

Programming concepts

Read online at pastopic.com/cambridge/computer-science-0478/notes/programming-concepts

Last checked 30 September 2026

Read first: **Program development and design**

© 2026 Pastopic. Free for personal study. Please share the link, not the file.

What you need to know

By the end of this topic you should be able to:

1. **Declare and use variables and constants:** give each one a name and a data type, store values in it, and know when a value should be a constant instead of a variable.
2. **Use the five basic data types:** integer, real, char, string and Boolean.
3. **Use input and output:** read a value into a variable, and display values and messages.
4. **Use the building blocks of every program:**
 - (a) **sequence:** statements run one after another, in order;
 - (b) **selection:** choose what to do with IF statements and CASE statements;
 - (c) **iteration:** repeat statements with count-controlled (FOR), pre-condition (WHILE) and post-condition (REPEAT ... UNTIL) loops;
 - (d) **totalling and counting;**
 - (e) **string handling:** find a string's length, take out part of it (a substring), and change it to upper or lower case;
 - (f) **operators:** arithmetic (+ − * / ^ MOD DIV), relational (= < <= > >= <>) and logical (AND OR NOT).
5. **Nest statements:** put selection and loops inside each other, up to three levels deep.
6. **Write a maintainable program:** one that another programmer can easily read, fix and change, by using meaningful identifiers (for variables, constants, arrays, procedures and functions), comments, and procedures and functions.

Writing your own procedures and functions, local and global variables, and the library routines ROUND and RANDOM are in the note on **Procedures, functions and library routines**; arrays and files have their own notes. This note uses them only where a question needs them.

Understand it

A program is a list of instructions that the computer carries out. In Paper 2 you read and write these instructions in Cambridge **pseudocode**, a halfway house between English and a programming language. It has a fixed set of keywords (in capitals) so examiners can read it, but minor slips in syntax are not penalised as long as the meaning is clear.

Variables and constants

A **variable** is a named place in memory that holds one value, which can **change** while the program runs. Before using a variable, you **declare** it: give its name (its **identifier**) and its data type.

```
DECLARE Score : INTEGER
DECLARE Surname : STRING
Score ← 0
Score ← Score + 5
```

The arrow ← means **assignment**: work out the value on the right, then store it in the variable on the left. `Score ← Score + 5` takes the old value of `Score`, adds 5, and stores the answer back in `Score`. The old value is lost.

A **constant** is also a named value, but it is fixed when the program is written and **cannot change while the program runs**. It is declared with its value, which must be a literal (a number, character or string written out, not a calculation):

```
CONSTANT SeatsPerCoach ← 52
CONSTANT ShopName ← "Green Leaf Grocers"
```

Use a constant for a value the program needs but must never alter. The benefits:

- the name says what the number means, so `Coaches ← Pupils / SeatsPerCoach` is easier to read than `Pupils / 52`;

- if the value changes one day (bigger coaches), you edit it **in one place** and every use updates;
- the program can't change it by accident.

Naming rules. In Cambridge pseudocode an identifier is made of letters and digits only, starts with a capital letter, and marks each new word with another capital (`TotalCost`, `Player2Name`). No spaces, underscores or other symbols (not `Total Cost` or `Total_Cost`), it can't start with a digit, it can't be a keyword such as `OUTPUT`, and once you have chosen a name, use exactly the same name every time. Pseudocode treats upper and lower case as the same, so `score` and `Score` are one variable.

Data types

The data type tells the computer what kind of value is stored, and so what it can do with it.

Data type	Holds	Examples
INTEGER	a whole number, positive, negative or zero	-4, 0, 2026
REAL	a number that can have a fractional (decimal) part	3.75, -0.5, 100.0
CHAR	exactly one character, in single quotes	'Q', '7', '#'
STRING	any number of characters (letters, digits, spaces, symbols), in double quotes	"Blue Whale", "AB12 9ZZ", ""
BOOLEAN	one of two values: TRUE or FALSE	TRUE

Choosing well:

- a count or anything you can't have a fraction of (people, goals, pages) is an **integer**; money, lengths, weights and averages are **real**;
- a phone number, postcode or ID with leading zeros or spaces is a **string**, because you never do arithmetic with it and a number would lose the leading 0;
- a yes/no fact, or a **flag** that records whether something has happened, is **Boolean**;
- "7" (a string) and 7 (an integer) are different: only the integer can be added to.

Input and output

`INPUT` takes a value from the user and stores it in a variable. `OUTPUT` displays values; separate several values with commas, and put fixed text in quotes.

```
OUTPUT "How many tickets do you want?"
```

```
INPUT Tickets
```

```
OUTPUT "You have booked ", Tickets, " tickets"
```

Always put a **prompt** (a message) before an input, and a **message** with every result, so the user knows what to type and what the numbers mean.

Operators

Arithmetic operators do calculations:

Operator	Meaning	Example	Result
+ - *	add, subtract, multiply	6 * 7	42
/	divide (the answer can have a decimal part)	17 / 5	3.4
^	raise to the power of	2 ^ 5	32
DIV	whole-number division: the quotient, with the fractional part thrown away	DIV (17, 5)	3
MOD	the remainder after whole-number division	MOD (17, 5)	2

You may write `17 DIV 5` and `17 MOD 5` instead; both forms are accepted. `DIV` and `MOD` go together: 17 people in cars of 5 means `DIV (17, 5) = 3` full cars and `MOD (17, 5) = 2` people left over. Two handy uses:

- `MOD (Number, 2) = 0` is true exactly when `Number` is even;
- `MOD (Number, 10)` is the last digit and `DIV (Number, 10)` removes it: for 4096 they give 6 and 409.

