

Data types and structures AS

Read online at pastopic.com/cambridge/computer-science-9618/notes/data-types-and-structures

Last checked 5 October 2026

Read first: [Algorithm design and problem-solving](#)

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

What you need to know

This topic is examined in Paper 2, usually inside longer programming questions. By the end of it you should be able to:

1. **Data types:** choose the right type for a value or variable from INTEGER, REAL, CHAR, STRING, BOOLEAN and DATE. Pseudocode also uses ARRAY and FILE.
2. **Records:** explain what a record structure is for (holding several items of **different** data types under one identifier), define one in pseudocode, and read data from it and save data into it.
3. **Array terms:** use the words **index**, **lower bound** and **upper bound**, and **dimension**.
4. **Choosing an array:** decide whether a task needs a 1D or a 2D array.
5. **Array pseudocode:** declare 1D and 2D arrays, and process them, including a **linear search** and a **bubble sort**.
6. **Files:** explain why files are needed, and write pseudocode that reads, writes and appends text files of one or more lines.
7. **Abstract data types (ADTs):** explain that an ADT is a collection of data together with a set of operations on that data.
8. **Stack, queue and linked list:** describe the key features of each, justify which one to use in a given situation, add, edit and delete data in them (by hand, on a diagram), and describe how each can be built from arrays. You do not have to write full pseudocode for these structures, though completing a few lines of it is sometimes asked.

Understand it

Data types

Every variable holds one type of data. The type decides what values it can hold and what you can do with it.

Type	Holds	Example values
INTEGER	whole numbers, positive or negative	0, 250, -17
REAL	numbers that can have a fractional part	5.42, -0.5, 12.0
CHAR	exactly one character, in single quotes	'Q', '7', ' '
STRING	any number of characters (even none), in double quotes	"Kiwi", "07700", ""
BOOLEAN	TRUE or FALSE only	FALSE
DATE	a date, written dd/mm/yyyy without quotes	21/10/2026

How to choose:

- **Money, averages, percentages and measurements** are REAL, even if today's value happens to be whole.
- **Counts and positions** (how many, which index) are INTEGER.

- **Codes and phone numbers** are STRING, even when they are all digits: you never do arithmetic on them, and a leading 0 must not be lost.
- A value in **single quotes** is a CHAR; in **double quotes** it is a STRING, so "18/04/2021" is a STRING but 18/04/2021 is a DATE.
- Something that is only ever "yes or no" (paid, confirmed, returned) is BOOLEAN.
- **An expression has the type of its result.** LENGTH(S) and INT(X) give INTEGER; MID, LEFT, RIGHT and NUM_TO_STR give STRING; IS_NUM(S) and NOT Done give BOOLEAN; Distance / 4 gives REAL; DAY, MONTH, YEAR and DAYINDEX give INTEGER; SETDATE and TODAY give DATE. Mixing types without converting, such as Total + "3.2", is an error.

Records

A **record** groups several related items, called **fields**, under **one identifier**. The fields can have **different data types**, which an array cannot do. You first define the record **type** with TYPE ... ENDTYPE, then declare variables of that type, then use **dot notation** to reach a field.

```
TYPE Booking
```

```
    DECLARE BookingRef : STRING
    DECLARE Seats       : INTEGER
    DECLARE Row         : CHAR
    DECLARE ShowDate    : DATE
    DECLARE Price       : REAL
    DECLARE Paid        : BOOLEAN
```

```
ENDTYPE
```

```
DECLARE ThisBooking : Booking
```

```
ThisBooking.Seats ← 3           // save data into a field
OUTPUT ThisBooking.BookingRef  // read data from a field
```

An **array of records** stores many records of the same type: DECLARE Bookings : ARRAY[1:400] OF Booking. Then Bookings[17].Paid ← TRUE sets one field of the 17th record. Benefits: all the data about one item stays together under one identifier; a loop can go through the records by index; and the code is shorter and easier to follow than many parallel arrays. Unused records are usually marked by a special value in one field, such as an empty string "" in the ID.

Arrays

An **array** is a set of elements that all have the **same data type**, under one identifier. Each element is reached by its **index**. The smallest index is the **lower bound**, and the largest is the **upper bound**.

```
DECLARE Temps : ARRAY[1:30] OF REAL           // 1D: 30 elements, Temps[1] to Temps[30]
DECLARE Seats : ARRAY[0:19, 1:8] OF CHAR      // 2D: 20 rows (0 to 19), 8 columns
Temps[4] ← 21.5
Seats[0, 3] ← 'X'                             // row 0, column 3
```

The number of elements in a dimension is upper bound – lower bound + 1, so Seats has $20 \times 8 = 160$ elements. To declare an array you need its identifier, its bounds (so its size) and its data type.

1D or 2D? Use a 1D array for a list of items (30 daily temperatures). Use a 2D array when each item needs two indexes, a row and a column: a grid of seats, a timetable of days × periods, a table with several numbers of the **same type** for each item (all 2D elements must share one type; for mixed types use an array of records).

Why use an array instead of 30 separate variables? One identifier for all the data, a loop can process every element using the index, and the size is easy to change.

Linear search

A **linear search** looks at each element in turn, from the first, until it finds the item or reaches the end. It works on unsorted data. A good search stops as soon as the item is found, so it uses a conditional loop:

```
Index ← 1
Found ← FALSE
WHILE Index <= 50 AND NOT Found
    IF Names[Index] = SearchName THEN
        Found ← TRUE
    ELSE
        Index ← Index + 1
    ENDIF
ENDWHILE
IF Found THEN
    OUTPUT "Found at position ", Index
ELSE
    OUTPUT "Not found"
ENDIF
```

Bubble sort

A **bubble sort** goes through the array comparing each element with the next one, and **swaps** them if they are in the wrong order. After the first pass the largest value has "bubbled" to the end, so each later pass can stop one place earlier. If a whole pass makes no swaps, the array is sorted and the sort can stop. Swapping needs a temporary variable, otherwise one value is overwritten.

```
PROCEDURE SortMarks()
    DECLARE Index, Boundary, Temp : INTEGER
    DECLARE Swapped : BOOLEAN
    Boundary ← 99                                // compare pairs 1–2 up to 99–100
    REPEAT
        Swapped ← FALSE
        FOR Index ← 1 TO Boundary
            IF Marks[Index] > Marks[Index + 1] THEN
                Temp ← Marks[Index]
                Marks[Index] ← Marks[Index + 1]
                Marks[Index + 1] ← Temp
                Swapped ← TRUE
            ENDIF
        NEXT Index
        Boundary ← Boundary - 1
    UNTIL NOT Swapped OR Boundary = 0
ENDPROCEDURE
```

For the values 7, 3, 9, 1, 5 the first pass gives 3, 7, 1, 5, 9 (three swaps, 9 is now in place). The second gives 3, 1, 5, 7, 9, the third 1, 3, 5, 7, 9, and the fourth makes no swaps, so the sort stops. Use > for ascending order and < for descending.

