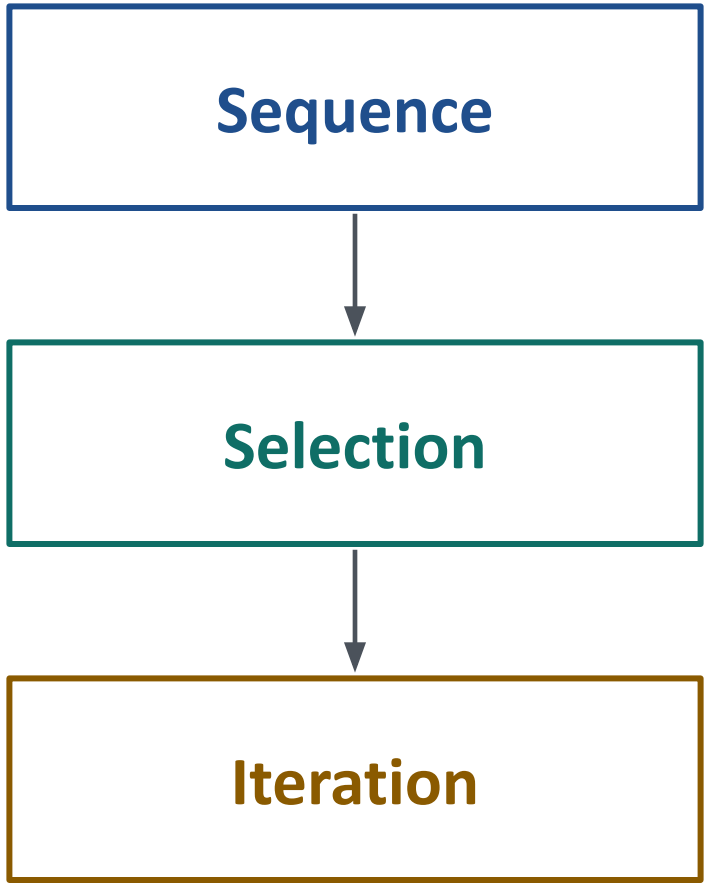
CSE 1111 — STRUCTURED PROGRAMMING LANGUAGE

Introduction to Structured Programming

Thinking in steps: how problems become programs

Sheikh Tanvir Ahmed
Lecturer
Department of Computer Science and Engineering
United International University (UIU)



the three building blocks

A computer does exactly what it is told, nothing more

TELLING A PERSON

"Add up these marks and tell me if she passed."

- Fills in the missing details itself
- Asks when something is unclear
- Guesses what you probably meant

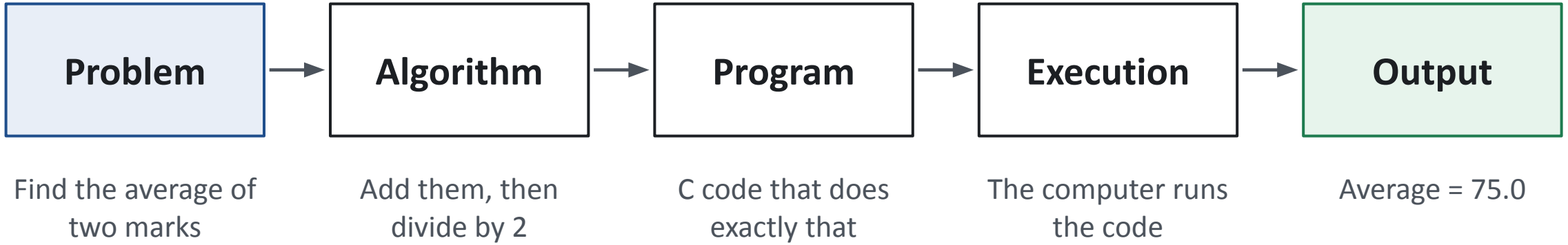
TELLING A COMPUTER

1. Start with total = 0
2. For each mark, add it to total
3. If total ≥ 40, print "Pass"
4. Otherwise print "Fail"

PROGRAMMING IS

writing instructions so precise that a machine with no common sense can follow them.