As in maths, `^` is done first, then `*` and `/`, then `+` and `-`: `2 + 3 * 4` is 14, but `(2 + 3) * 4` is 20. Use brackets whenever the order matters.

Relational operators compare two values and give TRUE or FALSE: = (equal to), <> (not equal to), <, <= (less than or equal to), >, >=. They are what go in the conditions of IF statements and loops.

Logical operators join or reverse conditions:

- A AND B is true only when **both** are true: Age >= 13 AND Age <= 19 means a teenager;
- A OR B is true when **at least one** is true: Mark < 0 OR Mark > 100 means an impossible mark;
- NOT A reverses a condition: NOT Finished is true while Finished is FALSE.

"From 13 to 19 inclusive" needs >= and <=; "between 13 and 19" with no "inclusive" is ambiguous, so read the question carefully.

Sequence

Sequence means the statements are carried out one after another, in the order written. Order matters: in the code below, swapping the last two lines would output 0 whatever the user types.

```
INPUT Width
INPUT Length
Area ← Width * Length
OUTPUT "Area: ", Area
```

Selection

Selection chooses which statements to run, depending on a condition.

An IF statement runs a block only when its condition is true; the optional ELSE block runs when it is false:

```
IF Temperature > 30
  THEN
    OUTPUT "Heat warning"
  ELSE
    OUTPUT "Normal conditions"
ENDIF
```

For more than two outcomes, **nest** a second IF inside the ELSE. Test the ranges in order so each value lands in exactly one branch:

```
IF Points >= 70
  THEN
    Award ← "Gold"
  ELSE
    IF Points >= 40
      THEN
        Award ← "Silver"
      ELSE
        Award ← "Bronze"
    ENDIF
  ENDIF
ENDIF
```

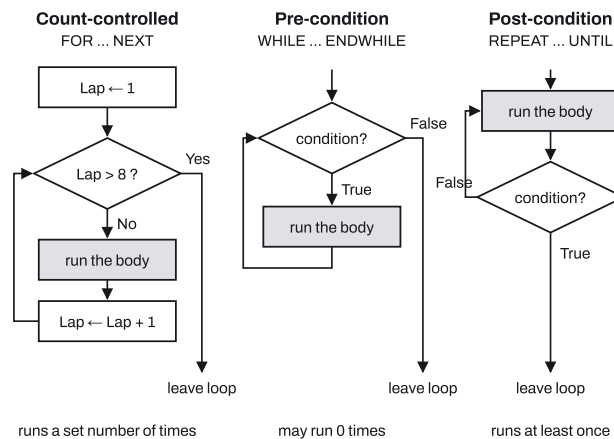
A CASE statement picks one branch by comparing **one variable** with a list of single values. OTHERWISE (always last) catches every value not listed. The branches are checked in order and only the first match runs.

```
INPUT Button
CASE OF Button
  'R' : OUTPUT "Red light"
  'A' : OUTPUT "Amber light"
  'G' : OUTPUT "Green light"
  OTHERWISE OUTPUT "Unknown button"
ENDCASE
```

Use CASE when you compare one variable with several exact values (menu choices, codes); use IF for ranges and for conditions that mix variables or use AND/OR.

Iteration

Iteration means repeating a group of statements in a loop. There are three kinds, and the difference is **when the program decides whether to go round again**.



Count-controlled loop (FOR ... NEXT). Use it when you know how many times to repeat before the loop starts. The loop variable counts for you: it takes each value from the start to the end in turn, and NEXT moves it on.

```
FOR Lap ← 1 TO 8
  OUTPUT "Starting lap ", Lap
NEXT Lap
```

This runs 8 times. STEP changes the jump: FOR Seat ← 2 TO 10 STEP 2 gives 2, 4, 6, 8, 10 (five times), and FOR Down ← 5 TO 1 STEP -1 counts backwards. In pseudocode a FOR loop always ends with NEXT and the variable's name, never ENDFOR.

Pre-condition loop (WHILE ... DO ... ENDWHILE). The condition is tested **before** each time round. The loop keeps going **while** the condition is true, so if it is false at the start, the body runs **zero** times.

```
Balance ← 50
WHILE Balance >= 12 DO
  Balance ← Balance - 12
  OUTPUT "Ride taken. Money left: ", Balance
ENDWHILE
```

With 50 at the start this runs 4 times (38, 26, 14, 2) and stops when Balance is 2. If Balance started at 10, it would not run at all.

Post-condition loop (REPEAT ... UNTIL). The condition is tested **after** each time round, so the body always runs **at least once**. It keeps going **until** the condition becomes true.

```
REPEAT
  OUTPUT "Enter a PIN of 4 digits"
  INPUT Pin
UNTIL Pin >= 1000 AND Pin <= 9999
```

Notice that the conditions work in opposite directions: WHILE loops **while true**, REPEAT loops **until true**. To turn one into the other, reverse the condition: WHILE Balance >= 12 becomes UNTIL Balance < 12.

WHILE and REPEAT are both **condition-controlled** loops: use one when you don't know in advance how many times it must run, for example until the user types a valid value or a stop value.