Text files

Variables and arrays are held in main memory (RAM), which is cleared when the program ends. A **file** is stored in secondary storage, so:

- the data is **kept** after the program stops and can be loaded next time, with no need to type it in again;
- other programs can use the same data;
- the amount of data is not fixed by the size of an array.

A **text file** stores lines of characters. Every line is a **STRING**, so a number, a **BOOLEAN** or a **DATE** must be **converted to a string** before it is written (**NUM_TO_STR**) and converted back after it is read (**STR_TO_NUM**). The file statements are:

```
OPENFILE "Scores.txt" FOR READ           // or WRITE, or APPEND
READFILE "Scores.txt", Line              // reads the next line into a STRING variable
WRITEFILE "Scores.txt", Line             // writes a STRING as a new line
EOF("Scores.txt")                        // TRUE when there are no more lines to read
CLOSEFILE "Scores.txt"
```

The three **modes**: **READ** to read lines from an existing file; **WRITE** creates a new file, **erasing** any file with that name; **APPEND** adds new lines to the **end** of an existing file. If the file name is in a variable or parameter, write the variable without quotes: `OPENFILE FileName FOR READ`.

Reading a whole file of unknown length:

```
OPENFILE "Scores.txt" FOR READ
WHILE NOT EOF("Scores.txt")
    READFILE "Scores.txt", Line
    Total ← Total + STR_TO_NUM(Line)
ENDWHILE
CLOSEFILE "Scores.txt"
```

Storing a record in a text file. There are two common designs:

- **One record per line**, with the fields joined by a **separator** character that never appears in the data, such as "KX4410#Tara Singh#TRUE". When the line is read back, the program splits it at each separator.
- **One field per line**, so each record takes several consecutive lines. Reading back is simpler (no splitting), but there are more file reads and writes per record.

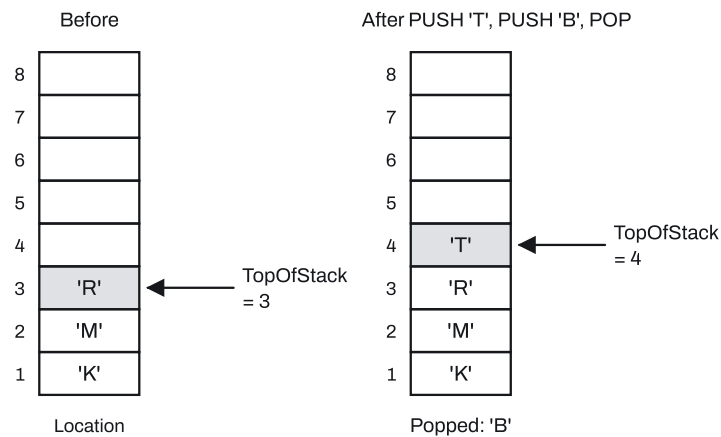
If a field may contain **any** character, no separator is safe. Instead store each such field's **length** in front of it as a fixed-width number, such as "07" for a 7-character field, or give every field a **fixed length**, padded with spaces. Then **MID** can cut each field out by position.

Abstract data types

An **abstract data type (ADT)** is a collection of data together with a set of operations on that data. It is "abstract" because it says what you can do (add an item, remove an item) and not how the data is stored. A program builds an ADT from what the language does have, usually arrays and integer variables used as **pointers** (a pointer holds the index of an element).

Stack

A **stack** is a **last in, first out (LIFO)** structure: items are added and removed at the same end, the **top**. Adding is a **push**; removing is a **pop**. One pointer, often called `TopOfStack`, gives the index of the top item.



Built from a 1D array `Stack [ 1 : 8 ]` and an integer `TopOfStack` that points to the last item added (0 means empty):

- **Push:** if `TopOfStack = 8`, the stack is full and the item cannot be added. Otherwise add 1 to `TopOfStack`, then store the item at `Stack [ TopOfStack ]`.
- **Pop:** if `TopOfStack = 0`, the stack is empty and nothing can be removed. Otherwise read `Stack [ TopOfStack ]`, then subtract 1 from `TopOfStack`.

```

FUNCTION Pop() RETURNS STRING
  DECLARE Item : STRING
  IF TopOfStack = 0 THEN
    RETURN "" // stack empty
  ENDIF
  Item ← Stack[TopOfStack]
  TopOfStack ← TopOfStack - 1
  RETURN Item
ENDFUNCTION

```

The array elements don't need setting to anything first: only the elements from the bottom up to the pointer count as being on the stack, and a push always writes before anything is read. A popped value is not erased either; it is simply no longer below the pointer, and a later push overwrites it. On diagrams, show popped locations as empty.

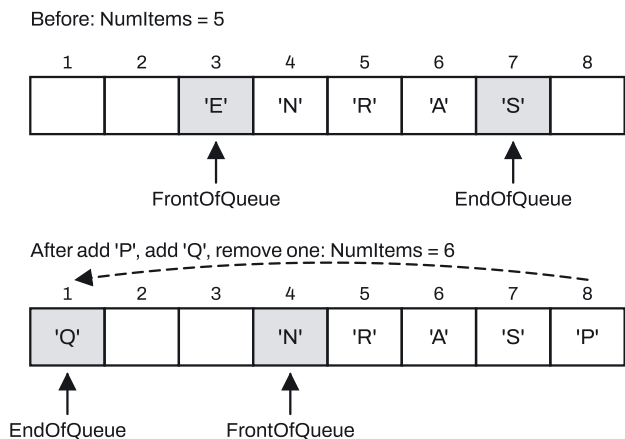
Read each question's rules carefully: the pointer might point to the **next free** location instead (then an empty stack has the pointer at the bottom index, and a push stores first and increments after), the stack might grow downwards through memory, and a second variable may count the items.