From a problem to an answer



NOTICE

The hard part is the middle: designing the algorithm. Writing the program is the translation step.

Same idea, two languages

ALGORITHM (plain steps)

1. Take the first mark
2. Take the second mark
3. Add them
4. Divide the sum by 2
5. Show the result

You do not need to understand the C yet.
Notice only that each line matches a step.

PROGRAM (C)

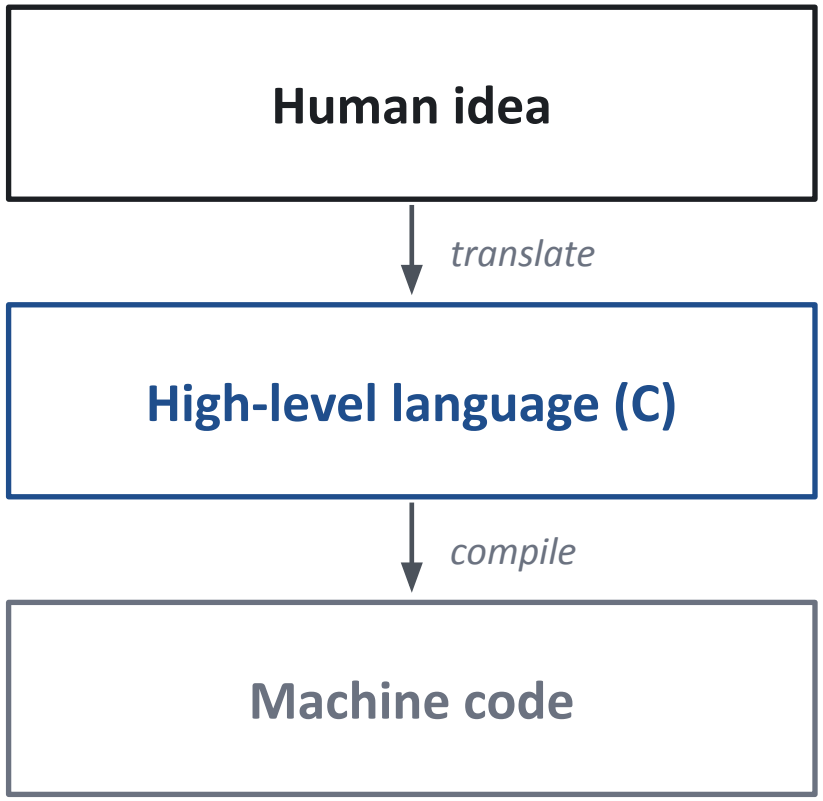
```c
#include <stdio.h>

int main(void) {
    double a = 70, b = 80;
    double avg = (a + b) / 2;
    printf("Average = %.1f\n", avg);
    return 0;
}
```

OUTPUT

```
Average = 75.0
```

A bridge between human thinking and machine instructions



"Average of two marks"

```
avg = (a + b) / 2;
```

binary instructions for one CPU family

WHY A LANGUAGE?

Machines only run machine code. Writing it by hand is slow and error-prone, so we write in a readable language and let a translator (a compiler for C) produce machine code.

A paradigm is a way of organising a program

Different problems suggest different ways of thinking. A paradigm is the style of thinking a language encourages.

Style	Organised around	Typical question
Structured	Clear control flow: sequence, selection, iteration	What steps, in what order?
Procedural	Procedures (functions) that perform tasks	What tasks can I reuse?
Object-oriented	Objects that combine data and behaviour	What things exist in this problem?

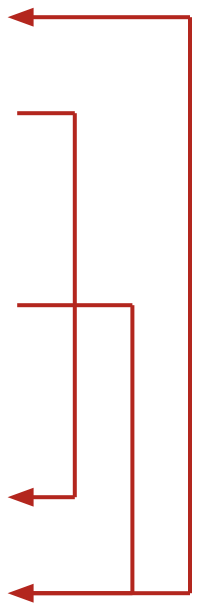
IN THIS COURSE

We learn structured programming in C. Procedural and object-oriented styles build on it.

