

Data types and structures

A-Level Computer Science 9618 · Topic 10 revision notes

Everything from the 5 lessons of this topic on a few pages: the key idea of each lesson, the diagram to remember, the points to learn and the traps examiners set. Read it, cover it, test yourself.

1. Data types and records

2. 1D and 2D arrays

3. Linear search and bubble sort

4. Text files

5. Stacks, queues and linked lists

thelucidstem.com

1 Data types and records

Syllabus 10.1

A **data type** decides which values a variable can store and which operations are allowed on it. 9618 pseudocode uses **INTEGER**, **REAL**, **CHAR**, **STRING**, **BOOLEAN** and **DATE**. A **record** is a **composite** (user-defined) data type that holds a set of related data items, called **fields**, which can be of **different data types**, all under **one identifier**. It is defined with **TYPE ... ENDTYPE** and each field is accessed with **dot notation**, such as `Pupil1.YearGroup`.

one identifier, five fields, four different data types

Pupil1 : StudentRecord

.LastName	"Khan"	STRING
.YearGroup	12	INTEGER
.FormGroup	'B'	CHAR
.DateOfBirth	14/03/2009	DATE
.Boarder	FALSE	BOOLEAN

Fig. 1: a record variable groups fields of different data types under one identifier

Worked example: Justify each choice

A library system stores, for each book: a book ID such as B0457, the number of copies, the replacement cost in dollars, the date it is next due back, a category code of one letter and whether it is on loan. Choose a data type for each item.

1. Book ID: it mixes a letter and digits, so **STRING**.
2. Number of copies: always a whole number, so **INTEGER**.
3. Replacement cost, such as 12.50: needs a fractional part, so **REAL**.
4. Due date: **DATE**. Category code: one letter, so **CHAR**. On loan: yes or no, so **BOOLEAN**.

Answer: **STRING**, **INTEGER**, **REAL**, **DATE**, **CHAR**, **BOOLEAN**

Type, then field.

- ✓ **Six types:** INTEGER, REAL, CHAR, STRING, BOOLEAN, DATE; justify each from the data and its use.
- ✓ **Digits that are not quantities** (phone numbers, IDs) are STRING.
- ✓ **Record:** `TYPE Name`, a DECLARE line per field, `ENDTYPE`; then DECLARE a variable of that type.
- ✓ **Access:** `Variable.Field`; in an array of records, `Array[Index].Field`.

Watch out!

"Phone number: INTEGER because it is a number" loses the mark.

Digits that are an identifier, not a quantity, belong in a STRING.

TYPE ... ENDTYPE only defines the type. You still need `DECLARE ThisBook : BookRecord` before any data can be stored. Writing

```
BookRecord.Title ← "Dune"
```

uses the type name, not a variable, and earns no mark.

The index belongs to the array, not the field:

```
Library[12].Copies, never
```

```
Library.Copies[12].
```

2 1D and 2D arrays

Syllabus 10.2

An **array** is a fixed-length data structure of **elements** that all have the **same data type**, held under one identifier. Each element is reached by its **index** (subscript). The **lower bound** is the index of the first element and the **upper bound** the index of the last, so `ARRAY[1:30]` has 30 elements. A **1D array** is a list with one index; a **2D array** is a table reached by **[row, column]**. Arrays are processed one element at a time with a **FOR loop**; a 2D array needs **nested** loops.

Scores : ARRAY[1:6] OF INTEGER

12	31	7	31	25	9
1	2	3	4	5	6

lower bound upper bound

Marks : ARRAY[1:3, 1:4] OF INTEGER

row 1	5	8	6	9
row 2	7	7	10	4
row 3	3	9	8	6

Marks[2, 3] = 10

Fig. 1: a 1D array of six integers and a 2D array of 3 rows by 4 columns

Worked example: Trace table

Scores holds 12, 31, 7, 31, 25, 9 (Fig. 1). Trace the pseudocode above and state the output.

1. Before the loop: Max = 12, MaxIndex = 1.
2. $i = 2$: $31 > 12$ is TRUE, so Max = 31, MaxIndex = 2.
3. $i = 3$: $7 > 31$ is FALSE. $i = 4$: $31 > 31$ is FALSE (not strictly greater), so MaxIndex stays 2.
4. $i = 5$: $25 > 31$ FALSE. $i = 6$: $9 > 31$ FALSE. The loop ends.

Answer:

i Scores[i] Scores[i] > MaxMaxMaxIndex

		12	1
2	31	TRUE	31 2
3	7	FALSE	
4	31	FALSE	

525 FALSE

69 FALSE

Output: Highest 31 at position 2. Because the test is $>$ and not $\geq$, the first position of a tied maximum is kept.

Bounds, then loops.