Uses: undo in an editor (the most recent action is undone first), the return addresses of procedure calls, the back button of a browser, and **reversing** the order of items.

Queue

A **queue** is a **first in, first out (FIFO)** structure: items join at the **end** (rear) and leave from the **front**. It needs two pointers, `FrontOfQueue` (the next item to be removed) and `EndOfQueue` (the last item added), and usually a count `NumItems`.

Built from an array `Queue [ 1 : 8 ]`, the pointers would run off the end after 8 items even though removed items leave free space at the start. So the queue is made **circular**: when a pointer passes the upper bound it **wraps around** to the lower bound.



- **Start:** FrontOfQueue $\leftarrow$ 1, EndOfQueue $\leftarrow$ 0, NumItems $\leftarrow$ 0.
- **Add (enqueue):** if NumItems = 8, the queue is full. Otherwise: if EndOfQueue = 8 set it to 1, else add 1 to it; store the item at Queue[EndOfQueue]; add 1 to NumItems.
- **Remove (dequeue):** if NumItems = 0, the queue is empty. Otherwise: read Queue[FrontOfQueue]; if FrontOfQueue = 8 set it to 1, else add 1 to it; subtract 1 from NumItems.

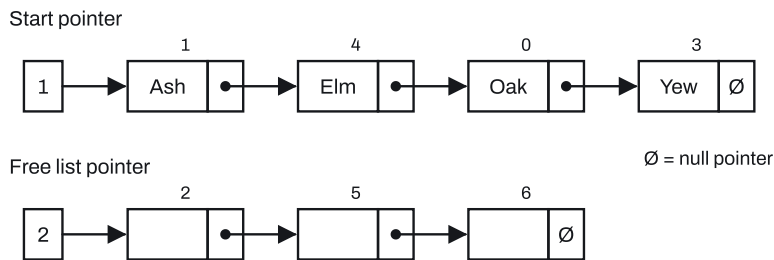
The count is what tells "full" from "empty", because the two pointers look the same in both cases (the end pointer sits just behind the front pointer). If FrontOfQueue = EndOfQueue, the queue holds exactly one item.

Uses: print jobs waiting for a printer, keys pressed waiting to be processed (a keyboard buffer), tasks waiting for the processor: anywhere items must be dealt with **in the order they arrive**.

Reversing a stack with a queue: pop every item off the stack and add each to the queue (the top item goes in first), then remove every item from the queue and push each back on. The item that was on top is now at the bottom.

Linked list

A **linked list** keeps items in order without storing them next to each other. Each item is a **node** holding the **data** and a **pointer** to the next node. A **start pointer** gives the first node, and the last node holds a **null pointer**, a value that can never be a real index (−1 if the indexes start at 0; 0 if they start at 1). Unused nodes form a second list, the **free list**, with its own **free list pointer**.



Built from arrays, with a Data array, a Pointer array and two integer variables (or one array of records with a Data field and a Pointer field). Here −1 is the null pointer:

Index	Data	Pointer
0	"Oak"	3
1	"Ash"	4

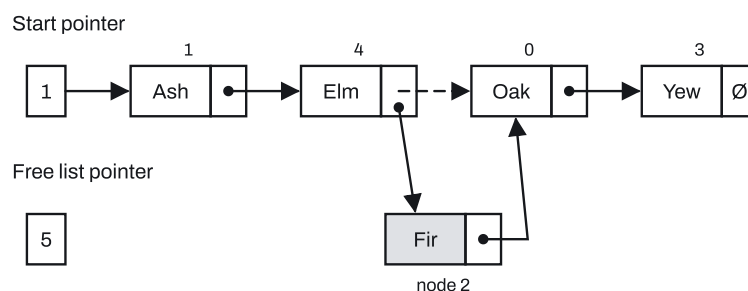
Index	Data	Pointer
2		5
3	"Yew"	-1
4	"Elm"	0
5		6
6		-1

`StartPointer = 1` and `FreeListPointer = 2`. Follow the pointers from 1: Ash (1) → Elm (4) → Oak (0) → Yew (3) → -1, the end. The order of the list comes from the pointers, not from where the items sit in the array. The free list is 2 → 5 → 6 → -1.

To **output the list in order**, set a variable to the start pointer; while it is not the null pointer, output the data at that index, then set the variable to the pointer stored at that index. Going down the array from index 0 gives the wrong order.

Adding "Fir" in alphabetical order:

1. Take the first free node, index 2, and put "Fir" in its data.
2. Move the free list pointer on to that node's pointer: `FreeListPointer = 5`.
3. Follow the list from the start to find the node that should come before Fir: Elm (4), because Elm < Fir < Oak.
4. Set the new node's pointer to Elm's old pointer (0, which is Oak): `Pointer[2] = 0`.
5. Set Elm's pointer to the new node: `Pointer[4] = 2`.



Steps 4 and 5 must be in that order: if Elm's pointer changed first, the link to Oak would be lost. If the new item goes at the very start, its pointer is the old start pointer and the start pointer changes to the new node.

Deleting "Ash" (the first node): set the start pointer to Ash's pointer (4), so the list now starts at Elm. Then return node 1 to the free list: its pointer becomes the old free list pointer and the free list pointer becomes 1. To delete a node in the middle, set the **previous** node's pointer to the deleted node's pointer, so the list skips it. **Editing** a node's data is just overwriting its `Data` (then move it if the order would change).

Why use a linked list? Inserting and deleting only change a few pointers: no items have to be shifted along, as they would in a sorted array. The cost is extra memory for the pointers, and no direct access: to reach the 50th item you must follow 49 pointers.

Key facts and formulas

There is no formula list for 9618. The Paper 2 insert defines the built-in functions (`MID`, `LENGTH`, `NUM_TO_STR` and so on), but the statements and facts below are not given: learn them.