Unstructured code is hard to follow

JUMPING AROUND (goto-style)

```
10  read mark
20  if mark < 40 goto 60
30  print Pass
40  goto 70
50  (never reached)
60  print Fail
70  if more goto 10
80  stop
```



BLOCKS WITH ONE ENTRY, ONE EXIT

repeat for every student:
    read mark
    if mark < 40
        print Fail
    else
        print Pass

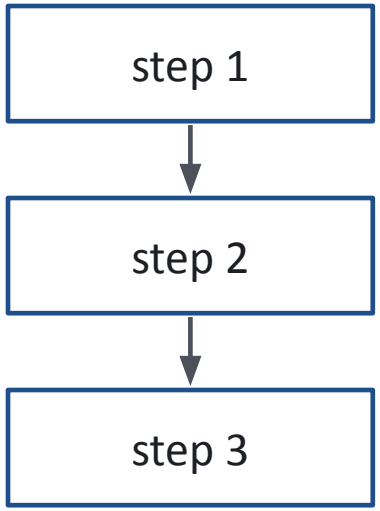
THE STRUCTURED IDEA

Build every program from a few simple blocks. You can read the code from top to bottom without chasing jumps.

Three control structures are enough

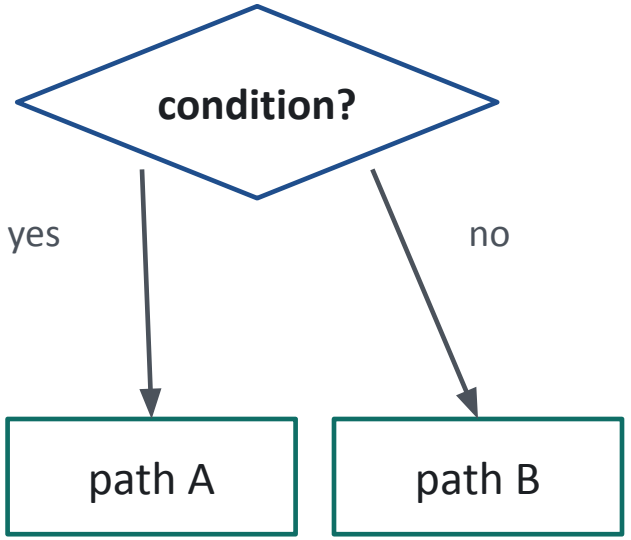
Sequence

one step after another