- ✓ **Elements** = upper bound – lower bound + 1; a 2D array has rows $\times$ columns.
- ✓ **Declare:** `DECLARE X : ARRAY[1:n]` OF type or `ARRAY[1:r, 1:c]` OF type.
- ✓ **Process:** a FOR loop for 1D; nested FOR loops for 2D, with totals reset in the right loop.
- ✓ **Choose:** 2D when each value has a row and a column; records when the data types differ.

Watch out!

Start the maximum at the first element, `Max  $\leftarrow$  Scores[1]`, not at 0: if every value were negative, a start of 0 would give the wrong answer.

Keep the index order the same as the declaration. If the array is `[1:3, 1:4]`, then `Marks[4, 2]` is out of range: there is no row 4.

Do not mix data types in one array. "A 2D array with a column of names and a column of marks" would need every element to be `STRING`, so the marks could not be used in arithmetic without conversion.

3 Linear search and bubble sort

Syllabus 10.2

A **linear search** compares the target with each element in turn, from the first, and stops when it is **found** or the **end of the array** is reached. It works on **unsorted** data; in the worst case it makes **n comparisons**. A **bubble sort** compares each pair of **adjacent** elements and **swaps** them (using a temporary variable) if they are in the wrong order. After each **pass** the largest unsorted value has reached its final place. The **improved** version uses a **flag** to stop as soon as a pass makes no swaps and shortens each pass by one.

bubble sort: the array after each pass

start	25	12	40	8	19	
pass 1	12	25	8	19	40	3 swaps
pass 2	12	8	19	25	40	2 swaps
pass 3	8	12	19	25	40	1 swap
pass 4	8	12	19	25	40	0 swaps: stop

green = in its final place: the largest unsorted value bubbles to the end each pass

Fig. 1: an improved bubble sort of five integers, shown after each pass

Worked example: Count the comparisons

Names holds Raj, Ola, Mei, Kit, Zed, Ann, Lou, Ivy in elements 1 to 8. State the output and the number of comparisons with SearchName for (a) Ann (b) Tom.

1. (a) Compare with Raj, Ola, Mei, Kit, Zed: no match, Index rises to 6.
2. Names[6] = Ann, so Found becomes TRUE and the WHILE condition is now FALSE: 6 comparisons.
3. (b) Tom matches none of the 8 names. Index goes up to 9, so $\text{Index} \leq 8$ is FALSE and the loop stops: 8 comparisons.
4. Found is still FALSE, so the second branch of the final IF runs.

Answer: (a) Ann found at position 6 after 6 comparisons; (b) Tom not found after 8 comparisons

Flags and neighbours.

- ✓ **Linear search:** WHILE not found AND not past the end; compare, then move on; report after the loop.
- ✓ **Bubble sort:** compare List[j] with List[j + 1]; swap with Temp if in the wrong order.
- ✓ **Bounds:** j stops one before the last element in use, so List[j + 1] stays in range.
- ✓ **Improve:** reset a Swapped flag every pass, stop when it stays FALSE, shorten each pass.

Watch out!

Do not put `OUTPUT "not found"` inside the loop: it would be output for every element that does not match. Decide only after the loop has finished, using the Found flag.

Writing the swap as `Nums[j] ← Nums[j + 1]` then `Nums[j + 1] ← Nums[j]` copies one value into both elements. Always use a temporary variable.

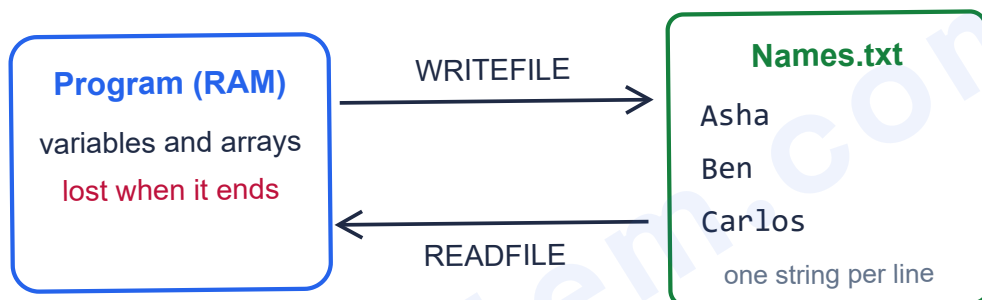
Set `Swapped ← FALSE` at the start of every pass, inside the REPEAT. Setting it once before the loop means a single swap in pass 1 keeps it TRUE for ever.

4 Text files

Syllabus 10.3

Data in variables is held in **RAM** and is **lost when the program ends**. A **file** is stored on **secondary storage**, so the data **persists** and can be used the next time the program runs or by another program. A **text file** is a sequence of **lines**, each read or written as a **string**, in order. A file is opened in one **mode**: **READ**, **WRITE** (creates a new file; any existing data is lost) or **APPEND** (adds after the existing data). Use **READFILE**, **WRITEFILE**, the **EOF** function and **CLOSEFILE**.

a text file outlives the program that wrote it



modes: READ, WRITE (new file, old data lost), APPEND (add to the end)
files are stored on secondary storage, so the data persists

Fig. 1: lines are written to and read from a text file that remains after the program ends

Worked example: Trace the read loop

Names.txt contains five lines: Asha, Ben, Carlos, Abdul, Bea. State the output.