Formula	Status
Data types: INTEGER (whole number), REAL (can have a fraction), CHAR (one character, 'A'), STRING ("text"), BOOLEAN (TRUE/FALSE), DATE (21/10/2026)	Learn it
Record type: <code>TYPE Name ... DECLARE Field : Type ... ENDTYPE</code> ; a field is reached with dot notation, <code>Var.Field</code> or <code>Array[Index].Field</code>	Learn it
1D array: <code>DECLARE Name : ARRAY[Lower:Upper] OF Type</code>	Learn it
2D array: <code>DECLARE Name : ARRAY[L1:U1, L2:U2] OF Type</code> ; an element is <code>Name[Row, Column]</code>	Learn it
Elements in one dimension of an array = upper bound – lower bound + 1; a 2D array has rows × columns elements	Learn it
Text files: <code>OPENFILE File FOR READ, WRITE or APPEND; READFILE File, Var; WRITEFILE File, Data; EOF(File); CLOSEFILE File</code>	Learn it
WRITE mode creates the file and erases what was there; APPEND adds to the end; READ only reads	Learn it
Bubble sort on n items: $n - 1$ passes always sort it; a sort that stops only after a pass with no swaps may need one more pass to confirm the order	Learn it
ADT: a collection of data and a set of operations on that data	Learn it
Stack: last in, first out (LIFO), one pointer to the top; queue: first in, first out (FIFO), front and end pointers	Learn it
Circular queue: when a pointer passes the upper bound it wraps around to the lower bound	Learn it
Linked list: nodes of data + pointer, a start pointer, a free list pointer, and a null pointer that can never be an index	Learn it

How to do it

In the Paper 2 questions we have tagged for this topic (112 questions, 2021–2025; 9 of them are the same question set on two papers), the types came up this often. 38 of the 112 questions mix types, so the counts add up to more than 112.

Question type	Questions
Write pseudocode or a flowchart that loops through a 1D or 2D array	27
Write or describe an algorithm that reads, writes or appends a text file	27
Design how data is kept in a text file	21
Choose the data type for a variable or value	20
Declare a record type and an array of records	14
Use array terms and declare 1D and 2D arrays	10
Queue: features, adding and removing, circular queue in an array	9
Linked list: set pointers, add or delete a node, build it from arrays	9

Question type	Questions
Stack: state after PUSH and POP, pointer values, Push and Pop	8
Search or update an array of records	8
Bubble sort	5

1. Write pseudocode or a flowchart that loops through a 1D or 2D array (27 questions)

1. **Declare** every local variable, including the index, with its type.
2. **Initialise** the index, counters and totals before the loop ($\text{Index} \leftarrow 1$, $\text{Count} \leftarrow 0$).
3. **Pick the loop.** A FOR loop to visit every element ($\text{FOR Index} \leftarrow 1 \text{ TO } 30$: the exact number of elements). A conditional loop (WHILE or REPEAT) when it must stop early: an item is found, a rogue value is input, or the array is full.
4. **Inside the loop**, use $\text{Array}[\text{Index}]$: test it, add it, count it, or store into it; then move the index on (a FOR loop does this itself).
5. **After the loop**, output or return the result.

Variations you will meet:

- **Fill an array from input** until a rogue value or until it is full: two conditions joined by AND in a WHILE (keep going while not the rogue value **and** $\text{Index} \leq 20$), and never store the rogue value.
- **Unused elements** marked by a value such as "", -1 or 999: test for it and skip those elements.
- **Largest or smallest:** start $\text{Max} \leftarrow \text{Data}[1]$, loop from 2, replace when bigger; keep the index too if the position is asked.
- **Fill a lookup array:** a BOOLEAN array whose index is a possible value (a total from 0 to 26) and whose element says whether that value is valid. Declare it with the bounds of the possible values, set every element to FALSE in a loop, then set the valid ones to TRUE. Nested FOR loops with STEP produce every combination: $\text{FOR A} \leftarrow 0 \text{ TO } 12 \text{ STEP } 4$ inside $\text{FOR B} \leftarrow 0 \text{ TO } 14 \text{ STEP } 7$ sets $\text{Valid}[\text{A} + \text{B}] \leftarrow \text{TRUE}$. To check a value: if it is outside the bounds, it is invalid; otherwise return the element at that index.
- **2D arrays:** a FOR loop over rows with a FOR loop over columns inside it, using $\text{Grid}[\text{Row}, \text{Col}]$. To search one row, fix the row and loop through the columns (from the right, $\text{FOR Col} \leftarrow 40 \text{ TO } 1 \text{ STEP } -1$ for 40 columns, to find the last one).
- **Output in reverse:** $\text{FOR Index} \leftarrow 100 \text{ TO } 1 \text{ STEP } -1$.
- **"Efficient"** means a conditional loop that stops once the job is done.
- **Flowcharts:** the same parts as boxes: initialise, input in the loop, test for the rogue value, test for full, store and increment, output after the loop.

2. Write or describe an algorithm that reads, writes or appends a text file (27 questions)

1. **Open** the file in the right mode: READ to read, WRITE for a new file, APPEND to add to an existing one.
2. **Loop** WHILE NOT EOF (File) when reading a file of unknown length (or a FOR loop if the number of lines is known).
3. **Read a line** into a STRING variable; **convert** it with STR_TO_NUM before any arithmetic or numeric comparison.
4. **Process** it: test it, total it, store it in an array, or join it to other strings.
5. **Write** with WRITEFILE, converting non-strings first.
6. **Close** every file after the loop.

When the question says "do not use pseudocode", write the same steps in words: "open Sales.txt in read mode", "repeat until the end of the file", "convert the line to a number", "if it is greater than 500, output the week number", "close the file".

Variations you will meet:

- **Copy from one file to another:** open the old file for READ and the new one for WRITE; read three lines into three variables, join them with a separator, write the new line; repeat until the end of the old file; close both.
- **Keep the last three lines:** each time a line is read, move LineY into LineX, LineZ into LineY and the new line into LineZ (that order, so nothing is lost). After the loop the three variables hold the last three lines.
- **Copy an array to a file** (skipping unused elements) or a file into an array (stopping at the end of the file **or** when the array is full).
- **Stop at a terminator line** or after a given number of lines: add the second condition to the loop.
- **Many files with numbered names:** build each name with &, such as `FileName & "." & Suffix`, and open each one for WRITE.