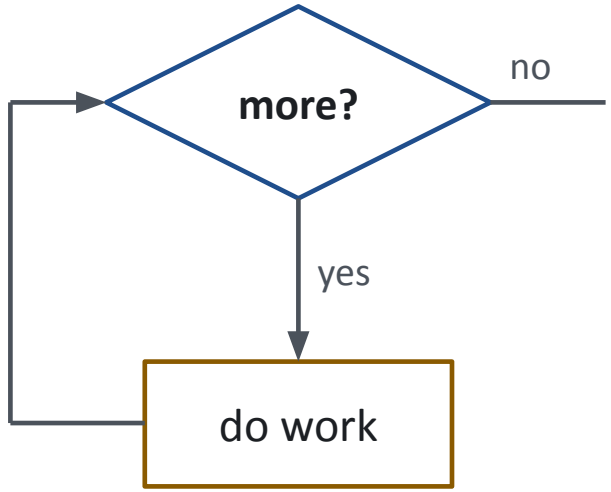
Selection

choose one of two paths



Iteration

repeat while needed

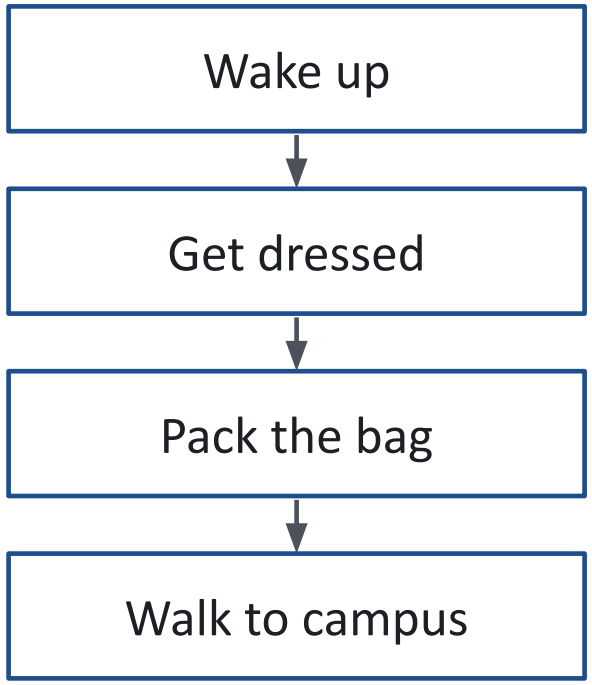


Every C program you will write in this course is built from just these three.

SEQUENCE

Order matters

Getting ready for an exam: each step needs the one before it.



THE SAME IDEA IN C: x must exist before y uses it

c

```c
#include <stdio.h>
int main(void) {
    int x = 5;
    int y = x * 2;
    printf("y = %d\n", y);
    return 0;
}
```

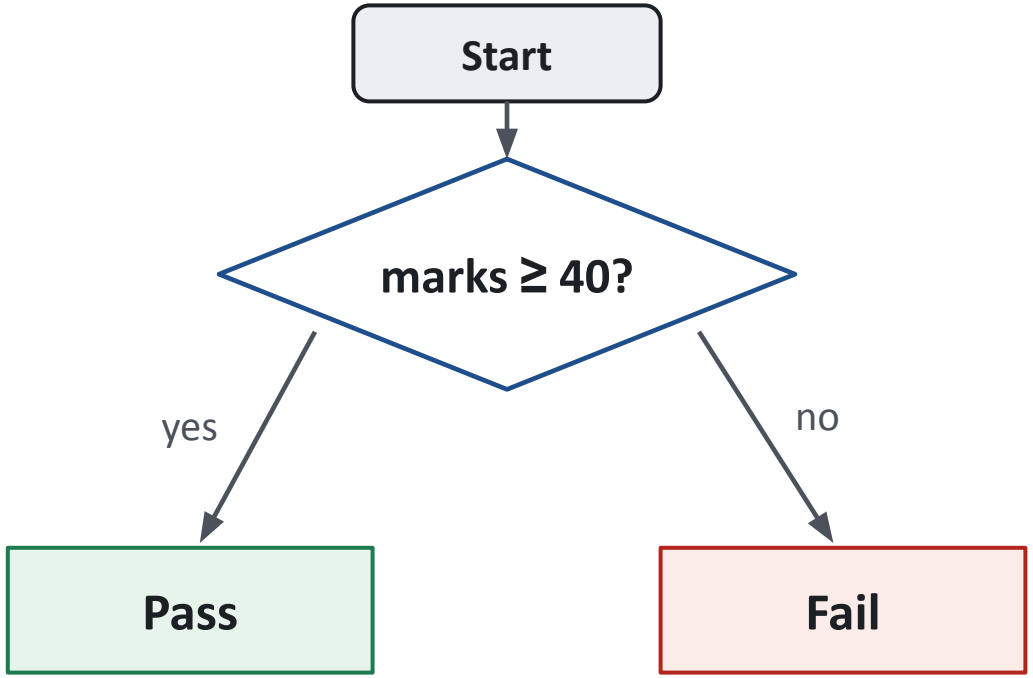
OUTPUT

```
y = 10
```

Statements run top to bottom, one after another.

