  
    // Get input  
    scanf("%d", &seatNumber);  
    if (seatNumber > NUMBER_OF_SEATS)  
        continue;  
    // Do something with seatNumber  
}
```

But should you use them? Why/ why not?



Tips to help 'follow the flow'

- Need to start thinking of our state/ variables 'in motion' (as they might run)
 - i.e. as you go through paths in your program they could change conditionally
 - As you 'go around again', state will likely be updated each 'loop iteration'
 - As you read code, try to see it running in your mind
 - Writing down variables and their values and how they change will help!



Seen already!

auto

break

case

char

const

continue

default

do

double

else

enum

extern

float

for

goto

if

int

long

register

return

short

signed

sizeof

static

struct

switch

typedef

union

unsigned

void

volatile

while

Summary

- You should understand
 - The flow of execution of a statement using operators (+ - etc.)
 - That programs have a flow from the first to the last program statements
 - That statements can affect this control flow (if, while etc.)
 - How logical operators (<, ==, > etc.) result in TRUE/FALSE values can be used to control this
 - The importance of recognising branch and loop patterns and 'seeing the flow you want' in your algorithms