3. Design how data is kept in a text file (21 questions)

These are short explanations, 1 to 3 marks each.

1. **Why a file:** the data is kept after the program ends (it is non-volatile), so it can be reloaded instead of typed in again; other programs can read it; there is no fixed size limit as there is with an array.
2. **What must change before writing:** every non-string item (INTEGER, REAL, BOOLEAN, DATE) must be converted to a string.
3. **Storing a record:** either one line per record with a **separator** between fields, or one field per line.
 - A good separator is a character that **never appears in the data** (so the line can be split correctly).
 - One field per line: **benefit**, no joining or splitting is needed; **drawback**, more lines to read and write for each record.
4. **Fields that can contain any character, of varying length:** a separator could appear inside the data, so store each field's length as a fixed number of digits before it (or use fixed-length fields). The reading program takes the length, then cuts out exactly that many characters with MID.
5. **Extracting a field from a line:** LEFT, MID and RIGHT by position (for fixed-length fields), or a loop that finds the separator. Then convert: STR_TO_NUM, or SETDATE (Day, Month, Year) for a date stored as "DDMMYYYY".
6. **A meaningful file name** each time a program runs: a fixed word plus the date, or plus the next number in a sequence ("Archive" & ...).
7. **Archiving records** before marking them unused: open a file in APPEND (or WRITE) mode and write each record, as one line with separators or as one field per line.

4. Choose the data type for a variable or value (20 questions)

1. **Read the example value or the description**, not just the name.
2. Apply the rules from "Understand it": fraction possible → REAL; count → INTEGER; one character in single quotes → CHAR; text or digit codes → STRING; yes/no → BOOLEAN; a date → DATE.
3. For an **expression**, give the type of its **result**: LEFT(PostCode, 3) is STRING, Age >= 18 is BOOLEAN, Price * 1.15 is REAL, LENGTH(S) + 1 is INTEGER.
4. One mark per row: write the type in capitals, exactly as the pseudocode names it.

Variation: evaluate an expression. A type mismatch, such as adding a number and a string, gives ERROR.

5. Declare a record type and an array of records (14 questions)

1. Write `TYPE RecordName`.

2. One `DECLARE Field : TYPE` per field, choosing each type from the example value and comment (a value used as an array index is `INTEGER`; a letter is `CHAR`; a value that can be 13.5 is `REAL`).
3. Finish with `ENDTYPE`.
4. Declare the array: `DECLARE Items : ARRAY[1:2000] OF RecordName`, with the bounds from the question **and** the record type name as its type.

Variations you will meet:

- **Benefits of an array of records:** items of different data types are held under a single identifier; records can be processed in a loop using the index (or one record reached directly by its index); less code that is easier to understand, test and maintain.
- **Mark an unused record:** set its key field to a value that can't be real data, such as " " for an ID, or a `BOOLEAN InUse` field set to `FALSE`. An unused optional field (a third club) can hold a value outside its range, such as 0.
- **Why not a 2D array:** all elements of an array share one data type, but the fields have different types.
- **Assign or test a field:** `Item[ThisIndex].Weight >= Min AND Item[ThisIndex].Weight <= Max`.

6. Use array terms and declare 1D and 2D arrays (10 questions)

1. **Dimensions:** count the indexes inside the brackets (`Rainfall[Day, Town]` is 2).
2. **Number of elements:** for each dimension, upper – lower + 1, then multiply: `[1:31, 0:5]` gives $31 \times 6 = 186$.
3. **Declaration:** `DECLARE Rainfall : ARRAY[1:31, 0:5] OF REAL`. A mark for the bounds and a mark for the rest (identifier, `ARRAY`, `OF`, type).
4. To declare an array you need its **data type** as well as its size.
5. **Spot the error:** an index outside the bounds (index 0 in an `ARRAY[1:50]`), or a value of the wrong type.

7. Queue: features, adding and removing, circular queue in an array (9 questions)

1. **Features:** `FIFO`; a front pointer and an end pointer; usually circular so space at the start is reused; a count of the items.
2. **Draw the state after operations:** do them one at a time, wrapping each pointer from the upper bound to the lower bound, and keep `NumItems` up to date. Show both pointers and the count.
3. **Describe adding:** check the queue is not full (`NumItems = size`); move the end pointer on, wrapping round if it was at the upper bound; store the item at the end pointer; add 1 to `NumItems`.
4. **Describe removing:** check the queue is not empty; read the item at the front pointer; move the front pointer on, wrapping round; subtract 1 from `NumItems`.
5. **Declare and initialise:** the array with its bounds and type, the two pointers and the count as `INTEGER`; front at the lower bound, count 0.
6. **Justify a queue** when items must be handled in the order they arrive.

8. Linked list: set pointers, add or delete a node, build it from arrays (9 questions)

1. **Fill the pointer column:** put the data in order (alphabetical or numerical), then give each node the index of the node after it; the last gets the null pointer. The start pointer is the index of the first item.
2. **Null pointer:** a value that can never be an index: `-1` for an array from 0, or 0 for an array from 1. "It points to nothing" is too vague; say it marks the **end of the list**.
3. **Add a node:** take the node at the free list pointer and store the data; move the free list pointer to that node's pointer; find the node it must follow; copy that node's pointer into the new node; point that node at the new node.

4. **Delete a node:** find it and the node before it; set the previous node's pointer to the deleted node's pointer (or move the start pointer if it was first); link the freed node to the front of the free list.
5. **Initialise:** every node points to the next one ($\text{Pointer}[i] \leftarrow i + 1$), the last holds the null pointer, the start pointer is null (the list is empty) and the free list pointer is the first index.
6. **Describe building it from arrays:** two integer variables (start pointer and free list pointer) that hold indexes; a 1D array for the data (of the data's type); a 1D INTEGER array for the pointers. Or a single array of records, each with a data field and a pointer field.
7. **Output in order:** follow the pointers from the start pointer until the null pointer, outputting each data item.