A decision chooses one path

Rule (assumed for this example): pass if marks are at least 40.




```c
#include <stdio.h>

int main(void) {
    int marks = 52;
    if (marks >= 40)
        printf("Pass\n");
    else
        printf("Fail\n");
    return 0;
```

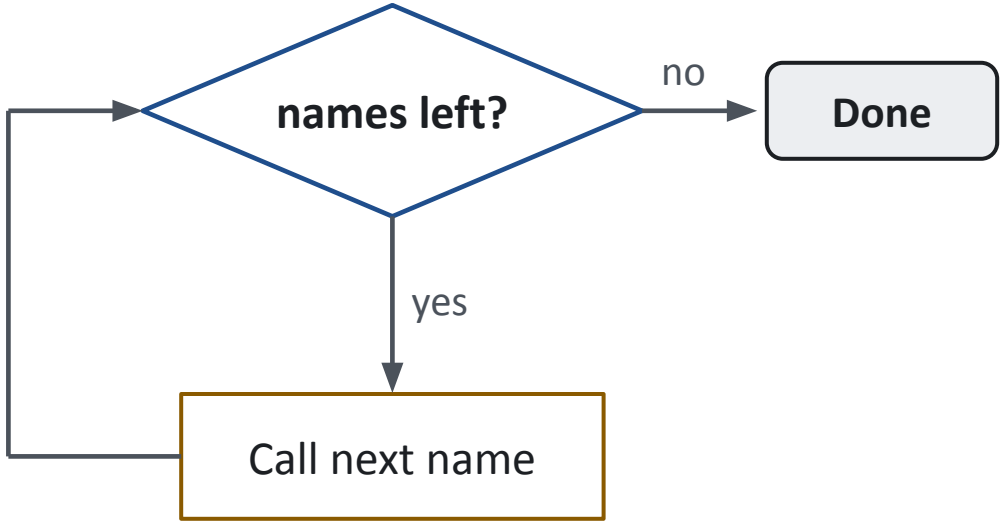
OUTPUT

```
Pass
```

What would be printed if marks were 35?

Repeat until the job is done

Marking attendance: call each name until the list ends.



One short loop replaces five (or five million) copies of the same line.

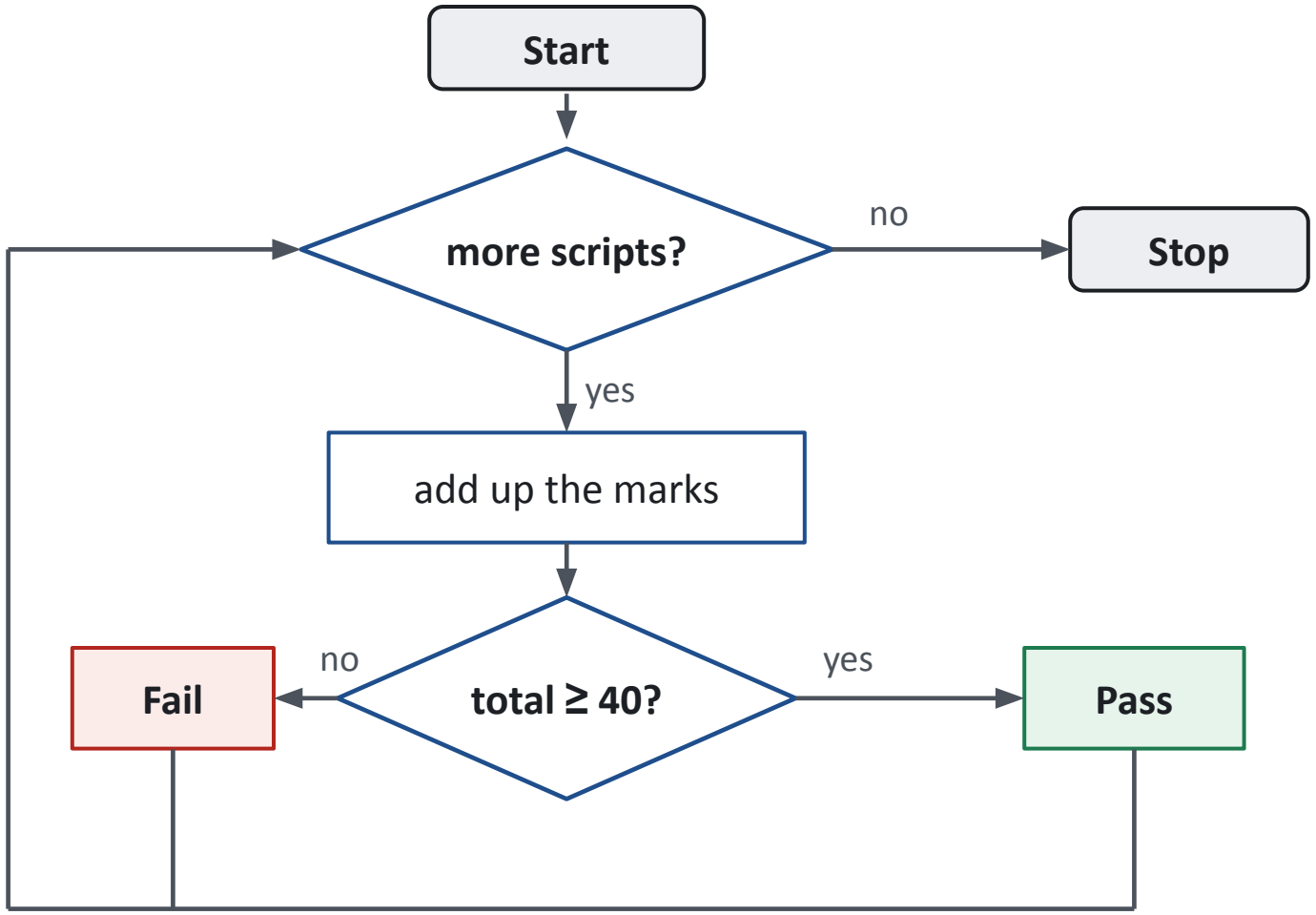
```c
#include <stdio.h>

int main(void) {
    for (int i = 1; i <= 5; i++)
        printf("Student %d\n", i);
    return 0;
}
```