1. Line 1 Asha: Total 1, first character A, CountA 1.
2. Line 2 Ben: Total 2. Line 3 Carlos: Total 3.
3. Line 4 Abdul: Total 4, CountA 2. Line 5 Bea: Total 5.
4. No lines are left, so EOF returns TRUE and the loop ends.

Answer: 2 of 5 names start with A

One mode at a time.

- ✓ **Why:** variables are lost when the program ends; a file stores data permanently for later use.
- ✓ **Modes:** READ, WRITE (new file, old data lost), APPEND (add to the end).
- ✓ **Read loop:** `WHILE NOT EOF(file),
 READFILE file, StringVar,
 process, ENDWHILE`.
- ✓ **Finish:** CLOSEFILE every file you opened, after the loop.

Watch out!

"Files are needed to store data" is too vague. Say that variables are lost when the program ends, so the data must be stored permanently in a file to be used again later.

READFILE reads one line into a STRING variable. If a line holds a number, it is still text until it is converted, so do not declare the READFILE variable as INTEGER.

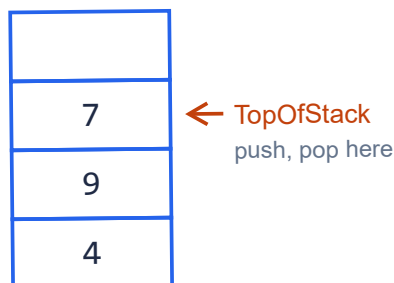
Always CLOSEFILE when you have finished. An examiner checks that every OPENFILE has a matching CLOSEFILE, with the same file name, after the loop rather than inside it.

5 Stacks, queues and linked lists

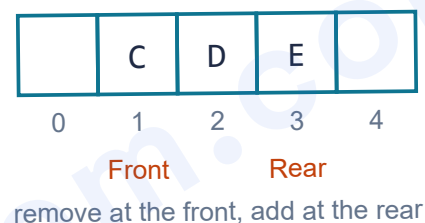
Syllabus 10.4

An **abstract data type (ADT)** is a collection of data together with a set of **operations** on that data. A **stack** is **last in, first out (LIFO)**: items are pushed onto and popped from the top, tracked by a **top pointer**. A **queue** is **first in, first out (FIFO)**: items join at the **rear** and leave from the **front**; a **circular queue** reuses freed spaces. A **linked list** is a set of **nodes**, each holding data and a **pointer** to the next node, with a **start pointer**, a **null pointer** at the end and a **free list** of unused nodes. All three can be **implemented using arrays**.

Stack (LIFO)



Queue (FIFO)



Linked list



each node holds data and a pointer to the next node; the last pointer is null

Fig. 1: a stack, a queue and a linked list (the stack after PUSH 4, PUSH 9, PUSH 2, POP, PUSH 7)

Worked example: Trace a stack

Stack is `ARRAY[0:5]` and `TopOfStack` starts at `-1`. Trace `PUSH 4`, `PUSH 9`, `PUSH 2`, `POP`, `PUSH 7`, `POP`, `POP`.

1. `PUSH 4, 9, 2`: `TopOfStack` goes 0, 1, 2 and the stack holds 4, 9, 2 (bottom to top).
2. `POP` returns 2; `TopOfStack` = 1.
3. `PUSH 7`: `TopOfStack` = 2 and 7 overwrites the old 2 in `Stack[2]` (this is the state in Fig. 1).
4. `POP` returns 7 (`TopOfStack` 1); `POP` returns 9 (`TopOfStack` 0).

Answer:

Operation `TopOfStack` Stack (bottom to top) Returned

start -1 empty

PUSH 4 0 4

PUSH	9	1	4, 9	
PUSH	2	2	4, 9, 2	
POP		1	4, 9	2
PUSH	7	2	4, 9, 7	
POP		1	4, 9	7
POP		0	4	9

Check, then move.

- ✓ **Stack (LIFO):** push = check full, top + 1, store; pop = check empty, read, top - 1.
- ✓ **Queue (FIFO):** add at the rear, remove from the front; circular pointers wrap with MOD; keep a count.
- ✓ **Linked list:** node = data + pointer; start pointer, null pointer, free list.
- ✓ **Insert:** new node's pointer first, then the previous node's pointer.

Watch out!

Do not say a pop "deletes" the value from the array. Only the pointer changes; the old value is simply overwritten by the next push.

In a circular queue, Rear = upper bound does not mean the queue is full; Rear can wrap round. Test fullness with the count (or by comparing the pointers carefully).

When inserting, set the new node's pointer before changing the previous node's pointer. Doing it the other way round loses the address of the rest of the list.