9. Stack: state after PUSH and POP, pointer values, Push and Pop (8 questions)

1. **Read the rules first:** does the pointer point to the last item or the next free space? Which way does the stack grow? What does "empty" look like?
2. **Trace the operations one at a time:** a PUSH moves the pointer and stores; a POP reads and moves the pointer back. Show the pointer, the changed locations and the unchanged ones.
3. **Count questions:** the item there longest is at the **bottom**. Pushes until full = top location – current top (for a "last item" pointer).
4. **Complete Push:** IF $\text{TopOfStack} = \text{Max}$ THEN RETURN FALSE, then increment the pointer, store, RETURN TRUE.
Complete Pop: test for empty ($\text{TopOfStack} = 0$, or < 1), otherwise read $\text{Stack}[\text{TopOfStack}]$, decrement, return the value.
5. If Pop returns a record (a value and a flag saying whether it exists), declare a local variable of that record type, set its fields, and return that variable.

10. Search or update an array of records (8 questions)

These are the long Paper 2 questions: a global array of records, some unused.

1. **Header** with the given name and parameters (FUNCTION ... RETURNS BOOLEAN if a value comes back).
2. **Loop over the whole array** by index; for "efficient", a WHILE that also stops once found (or once a limit is reached).
3. **Test the fields with dot notation:** $\text{Loans}[\text{Index}].\text{StudentID} = \text{ThisID}$ AND $\text{Loans}[\text{Index}].\text{OnLoan} = \text{TRUE}$.
4. **Act:** count, total, update a field, or fill the first unused record (key field = " ") and set every field.
5. **After the loop:** output the counts with a message, or return TRUE/FALSE for found or not found. Output "not found" only **after** the loop, once.

11. Bubble sort (5 questions)

1. An **outer loop** for the passes and an **inner loop** that compares neighbours, $\text{Data}[\text{Index}]$ with $\text{Data}[\text{Index} + 1]$.
2. The inner loop goes to one less than the last index, so $\text{Index} + 1$ stays in bounds.
3. **Swap** using a temporary variable.
4. **Efficient:** a Swapped flag set to FALSE at the start of each pass and TRUE on a swap, with the outer loop ending when a pass makes no swaps; and the inner loop's end decreasing by 1 after each pass.
5. **Parallel arrays or a 2D array:** compare by the key (the mark column), but swap **every** linked value: both arrays, or the whole row (each column of the 2D array) with a temporary variable each.
6. **Describing it in words:** repeat passes through the array; in each pass compare each pair of neighbours and swap if out of order; stop when a pass makes no swaps.

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 gym stores data about class bookings. Each booking will be held in a record of type `ClassBooking`.

Field	Example value	Comment
BookingRef	"GX2291"	unique to each booking
MemberNum	4417	used as an array index
ClassCode	'Y'	a letter for the type of class
ClassDate	21/10/2026	the date of the class
Fee	6.50	the cost of the class
Attended	FALSE	TRUE once the member has attended

(a) Write pseudocode to declare the record type `ClassBooking`. **[4]**

(b) A 1D array `Booking` of 300 elements stores the bookings. Write pseudocode to declare it. **[2]**

(c) Write one pseudocode statement to change the fee of the 15th booking to 8.25. **[1]**

(d) Unused elements have `BookingRef` set to an empty string. Write pseudocode for a function `AttendedFees ( )` that returns the total fee for all bookings that were attended. **[5]**

(a)

```

TYPE ClassBooking
    DECLARE BookingRef : STRING
    DECLARE MemberNum  : INTEGER
    DECLARE ClassCode  : CHAR
    DECLARE ClassDate  : DATE
    DECLARE Fee        : REAL
    DECLARE Attended   : BOOLEAN
ENDTYPE

```

B1 TYPE and ENDTYPE; **B1** BookingRef and MemberNum; **B1** ClassCode and ClassDate; **B1** Fee and Attended.

(b) DECLARE `Booking` : ARRAY[1:300] OF `ClassBooking` **B1** for the bounds, **B1** for OF `ClassBooking`.

(c) `Booking[15].Fee ← 8.25` **B1**

(d)

```

FUNCTION AttendedFees() RETURNS REAL
    DECLARE Index : INTEGER
    DECLARE Total : REAL

```

```

Total ← 0
FOR Index ← 1 TO 300
    IF Booking[Index].BookingRef <> "" AND Booking[Index].Attended = TRUE THEN
        Total ← Total + Booking[Index].Fee
    ENDIF
NEXT Index
RETURN Total
ENDFUNCTION

```

B1 header with RETURNS REAL and local variables declared; **B1** Total set to 0 before a loop over all 300 elements; **B1** dot notation with the index; **B1** test for used **and** attended, then add the fee; **B1** return the total after the loop.

With three attended bookings at 6.50, 8.25 and 6.50, and one booking not attended, the function returns 21.25.

Example 2

(a) A stack is stored in an array `Stack[1:6]`. `TopOfStack` holds the index of the last item added, and is 0 when the stack is empty. At present `Stack[1]` is 'H', `Stack[2]` is 'W', `Stack[3]` is 'D' and `TopOfStack` is 3.

(i) The operations PUSH 'L', PUSH 'P', POP, POP, POP, PUSH 'G' are carried out in that order. Give the contents of the stack, the value of `TopOfStack`, and the values popped. **[3]**

(ii) State how many more items can then be pushed before the stack is full. **[1]**

(b) A circular queue is stored in an array `Queue[1:6]`. It holds 'A' at 5, 'B' at 6, 'C' at 1 and 'D' at 2, with `FrontOfQueue` 5, `EndOfQueue` 2 and `NumItems` 4.

(i) One item is removed, then 'E' and 'F' are added. Give the item removed and the new values of the three variables. **[3]**

(ii) Describe how an item is added to this queue. **[4]**

(a)(i) Step by step: PUSH 'L' → 'L' at 4, top 4. PUSH 'P' → 'P' at 5, top 5. POP → 'P', top 4. POP → 'L', top 3. POP → 'D', top 2. PUSH 'G' → 'G' at 3, top 3.