OUTPUT

```
Student 1
Student 2
Student 3
Student 4
Student 5
```

One process, all three structures



Iteration

the loop that repeats for every script

Sequence

add up marks, then decide

Selection

pass or fail for each script

C gives us a statement for each structure

Structure	C statements
Sequence	statements run in order; { } groups them into a block
Selection	`if`, `if-else`, `else if`, `switch`
Iteration	`while`, `for`, `do-while`
Reusable pieces	functions such as `main()` and `printf()`

A NOTE ON GOTO

C also has `goto`, which jumps anywhere. Structured programmers avoid it; so will we.

A small language that shows you how programs really work

Few rules	A small set of keywords, so you can learn the whole core language.
Visible types and memory	You declare types and see how data is stored, which most languages hide.
A real compile step	You learn how source code becomes a running program.
A shared syntax	C++, Java, C# and JavaScript all borrow its braces, loops and conditions.

PRACTICE

Which control structure is it?

CLASSIFY	A traffic light turns green only when the road ahead is clear.
CLASSIFY	A teacher reads out every name on the register.
CLASSIFY	Buy a ticket, enter the hall, find your seat, sit down.
CLASSIFY	Keep stirring the soup until the sugar dissolves.
THINK	A vending machine: insert coins, choose an item, get it, or get your coins back. Which structures appear?

PRACTICE

From a real process to ordered steps

ORDER	Put these in a sensible order: pour water, add tea, boil water, drink, fill kettle.
CONVERT	Write the steps for withdrawing money from an ATM. Mark every step that depends on a decision.
CHOOSE	Which structure do you need to print the numbers 1 to 100? To decide whether someone may vote?
EXPLAIN	Why can’t you tell a computer “check if the mark is good”?

CHALLENGE

Who may sit for the examination?

> **ASSUME THIS RULE**
>
> A student may sit for the final examination if attendance is at least 70% **and** all dues are paid.

DESIGN	Write an algorithm that takes a student's attendance and dues status and prints Eligible or Not eligible.
THINK	What happens at exactly 70%? What if dues are unpaid but attendance is 100%?