Converting a FOR loop into a condition-controlled loop. You must now do the counting yourself: set the counter before the loop, add 1 inside it, and test it in the condition.

```
Lap ← 1
WHILE Lap <= 8 DO
  OUTPUT "Starting lap ", Lap
  Lap ← Lap + 1
ENDWHILE
```

Totalling and counting

Totalling keeps a running total: a variable set to 0 before the loop has **each value added** to it inside the loop.

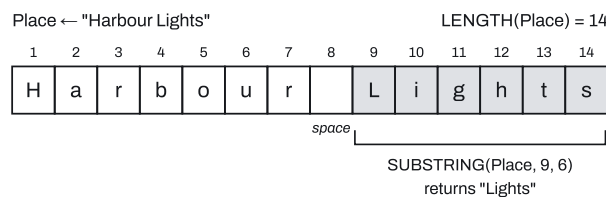
Counting keeps a tally: a variable set to 0 before the loop has **1 added** each time something happens (usually inside an IF).

```
Total ← 0
Heavy ← 0
FOR Parcel ← 1 TO 5
  INPUT Weight
  Total ← Total + Weight
  IF Weight > 20
    THEN
      Heavy ← Heavy + 1
  ENDIF
NEXT Parcel
OUTPUT "Total weight: ", Total, " kg; heavy parcels: ", Heavy
```

For weights 12, 25, 8, 31 and 19 this outputs a total of 95 kg and 2 heavy parcels. `Total ← Total + Weight` is totalling; `Heavy ← Heavy + 1` is counting. Starting either at 0 **before** the loop matters: set inside the loop, it would be wiped every time round.

String handling

A string is a row of characters, each with a **position**. In Cambridge pseudocode the first character is position 1 (some programming languages start at 0; the syllabus accepts either, so say which you are using if it matters). Spaces and punctuation are characters too.



The four string operations, with `Place ← "Harbour Lights"`:

Operation	Returns	Example	Result
LENGTH(s)	the number of characters, spaces included	LENGTH(Place)	14
SUBSTRING(s, start, length)	length characters, beginning at position start	SUBSTRING(Place, 1, 7)	"Harbour"
UCASE(s)	the string or char in upper case	UCASE(Place)	"HARBOUR LIGHTS"
LCASE(s)	the string or char in lower case	LCASE('K')	'k'

- SUBSTRING takes **a start and a length**, not a start and an end: the characters from position 9 to 14 are SUBSTRING(Place, 9, 6), because $14 - 9 + 1 = 6$.
- One character at a time:** SUBSTRING(Word, Position, 1) inside FOR Position ← 1 TO LENGTH(Word) visits every character in turn. This is how you count, check or change particular characters.
- Comparing without worrying about case:** UCASE(Answer) = "YES" is true for yes, Yes and YES.
- A length check** (making sure an input has the right number of characters) is written with LENGTH, for example LENGTH>Password) >= 10.

Nested statements

Statements can go inside each other: an IF inside a loop, a loop inside a loop, an IF inside an IF. This is **nesting**. You won't be asked to write more than three levels.

```
FOR Row ← 1 TO 3
  FOR Column ← 1 TO 4
    OUTPUT Row * Column
  NEXT Column
NEXT Row
```

The inner loop runs all 4 times for **each** pass of the outer loop, so the OUTPUT runs $3 \times 4 = 12$ times. Nested loops are the natural way to visit every cell of a 2D array or every round of every competitor. Close inner blocks before outer ones (NEXT Column before NEXT Row) and indent each level so the structure is visible.

Maintainable programs

A program is **maintainable** when a different programmer, months later, can quickly understand it and safely change it. The syllabus names these techniques:

- **Meaningful identifiers** for variables, constants, arrays, procedures and functions: DiscountRate says what it holds; D or X2 doesn't.
- **Comments**: notes for humans that the computer ignores. In pseudocode they start with //. A good comment says what a section does or why, for example // keep asking until the age is from 5 to 18.
- **Procedures and functions**: split the program into named sections, each doing one job. Each can be tested on its own and reused instead of copying the same code.
- **Constants** for fixed values, so a change is made in one place.
- **Indentation and white space**, so the blocks inside loops and IF statements stand out.

Key facts and formulas

There is no formula sheet for Paper 2; learn these.

Fact	Status
DECLARE Name : TYPE declares a variable; CONSTANT Name ← value declares a constant	Learn it
Variable: value can change while the program runs. Constant: value is fixed and can't change while it runs	Learn it
Five data types: INTEGER, REAL, CHAR, STRING, BOOLEAN	Learn it
← assigns; = compares	Learn it
DIV(a, b) whole-number quotient; MOD(a, b) remainder; ^ power; / ordinary division	Learn it
Relational: = <> < <= > >=; logical: AND OR NOT	Learn it
IF ... THEN ... ELSE ... ENDIF; CASE OF ... OTHERWISE ... ENDCASE	Learn it
FOR ... TO ... NEXT count-controlled; WHILE ... DO ... ENDWHILE pre-condition (may run 0 times); REPEAT ... UNTIL post-condition (runs at least once)	Learn it
Totalling: Total ← Total + Value; counting: Count ← Count + 1; both start at 0 before the loop	Learn it
LENGTH(s), SUBSTRING(s, start, length), UCASE(s), LCASE(s); first character is position 1	Learn it
Characters from position a to b: SUBSTRING(s, a, b - a + 1)	Learn it
Comments start with //	Learn it
Maintainable: meaningful identifiers, comments, procedures and functions (also constants, indentation)	Learn it

How to do it

In the Paper 2 questions we have tagged for this topic (115 questions, 2021–2025), 2 are only about test data and validation, so 113 appear in the table. 27 of them mix more than one type, so the counts add up to 147. Papers from 2021–2022 began with a pre-release question; since 2023 every Paper 2 ends with a 15-mark program.