The stack holds 'H' (1), 'W' (2) and 'G' (3), with 4 to 6 empty and `TopOfStack` = 3. **B1** for the contents, **B1** for the pointer. The values popped are 'P', 'L' and 'D', in that order. **B1**

(a)(ii) $6 - 3 = 3$ items. **B1**

(b)(i) Removing takes 'A' from the front (index 5), and the front pointer moves to 6. Adding 'E': the end pointer moves from 2 to 3. Adding 'F': it moves to 4.

`FrontOfQueue` = 6 **B1**, `EndOfQueue` = 4 **B1**, `NumItems` = $4 - 1 + 2 = 5$ **B1**. The item removed is 'A'. (Index 5 is now free, so one more item could be added: the end pointer would go to 5.)

(b)(ii)

1. If `NumItems` is 6, the queue is full: report an error and stop. **B1**
2. Otherwise, if `EndOfQueue` is 6 set it to 1 (wrap around), else add 1 to it. **B1**
3. Store the new item in `Queue` at the index `EndOfQueue`. **B1**
4. Add 1 to `NumItems`. **B1**

Example 3

A linked list of city names is stored in two arrays, `City` and `Next`, with indexes 1 to 6. The value 0 is used as the null pointer.

Index	City	Next
1	"Lima"	
2	"Cairo"	
3	"Oslo"	
4	"Bern"	
5	"Kyiv"	
6		0

(a) Complete the `Next` column so that the list is in alphabetical order, and give the start pointer and the free list pointer. **[3]**

(b) Explain why 0 is a suitable null pointer here. **[1]**

(c) Describe how "Dublin" is added to the list so that it stays in alphabetical order. **[4]**

(d) Give one advantage of storing the cities in a linked list rather than in a sorted array. **[1]**

(a) Alphabetical order is Bern (4), Cairo (2), Kyiv (5), Lima (1), Oslo (3).

Index	City	Next
1	"Lima"	3
2	"Cairo"	5
3	"Oslo"	0
4	"Bern"	2
5	"Kyiv"	1
6		0

B1 Bern → Cairo → Kyiv correct; **B1** Lima → Oslo and Oslo's null pointer 0. Start pointer = 4 and free list pointer = 6. **B1**

(b) The indexes run from 1 to 6, so 0 can never be the index of a node; it can only mean the end of the list. **B1**

(c)

1. Store "Dublin" in the node at the free list pointer, index 6. **B1**
2. Set the free list pointer to that node's `Next` value (0: the free list is now empty). **B1**
3. Follow the pointers from the start to find the node Dublin follows: Cairo (2), because Cairo < Dublin < Kyiv. Set `Next [ 6 ]` to Cairo's pointer, 5. **B1**
4. Set Cairo's pointer, `Next [ 2 ]`, to 6. **B1**

The list is now Bern → Cairo → Dublin → Kyiv → Lima → Oslo.

(d) A new item is put in order by changing two pointers; no items have to be moved along the array. **B1**

Example 4

A text file `Scores.txt` holds one line per player in the form `<Name>,<Score>`, for example `"Ravi, 1450"`. Names never contain a comma. The global arrays `Names` (STRING) and `Scores` (INTEGER) each have 50 elements.

(a) Write pseudocode for a function `LoadScores()` that copies the names and scores from the file into the arrays, starting at index 1, stops at the end of the file or when the arrays are full, and returns the number of players loaded. [7]

(b) Write pseudocode for a procedure `SortScores()` that takes the number of players loaded as a parameter and uses an efficient bubble sort to put both arrays into **descending** order of score. [6]

(a)

```

FUNCTION LoadScores() RETURNS INTEGER
    DECLARE Line, ThisName : STRING
    DECLARE Count, Pos : INTEGER
    Count ← 0
    OPENFILE "Scores.txt" FOR READ
    WHILE NOT EOF("Scores.txt") AND Count < 50
        READFILE "Scores.txt", Line
        Pos ← 1
        WHILE MID(Line, Pos, 1) <> ','
            Pos ← Pos + 1
        ENDWHILE
        ThisName ← LEFT(Line, Pos - 1)
        Count ← Count + 1
        Names[Count] ← ThisName
        Scores[Count] ← STR_TO_NUM(RIGHT(Line, LENGTH(Line) - Pos))
    ENDWHILE
    CLOSEFILE "Scores.txt"
    RETURN Count
ENDFUNCTION

```

- **B1** variables declared and Count initialised;
- **B1** file opened in READ mode and closed after the loop;
- **B1** loop until the end of the file **or** the arrays are full;
- **B1** a line read inside the loop;
- **B1** the position of the comma found;
- **B1** name and score cut out with LEFT and RIGHT, the score converted with STR_TO_NUM;
- **B1** both stored at the next index, and the count returned.

For the line `"Ravi, 1450"` the comma is at position 5, so the name is `LEFT(Line, 4) = "Ravi"` and the score is `RIGHT(Line, 9 - 5) = "1450"`, converted to 1450.

(b)

```

PROCEDURE SortScores(NumPlayers : INTEGER)
    DECLARE Index, Boundary, TempScore : INTEGER
    DECLARE TempName : STRING
    DECLARE Swapped : BOOLEAN
    Boundary ← NumPlayers - 1

```

```

REPEAT
    Swapped ← FALSE
    FOR Index ← 1 TO Boundary
        IF Scores[Index] < Scores[Index + 1] THEN
            TempScore ← Scores[Index]
            Scores[Index] ← Scores[Index + 1]
            Scores[Index + 1] ← TempScore
            TempName ← Names[Index]
            Names[Index] ← Names[Index + 1]
            Names[Index + 1] ← TempName
            Swapped ← TRUE
        ENDIF
    NEXT Index
    Boundary ← Boundary - 1
UNTIL NOT Swapped OR Boundary < 1
ENDPROCEDURE

```

- **B1** outer loop for the passes and inner loop through the used elements;
- **B1** compare neighbours with < (descending);
- **B1** swap the scores with a temporary variable;
- **B1** swap the names too, so each name stays with its score;
- **B1** Swapped flag reset each pass and set on a swap, the outer loop ending when there are no swaps;
- **B1** the inner loop's end reduced by 1 each pass.