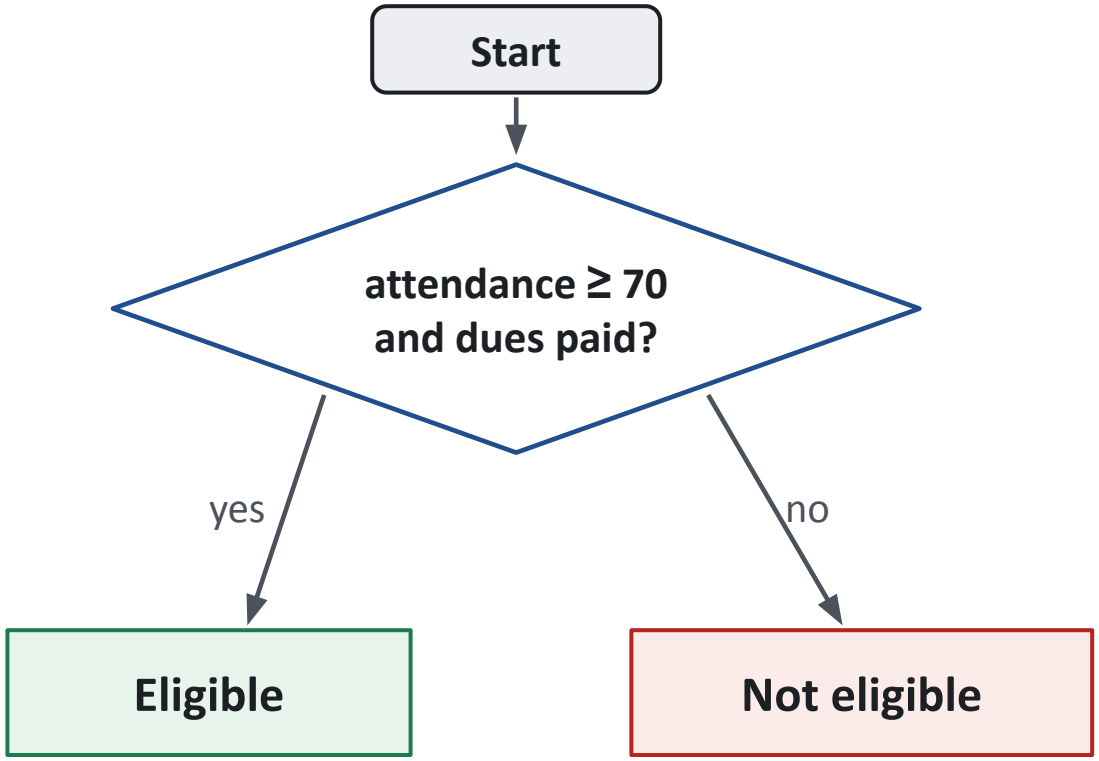
ONE POSSIBLE SOLUTION

Eligibility algorithm

1. Read attendance and dues status
2. **If** attendance ≥ 70 **and** dues are paid:

   print Eligible
3. **Otherwise**:

   print Not eligible



CHECK THE EDGES

Exactly 70% is eligible because the rule says “at least”. Both conditions must hold.

COMMON MISUNDERSTANDINGS

Precision is part of the job

What people say	What goes wrong	Be precise
“Pass if the mark is above 40”	Is 40 itself a pass? “Above” says no.	marks ≥ 40 (or marks > 40, deliberately)
“Add the marks”	Which marks? How many? Starting from what?	Start total at 0, add each of the 5 marks
“Repeat until finished”	Finished by what test?	Repeat while names are left
“Code is the hard part”	A wrong plan gives a wrong program.	Design the algorithm first

SUMMARY AND NEXT STEP

What to carry forward

- A **problem** becomes an **algorithm**, then a **program**.
- A language lets us state steps a machine can follow.
- **Structured programming** builds everything from sequence, selection and iteration.
- C has a statement for each.

NEXT

If we want a computer to remember information while a program runs, how can we store it?



a place with a name and a value