Question type	Questions
Write a 15-mark scenario program	21
Write a short pseudocode algorithm	20
Modify an algorithm	15
Answer pre-release task questions (2021–2022 only)	14
Identify or match programming constructs	12
Find and correct errors in given pseudocode	10
Explain how to make a program maintainable, or suggest meaningful identifiers	10
Identify, match or describe data types	9
Write string-handling statements	8
Declare variables with a data type	6
Use or classify operators	6
State the purpose of an algorithm or describe its processes	6
Describe variables and constants	5
Complete an incomplete algorithm	5

1. Write a 15-mark scenario program (21 questions)

The last question on the paper. A scenario names one or more arrays (often a 1D array of names and a 2D array of values for the same people, at the same index) and lists requirements as bullets. You may answer in pseudocode or a programming language. It is marked as a whole, in bands: up to **9** for using the right techniques and data structures to meet every requirement, and up to **6** for a clear, commented, logical, accurate solution.

- Read the whole question twice.** Underline every array and variable name given, and use those names exactly. Note whether you must declare things (sometimes "you do not need to declare any arrays or variables", sometimes "must be declared").
- Plan one section per bullet**, in the order given, each starting with a comment.
- Inputs:** prompt, input, and validate with a loop that repeats until the value is acceptable (not a single IF). Put it inside the loops that visit each person and each item, so every value is checked.
- Use nested FOR loops** for 2D arrays: outer loop for the person (row), inner loop for the item (column).
- Build results from the standard patterns:** totals and counts set to 0 in the right place (inside the outer loop if they are per person), averages after the loop, bands with nested IFs, and for a highest or lowest value a variable that starts at the first value (or an extreme value) and is replaced when a bigger (smaller) one turns up.
- Output every result with a message**, and output names rather than index numbers when the question asks "who" or "which".
- Check each bullet off** at the end. A missing requirement costs more than a small syntax slip, which is ignored.

Variations you will meet:

- A menu:** wrap the program in REPEAT ... UNTIL Choice = <stop value>, validate the choice, and use CASE or IF to run the chosen part.
- Procedures or functions required:** write them with the names and parameters given, and call them.
- Random numbers or sorting** as one of the bullets: use ROUND(RANDOM() * n, 0) or a bubble sort (see the notes on library routines and on standard methods).
- Per-person bands and counts:** for each person decide the band with nested IFs, output it, and add 1 to that band's counter; output the counters after the outer loop.

2. Write a short pseudocode algorithm (20 questions)

Worth 4–6 marks, one mark per point. The points are almost always drawn from: a prompt and INPUT into a sensible variable; the calculation; the right condition in an IF or CASE; a loop with the right start and stop; and OUTPUT of the result with a message.

1. List the bullets in the question; each usually earns a mark.
2. Write the statements in that order. Initialise any total, counter or flag **before** a loop.
3. Use brackets in calculations, for example `Cost ← (Bags * 4.50) + Delivery`.
4. Finish with an output that has a message.

Variations you will meet:

- **A CASE statement on a menu number** with an OTHERWISE for anything else, for example `OTHERWISE OUTPUT "Invalid choice"`.
- **An IF with a discount or surcharge**: calculate the full amount first, then IF the condition holds, change it (`Total ← Total * 0.9` takes 10% off).
- **Check a string for a character**: use a flag set to FALSE, a position starting at 1, and a condition-controlled loop that looks at `SUBSTRING(Text, Position, 1)` and stops when the character is found **or** the position passes `LENGTH(Text)`. Afterwards, output one message for found and another for not found. (For example, checking whether a product code contains a hyphen.)
- **Convert a flowchart to pseudocode**: each process box becomes an assignment, a decision that loops back becomes a WHILE or REPEAT loop, a decision that doesn't loop back becomes an IF. Any correct loop type is accepted. If a loop only repeats a simple sum, one operator may do the same job and is also accepted: a loop that keeps taking 7 away from Number until less than 7 is left gives the same answer as `MOD(Number, 7)`.

3. Modify an algorithm (15 questions)

You are asked to change a given algorithm, either by describing the change ("explain how ... any code used must be fully explained") or by rewriting it.

1. Say **what** you add and **where** (after which line, inside or outside the loop).
2. If you write code, explain each line in words as well when the question says "explain".

Variations you will meet:

- **Re-input until valid**: the new loop goes right after the existing INPUT. With WHILE, the condition describes **bad** data (for example `WHILE Guests < 2 OR Guests > 8 DO`); inside the loop, tell the user what went wrong and take a fresh input, otherwise the loop never ends. The alternative is to put the prompt and INPUT inside REPEAT ... UNTIL <condition for good data>.
- **Count-controlled to condition-controlled** (or back): replace `FOR Count ← 1 TO n ... NEXT Count` with a counter set before the loop, `Count ← Count + 1` inside it, and a condition that stops after the last value: `WHILE Count <= n` or `UNTIL Count > n`. Going the other way, remove the manual counter lines.
- **Run for any number of items** instead of a fixed number: input how many first and use it as the loop's end value, or loop until the user enters a stop value.
- **Add a counter or total**: declare it, set it to 0 before the loop, update it inside (inside the IF for a count), output it after the loop.
- **Round a result**: `Price ← ROUND(Price, 2)` rounds to 2 decimal places.