For the scores 1450, 980, 2210, 1720 (Ravi, Ana, Joe, Mia) the passes give 1450, 2210, 1720, 980, then 2210, 1720, 1450, 980, then a pass with no swaps. The final order is Joe, Mia, Ravi, Ana.

Common mistakes

- **Wrong data types.** Money as INTEGER; a phone number or ID as INTEGER; "18/04/2021" (in quotes) as DATE. Look at the quotes and whether a fraction is possible.
- **Using a type name as an identifier**, such as `DECLARE Date : DATE`. It is a keyword.
- **Forgetting the record type in the array declaration** (`ARRAY[1:500] OF RECORD`), or leaving out `ENDTYPE`.
- **Missing dot notation or index.** `Loan.OnLoan` or `Loan[Index]` alone is not a field; write `Loan[Index].OnLoan`.
- **Looping one element too many or too few**, such as `FOR Index ← 0 TO 30` for `ARRAY[1:30]`, or a bubble sort inner loop that reaches the last index, so `Index + 1` is out of bounds.
- **Not closing the file**, opening it in the wrong mode, or putting a parameter's name in quotes (`OPENFILE "FileName"` opens a file literally called `FileName`).
- **Testing the wrong thing for the end of a file:** comparing `EOF()` with "", or testing the file name. `EOF(File)` is a `BOOLEAN`; loop `WHILE NOT EOF(File)`. Checking for an empty line instead fails if the file has blank lines.
- **Writing a number or BOOLEAN to a text file without converting it to a string**, or doing arithmetic on a line read from a file without `STR_TO_NUM`.
- **Swapping without a temporary variable**, so both elements end up with the same value; or in parallel arrays, swapping only the key array, so names and scores no longer match.
- **Outputting a linked list in array order** (index 0 to 9) instead of following the pointers; testing the node's data for empty instead of testing the pointer for null.

- **Adding to a linked list without the free list:** searching for any empty element instead of using the free list pointer, and not updating the previous node's pointer.
- **Giving the null pointer a vague meaning** ("it points to nothing"). Say it shows the end of the list (or an empty list).
- **Confusing stacks and queues:** describing a stack with "first in, first out", or a queue with one pointer. Forgetting to wrap a circular queue's pointer.
- **Copying the question back** in "describe" answers ("check if the array is full"). Say how: "if the index is greater than 20".

How to get the marks

- **"Write pseudocode"** is marked feature by feature (declarations, initialisation, loop, the key test, the update, the result). Every bullet in the question is usually a mark, so cover each one.
- **"All variables used must be declared"** means a `DECLARE` with a type for every local variable, including the loop index.
- **"Efficient"** means stop as soon as possible: a conditional loop that ends when found, and a bubble sort with a swap flag and a shrinking inner loop.
- **Use the given identifiers** exactly (`Loan`, `StudentID`, `ThisStack`) and Cambridge syntax: `←`, `ENDTYPE`, `ENDWHILE`, `NEXT Index`, file statements in capitals.
- **"Do not use pseudocode"** means numbered steps in words, precise enough for a beginner to code from: name the file and mode, the loop and when it ends, the conversion, the test.
- **Diagram questions** (stack, queue, linked list) are marked on the pointer values, the cells that change, and the cells that stay the same. Always show every pointer.
- **"Explain" and "justify"** need a reason: why a separator works (it never appears in the data), why 0 is a null pointer (it is not a valid index), why a queue (items are processed in order of arrival).
- **"State" or "Identify"** needs one short answer; only the first is marked if you give several.
- **Benefits** must fit the thing asked about: for an array of records, mixed types under one identifier and processing in a loop; for a text file, data kept after the program ends and no fixed size.

Quick check

1. Give the most suitable data type for: (i) 'Q'; (ii) "07700900123"; (iii) 12.5; (iv) FALSE; (v) 03/04/2026; (vi) the number of tickets sold.
2. `DECLARE Grid : ARRAY[0:19, 1:8] OF CHAR`. State the number of dimensions, the number of elements, and the lower and upper bounds of the first dimension.
3. A stack in `ARRAY[1:6]` is empty, with `TopOfStack = 0` pointing to the last item added. These operations are carried out: `PUSH 4`, `PUSH 9`, `POP`, `PUSH 2`, `PUSH 7`, `POP`. Give the stack's contents, `TopOfStack` and the values popped.
4. A bubble sort compares every pair of neighbours on each pass and stops after a pass with no swaps. It sorts 8, 2, 6, 4, 1 into ascending order. Give the order after the first pass, and the number of passes it makes in total.
5. A program adds a line to `Log.txt` every time a user logs in. Explain why the file should be opened in APPEND mode, and state what must be done to the BOOLEAN value `Success` before it is written to the file.

Answers

1. (i) CHAR; (ii) STRING (a phone number: no arithmetic, and the leading 0 must be kept); (iii) REAL; (iv) BOOLEAN; (v) DATE; (vi) INTEGER.
2. 2 dimensions; $20 \times 8 = 160$ elements; first dimension lower bound 0, upper bound 19.
3. PUSH 4 (top 1), PUSH 9 (top 2), POP gives 9 (top 1), PUSH 2 (top 2), PUSH 7 (top 3), POP gives 7 (top 2). The stack holds 4 at 1 and 2 at 2; `TopOfStack = 2`; the values popped were 9 then 7.
4. First pass: 2, 6, 4, 1, 8. The second gives 2, 4, 1, 6, 8, the third 2, 1, 4, 6, 8 and the fourth 1, 2, 4, 6, 8. The fifth pass makes no swaps, so there are 5 passes in total (the last only confirms the order).
5. APPEND adds the new line to the end and keeps the lines already there; WRITE would create a new file and erase every earlier log entry. Success must be converted to a string (such as "TRUE") because a text file holds only strings.

Past questions to try

- [9618/22/O/N/24 Q3](#): a stack whose pointer points to the next free location, and completing Push.
- [9618/21/M/J/24 Q3](#): a linked list with a free list, built from two arrays.
- [9618/21/O/N/24 Q3](#): reading a text file and keeping its last three lines.
- [9618/23/O/N/23 Q3](#): a circular queue in an array, and the steps to add an item.