4. Answer pre-release task questions (14 questions)

From 2021 to 2022 only: the first question (20 marks) asked about a program candidates had prepared in advance. The format has gone, but the skills return in other questions:

- **Name a constant**, give its value and say why it is a constant: for a sports-day program, `CONSTANT MaxEntrants ← 40`, because the limit is fixed and must not change while the program runs.
- **Name a variable**, its data type and use: `RaceTime : REAL`, holds one runner's time in seconds.
- **Describe an array**: its name, data type, size, an example of what it holds and its use: `EventName[1:12]` of `STRING`, holding event names such as "100 m", one per event.
- **Explain how a task works**: go through the steps in order and explain every statement you mention, not just list code.

5. Identify or match programming constructs (12 questions)

1. Learn the names and the one-line meaning of each: **sequence**, **selection**, **iteration**, **totalling**, **counting**, **input**, **output**, **assignment**, and the three loop kinds.
2. For "give the line number of ...", point at the line that **does** it:
 - IF or CASE OF is selection; FOR, WHILE or REPEAT starts iteration;
 - `X ← X + 1` is counting; `X ← X + Value` is totalling; any `←` is an assignment;
 - FOR is count-controlled, WHILE pre-condition, REPEAT ... UNTIL post-condition.

- For "describe two types of iteration", name each and say how it stops: a **FOR** loop repeats a set number of times; a **WHILE** loop tests its condition first and may never run; a **REPEAT** loop tests its condition at the end and always runs at least once.

A **FOR** loop header is iteration, not counting, even though it counts the passes.

6. Find and correct errors in given pseudocode (10 questions)

You are told how many errors there are. For each, give the **line number** and the **whole corrected line**.

- Read the description of what the code should do, then check each line against it.
- Check in turn: data types in **DECLARE**; starting values; loop limits (does it run the right number of times?); the operator in each calculation and condition (***** or **/**, **>** or **>=**); counting versus totalling; the variable named after **NEXT** and in each **OUTPUT**; and that every block ends with its own keyword (**ENDIF**, **ENDCASE**, **NEXT**, **ENDWHILE**, **UNTIL**).
- Dry-run it with a couple of simple values if you are unsure; a wrong answer shows you where to look.

7. Explain how to make a program maintainable, or suggest meaningful identifiers (10 questions)

- Give the syllabus techniques: **meaningful identifiers**, **comments**, **procedures and functions**; constants and indentation are also accepted.
- If asked for a reason or example, pair each technique with how it helps another programmer: comments explain what each section does; clear names show what each variable holds; procedures split the code into small parts that can be tested and reused.
- For "suggest meaningful identifiers", replace each single letter with a name describing its job, worked out from how it is used: a variable added to each time round is a total (**TotalRainfall**); one with 1 added is a counter (**WetDays**); one used inside square brackets is an index (**DayNumber**).

Only techniques that help a person read and change the code count here; anything about checking the data or making the program run correctly answers a different question.

8. Identify, match or describe data types (9 questions)

- Decide what the data **is**: whole number → integer; can have a decimal part → real; a single character → char; text or a code → string; one of two values → Boolean.
- To "describe" a type, say what it holds and give an example: "a char holds exactly one character, such as 'H'"; a string holds a sequence of characters, such as "Hello there".
- If each answer must be different, use each type once.
- Use programming data types. "Text" and "number" are not pseudocode data types.

9. Write string-handling statements (8 questions)

- Store** a string with its quotes: **Motto** ← "Learn by doing".
- Extract** part of it by counting positions, spaces included, and using **SUBSTRING(s, start, length)**.
- Change case** with **UCASE()** or **LCASE()**, and output or store the result.
- Length** with **LENGTH()**.
- If asked what is output, count characters carefully: in "Learn by doing", the space after **Learn** is position 6, so by is **SUBSTRING(Motto, 7, 2)**, and **LENGTH(Motto)** is 14.

Variations you will meet: process each character in a loop, find the position of a character, count characters of one kind, or write the extracted part to a file (see **File handling**).

10. Declare variables with a data type (6 questions)

Write one line each: **DECLARE Identifier : DATATYPE**, using the name given and the data type that matches the description ("holds a whole number" → **INTEGER**, "a flag" → **BOOLEAN**). Then, if asked, store a value of the right type: a char in single quotes, a string in double quotes.

11. Use or classify operators (6 questions)

- Evaluate:** work out **^**, **DIV** and **MOD** carefully. **DIV** throws away the fractional part; **MOD** gives what is left over.
- Classify:** arithmetic (**+** **-** ***** **/** **^** **DIV** **MOD**) does calculations; relational, also called comparison or Boolean operators (**=** **<>** **<** **<=** **>** **>=**), compare two values and give **TRUE** or **FALSE**; logical (**AND** **OR** **NOT**) combine or reverse conditions.
- State how each is used, with an example:** give a full pseudocode statement, for example **Change** ← **Paid** - **Cost** (arithmetic), **IF** **Speed** > 70 (relational), **IF** **Speed** > 70 **AND** **Zone** = 'S' (logical).

12. State the purpose of an algorithm or describe its processes (6 questions)

1. Dry-run it with simple data: write down each variable's value as the code changes it.
2. Look at the output: what does it tell the user about the input?
3. **Purpose:** one sentence saying what the algorithm achieves for the user ("it outputs the largest of the numbers typed in"), not a list of the keywords it uses.
4. **Processes:** describe each stage in the order it happens and what it does to the data: what is input and where it is stored, what the loop goes through, what is compared or changed, and what is output.

13. Describe variables and constants (5 questions)

- **Variable:** a named memory location whose value **can change** while the program runs. Use one for anything input or calculated, for example `Score`.
- **Constant:** a named memory location whose value is set once and **can't change** while the program runs. Use one for a fixed value such as `CONSTANT DaysInWeek ← 7`; it is easier to read and to update.
- Always include the phrase "while the program is running" (or "during execution"): that is the difference the mark is for.

14. Complete an incomplete algorithm (5 questions)

1. Read what the algorithm is meant to do and what each variable is for.
2. Fill each gap with the piece of the pattern that is missing: a starting value (`Count ← 0`), a loop limit, a condition, an update line (`Count ← Count + 1`), a calculation (`Mean ← Total / Count`), or a closing keyword (`NEXT Index, ENDWHILE`).
3. Check that the finished algorithm would give the right output for simple data.

Worked examples

The bold codes show where marks are earned, as in Cambridge mark schemes. The number is how many marks.

- **B1** mark for a correct result on its own, with no method needed.
-

Example 1

A farm packs eggs into boxes that each hold 12 eggs.

(a) Write a pseudocode statement to declare the box size as a constant, and give one reason why a constant is suitable. **[2]**

(b) Write pseudocode to:

- input the number of eggs collected, repeating the input until it is from 1 to 5000 inclusive
- output the number of full boxes and the number of eggs left over, with suitable messages. **[5]**

(c) State what is output when 200 eggs are input. **[1]**

(a) `CONSTANT BoxSize ← 12` **B1**. The number of eggs in a box never changes while the program runs, and if the farm changes box size the value only needs editing in one place. **B1**

(b)

```

DECLARE Eggs : INTEGER
DECLARE Boxes : INTEGER
DECLARE LeftOver : INTEGER
OUTPUT "How many eggs were collected today?"
INPUT Eggs
WHILE Eggs < 1 OR Eggs > 5000 DO
    OUTPUT "Please enter a number from 1 to 5000"
    INPUT Eggs
ENDWHILE
Boxes ← DIV(Eggs, BoxSize)
LeftOver ← MOD(Eggs, BoxSize)

```

OUTPUT "Full boxes: ", Boxes

OUTPUT "Eggs left over: ", LeftOver

- Prompt and input of the number of eggs. **B1**
- A loop with the correct condition for invalid data (< 1 OR > 5000). **B1**
- An error message and the input repeated **inside** the loop. **B1**
- Full boxes with DIV **and** eggs left over with MOD. **B1**
- Both results output with messages. **B1**

(c) Full boxes: 16, Eggs left over: 8, because $200 = 16 \times 12 + 8$. **B1**

Example 2

A school makes computer usernames from students' names.

(a) Write pseudocode that inputs a student's first name and family name, then outputs their username: the first three letters of the family name in upper case, then the first letter of the first name in lower case, then the number of characters in the family name. You may assume every family name has at least three letters. **[4]**

(b) State the username for Priya Ramanathan. **[1]**

(c) Write pseudocode that inputs a sentence and outputs the number of words in it. Words are separated by single spaces. **[4]**

(a)

OUTPUT "Enter the first name"

INPUT FirstName

OUTPUT "Enter the family name"

INPUT FamilyName

OUTPUT "Username: ", UCASE(SUBSTRING(FamilyName, 1, 3)), LCASE(SUBSTRING(FirstName, 1, 1)), LENGTH(FamilyName)

- Both names input into two variables. **B1**
- SUBSTRING(FamilyName, 1, 3) changed with UCASE. **B1**
- SUBSTRING(FirstName, 1, 1) changed with LCASE. **B1**
- LENGTH(FamilyName) and all three parts output together. **B1**

(b) RAMP10 ("Ramanathan" has 10 letters). **B1**

(c) Count the spaces; there is one word more than there are spaces.

OUTPUT "Type a sentence"

INPUT Sentence

Spaces $\leftarrow$ 0

FOR Position $\leftarrow$ 1 TO LENGTH(Sentence)

 IF SUBSTRING(Sentence, Position, 1) = " "

 THEN

 Spaces $\leftarrow$ Spaces + 1

 ENDIF

NEXT Position

OUTPUT "Number of words: ", Spaces + 1

- Counter set to 0 before the loop. **B1**
- A loop from 1 to LENGTH(Sentence). **B1**
- Each character taken with SUBSTRING(..., Position, 1) and compared with a space. **B1**
- Counter increased by 1 for a space, and Spaces + 1 output with a message. **B1**

For "Tides turn at dawn" the loop finds 3 spaces and outputs 4.

Example 3

This algorithm should input the number of cups of coffee sold in each of the 8 hours a café is open. It should output the total number of cups, the number of busy hours (hours with **more than** 30 cups sold) and the mean number of cups per hour, which may not be a whole number. There are four errors.

```

01 DECLARE Hour : INTEGER
02 DECLARE Sold : INTEGER
03 DECLARE Total : INTEGER
04 DECLARE Busy : INTEGER
05 Total ← 0
06 Busy ← 0
07 FOR Hour ← 1 TO 8
08     OUTPUT "Cups sold in hour ", Hour
09     INPUT Sold
10     Total ← Total + Sold
11     IF Sold >= 30
12         THEN
13             Busy ← Busy + Sold
14     ENDIF
15 NEXT Sold
16 OUTPUT "Total cups: ", Total
17 OUTPUT "Busy hours: ", Busy
18 OUTPUT "Mean cups per hour: ", DIV(Total, 8)

```

(a) Identify the line number of each error and write the corrected line. **[4]**

(b) Rewrite lines 07 to 15 of the corrected algorithm using a post-condition loop instead of a count-controlled loop. **[3]**

(c) The inputs are 25, 31, 30, 42, 18, 36, 29, 33. State the three outputs of the corrected algorithm. **[2]**

(a)

- Line 11: IF Sold > 30 (30 cups is not **more than** 30). **B1**
- Line 13: Busy ← Busy + 1 (a busy hour is counted, not totalled). **B1**
- Line 15: NEXT Hour (NEXT names the loop variable). **B1**
- Line 18: OUTPUT "Mean cups per hour: ", Total / 8 (DIV would throw away the fractional part). **B1**

(b)

```

Hour ← 1
REPEAT
    OUTPUT "Cups sold in hour ", Hour
    INPUT Sold
    Total ← Total + Sold
    IF Sold > 30
    THEN
        Busy ← Busy + 1
    ENDIF
    Hour ← Hour + 1
UNTIL Hour > 8

```

- Hour ← 1 before the loop. **B1**
- Hour ← Hour + 1 inside the loop. **B1**
- REPEAT ... UNTIL Hour > 8, so it runs exactly 8 times. **B1**

(c) Total cups: 244 and Busy hours: 4 (31, 42, 36 and 33) **B1**; Mean cups per hour: 30.5 ($244 \div 8$) **B1**. The algorithm as first written would output 172 for the busy hours (it adds $31 + 30 + 42 + 36 + 33$) and a mean of 30.

Example 4

Six classes run a charity stall for five days. The 1D array `ClassName[1:6]` holds the class names. The 2D array `Takings[1:6, 1:5]` is to store the amount, in whole dollars, each class takes on each day; the first index is the same as the class's index in `ClassName[ ]`. The 1D array `ClassTotal[1:6]` is to store each class's total.

Write a program that:

- inputs the amount taken by every class on every day, validating that each is from 0 to 500 inclusive
- calculates each class's total and stores it in `ClassTotal[ ]`

- outputs each class's name, total, and mean takings per day rounded to 2 decimal places
- counts and outputs the number of times a class took nothing on a day
- outputs the total raised by all six classes and the name of the class that raised the most (assume there is no tie).

You must use pseudocode or program code and add comments to explain how your code works. You do not need to declare any arrays or variables. All inputs and outputs must contain suitable messages. **[15]**

```
// store every day's takings, checking each is from 0 to 500
FOR Class ← 1 TO 6
  FOR Day ← 1 TO 5
    OUTPUT "Takings for ", ClassName[Class], " on day ", Day
    INPUT Amount
    WHILE Amount < 0 OR Amount > 500 DO
      OUTPUT "Amount must be from 0 to 500, please enter it again"
      INPUT Amount
    ENDWHILE
    Takings[Class, Day] ← Amount
  NEXT Day
NEXT Class

// totals for each class, days with no takings, grand total and best class
EmptyDays ← 0
GrandTotal ← 0
Best ← 1 // index of the best class so far
FOR Class ← 1 TO 6
  ClassTotal[Class] ← 0 // reset for each class
  FOR Day ← 1 TO 5
    ClassTotal[Class] ← ClassTotal[Class] + Takings[Class, Day]
    IF Takings[Class, Day] = 0
      THEN
        EmptyDays ← EmptyDays + 1
      ENDIF
  NEXT Day
  GrandTotal ← GrandTotal + ClassTotal[Class]
  IF ClassTotal[Class] > ClassTotal[Best]
    THEN
      Best ← Class
  ENDIF
  // name, total and mean per day for this class
  OUTPUT ClassName[Class], " raised $", ClassTotal[Class]
  OUTPUT "Mean per day: $", ROUND(ClassTotal[Class] / 5, 2)
NEXT Class

// results for the whole stall
OUTPUT "Days with no takings: ", EmptyDays
OUTPUT "Total raised by all classes: $", GrandTotal
OUTPUT "Most raised by: ", ClassName[Best]
```

How it earns the marks:

- **Techniques and data structures (up to 9):** every bullet is met: nested loops over classes and days; input with a validation loop and a message; totalling per class (reset inside the outer loop) and overall; counting zero days inside an IF; the mean after each class's inner loop, rounded; the best class found by comparing totals and keeping its index; every output with a message. It stores the data in the three arrays named in the question.
- **Clarity and accuracy (up to 6):** a comment for each section, meaningful names (EmptyDays, GrandTotal, Best), and the sections in a logical order.

A dry run with just the first two classes (loops to 2): 7A takes 40, 0, 55, 120, 35 and 7B takes 90, 85, 0, 0, 110. 7A's total is 250 (mean 50.00) and 7B's is 285 (mean 57.00); there are 3 days with no takings, the grand total is 535 and 7B raised the most.

Common mistakes

- **Names a real language would reject.** `Total Cost` (a space) or `No.Of.Pupils` (full stops) can't be identifiers, and switching between `Total` and `TotalCost` in one answer looks like two different variables. Choose one single-word name and use it every time.
- **Describing a constant as "a value that doesn't change".** Examiners want the key idea that it stays the same **while the program is running**; without it, the description is incomplete.
- **Picking a type that can't hold the data.** A price such as 2.49 needs `REAL`, not `INTEGER`, which would lose the pence; and "number" or "text" are not answers at all, because the pseudocode types are `INTEGER`, `REAL`, `CHAR`, `STRING` and `BOOLEAN`.
- **Swapping pre- and post-condition loops.** A `WHILE` loop tests first and may never run; a `REPEAT` loop tests last and always runs once. Calling a `FOR` loop "counting" is another common slip: it is iteration.
- **Ending a `FOR` loop with `ENDFOR`,** or closing blocks with the wrong keyword. In pseudocode it is `NEXT` plus the variable.
- **Forgetting to set a total or counter to 0** before the loop, which examiners mention nearly every year.
- **Not knowing `UCASE`, `LCASE` and `SUBSTRING`,** or writing a programming language's version (such as `.upper ( )`) when pseudocode is asked for.
- **Skipping spaces and punctuation** when working out a length or a position: every character counts, so `LENGTH( "Hi, Jo" )` is 6.
- **Missing quotes** around string values, or using double quotes for a single `CHAR`.
- **Confusing `DIV` and `MOD`:** `DIV` is how many whole times it goes, `MOD` is what's left over. Knowing this in words is not enough; practise writing them as statements, such as `Weeks ← DIV( Days, 7 )`.
- **Answering "explain" with code alone.** When a question says "explain" or "any code must be fully explained", say in words what each change does and where it goes.
- **Giving general good habits instead of maintainability techniques.** A maintainability question is only about making code easy for another person to read and change. Stick to the syllabus techniques: clear names, comments, and procedures and functions (constants and indentation also help). Answer it as a stand-alone question, even when it follows a question about code.
- **Declaring everything in the 15-mark program** when the question says you don't need to. It wastes time and earns nothing.

How to get the marks

- **Short algorithms are marked point by point.** Each bullet in the question is usually one or two marks: make each one visible on its own line.
- **Messages count.** When a question says "suitable messages", every input needs a prompt and every output needs text with it.
- **Pseudocode means pseudocode.** Only the 15-mark program may be in a programming language. Elsewhere, write Cambridge-style pseudocode; small syntax slips don't lose marks, but wrong keywords or missing structure do.
- **Error questions:** one mark per error, for the line number **and** the fully corrected line. Write the whole statement, not "change `>=`".
- **"State"** needs a short answer with no explanation (a data type, a line number, an output). **"Describe"** needs the features in a sentence or two. **"Explain"** needs reasons: say why or how.
- **Examples:** when asked for an example, give a real value or a pseudocode statement in the right form (`'A'` for a char, `"Hi there"` for a string, `IF Age >= 18` for a relational operator).
- **The 15-mark program:** the techniques mark depends on covering **every** requirement with a suitable technique and storing the data in the named arrays; the other mark depends on comments throughout, meaningful identifiers, and a solution that works in a logical order.
- **Loops:** say which kind you have used when asked, and make sure the condition stops the loop at the right time; an off-by-one limit loses the loop mark.

Quick check

1. Evaluate: (i) `DIV( 47, 6 )` (ii) `MOD( 47, 6 )` (iii) `3 ^ 4` (iv) `47 / 4`.
2. State the most suitable data type for: (i) the number of pupils in a school; (ii) the price of a sandwich, 3.75; (iii) a grade such as `'B'`; (iv) a postcode such as `"SW1A 2AA"`; (v) whether a light is switched on.
3. `Word ← "Kilimanjaro"`. State the value of: (i) `LENGTH( Word )` (ii) `SUBSTRING( Word, 5, 4 )` (iii) `UCASE( SUBSTRING( Word, 1, 4 ) )`.
4. `Count` is 10 at the start of each loop below. How many times does each body run? (i) `WHILE Count < 10 DO ... ENDWHILE`, where the body adds 1 to `Count`; (ii) `REPEAT ... UNTIL Count >= 10`, where the body adds 1 to `Count`; (iii) `FOR Step ← 3 TO 12 STEP 3 ... NEXT Step`.
5. Give the line number of the statement that is (i) selection (ii) counting (iii) totalling (iv) a post-condition loop, in:

```
01 Stamps ← 0
02 Bill ← 0
03 REPEAT
04     INPUT Cost
```

```
05    Bill ← Bill + Cost
06    IF Cost > 5
07        THEN
08            Stamps ← Stamps + 1
09        ENDIF
10 UNTIL Cost = 0
```

Answers

1. (i) 7 (ii) 5, since $47 = 7 \times 6 + 5$ (iii) 81 (iv) 11.75.
2. (i) integer (ii) real (iii) char (iv) string (v) Boolean.
3. The positions are K1 i2 l3 i4 m5 a6 n7 j8 a9 r10 o11. (i) 11 (ii) "manj" (positions 5 to 8) (iii) "KILL".
4. (i) 0 times: the condition is false at the first test. (ii) Once: the body runs before the test, then Count is 11 and the loop stops. (iii) 4 times, for 3, 6, 9 and 12.
5. (i) line 06 (ii) line 08 (iii) line 05 (iv) line 03 (the loop begins at REPEAT and its condition is tested at line 10).

Past questions to try

- [0478/21/M/J/24 Q5](#): input a number and use a CASE statement with OTHERWISE.
- [0478/23/M/J/24 Q7](#): store a string, extract two words with SUBSTRING and output it in upper case.
- [0478/22/M/J/24 Q5](#): describe arithmetic, Boolean and logical operators with a pseudocode example of each.
- [0478/21/M/J/25 Q3](#): rewrite a count-controlled loop as a condition-controlled loop.