Programming Elements

definitions are highlighted!

Coding standards

- Proper use of indentation and white space — use blank lines and indentation to separate logical sections
- Use of meaningful identifiers — choose descriptive and self-explanatory variable and function names; avoid single-character identifiers unless widely-recognised (e.g. “i” in loops)
- Inclusion of comments to clarify code — include essential details such as the programmer’s name, date, program description and version control information; provide clear explanations of helper functions and algorithms, including their objectives, expected inputs and outputs; avoid unnecessary explanations of syntax
    - Must explain string operations (e.g. concatenation, searching, splitting)
    - Must explain conversions between types (e.g. converting string to integer)
- Each of these carry 2 marks in the A level practical examination.

Array

- A static data structure made up of a finite, ordered set of elements of the same type
    - Finite — there is a specific number of elements in an array
    - Ordered — each element has a numbered position in the array called the array index
    - List vs Array:
        - List — can store different types of elements
        - Array — can only store elements of the same type

- Has a fixed size determined at the time of declaration, meaning that the number of elements cannot be changed during execution
- Elements of an array are all stored in contiguous (i.e. adjacent) memory locations.
- Access to specific elements in the array is done using the name of the array with an index in square brackets.
- Declaring arrays in pseudocode:
    - Format — “DECLARE <identifier> : ARRAY [<L1 : U1>, <L2 : U2>] OF <data type>”
        - L1, L2, U1 and U2 represent the lower and upper bounds of each dimension respectively.
        - e.g. “DECLARE Attendance: ARRAY [1:5, 1:4] OF CHAR”
    - When specifying an element, we will say “Attendance [3,2]” — this refers to index 3 in the first dimension (1:5) and index 2 in the second dimension (1:4).
- In the A Level syllabus, we won’t go beyond 2D arrays.

Trace table

- A structured method of recording the result of each step on paper during a dry run
    - A dry run is a method of debugging which involves manually stepping through the code on paper to observe its behaviour before execution.
- Consists of columns where each column represents a variable/condition/output, but not every variable/condition/output in an algorithm requires a column
- The purpose of using this is to simulate how a computer would execute an algorithm line by line — by tracking variable changes, condition evaluations and outputs, we can identify logical errors and verify correctness.

    - Can also be used to determine what a given algorithm has been designed to do
- If a variable's value does not change in a particular step, it does not need to be recorded again.
    - Helps to maintain clarity while analysing algorithm behaviour.
    - Thus, a trace table may contain empty cells.
- e.g.

```
num ← 0
total ← 0
FOR i ← 1 TO 3
    FOR j ← 1 TO 3
        num ← j * i
        total ← total + num
    ENDFOR
ENDFOR
OUTPUT total
```

i	j	num	total	OUTPUT
		0	0	
1	1	1	1	
	2	2	3	
	3	3	6	
2	1	2	8	
	2	4	12	
	3	6	18	
3	1	3	21	
	2	6	27	
	3	9	36	
				36

Data verification

- *A way of preventing errors when data is copied from one medium to another*
- Involves checks to ensure that the data is accurately copied from the source to the destination with no alteration during the process; if differences are detected, it indicates an error in the transfer
- Does not guarantee data accuracy, as incorrect data could have been entered initially
- Can take place when:
    - Data is entered at the human-machine interface
    - Data is copied between components within a computerized system
- Can help to detect errors that occur in the transfer of data
    - *Transcription error — a specific data entry error that occurs when information is misinterpreted or incorrectly transferred*

        - Often due to human operators or optical character recognition (OCR) programs
    - *Transposition error — an error that occurs when two adjacent characters are accidentally switched or reversed during data entry*
        - Typically due to typing speed or inattention
        - Results in incorrect character placement, leading to errors in the data
- Techniques:
    - Double entry — the data is entered twice and the computer compares the 2 entries to ensure consistency
        - e.g. re-entering password during the account registration process
    - Visual check — having a person review the data entered to compare it with the original source and ensure that they match
        - e.g. data is displayed on the screen after being entered for the user to check and confirm that it is correct

Data validation

- *An automated process used to ensure that the input data meets a specified set of rules, ensuring that it is sensible, complete, reasonable and within acceptable boundaries*
    - Prevents:
        - Poor data entries from crashing the program
        - Users from receiving impossible/nonsensical outputs
- Helps to identify inputs that are obviously incorrect, but this does not verify the accuracy of the data either as data entered could be reasonable yet incorrect
- Data is only processed after it passes the validation checks.

- Validation checks:
  - Range check — ensures that the data falls within a predefined range
  - Format check — ensures that the data follows a specific pattern or format
  - Length check — ensures that the data has the correct number of characters
  - Presence check — ensures that all the necessary fields are filled out
  - Type check — ensures that the data is of the correct data type
  - Check digit — uses a check digit (an extra character added which is calculated from the other characters) to allow the data to be self-checking
    - e.g.

Example of Check Digit: Modulus-11 Arithmetic

The modulus-11 method to calculate a check digit follows this algorithm:

1. Starting from the units column, assign weights to each digit. The units column gets the weight 2, the tens column gets the weight 3, the hundreds column gets the weight 4 and so on.
2. Multiply each digit by its corresponding weight.
3. Add the individual products.
4. Divide the total sum of products by 11 and note the remainder.
5. Subtract the remainder from 11.

The result is the check digit. If the result is 10, the check digit is represented by the character 'X'. Otherwise, it is an integer.

Example: Product code `1234` (with the fifth digit as the check digit)

Product code (D)	1	2	3	4
Weight (W)	5	4	3	2
Product (D*W)	5	8	9	8
Sum of Product	5+8+9+8 = 30			
Remainder after dividing by 11	30%11 = 8			
Subtract remainder from 11	11 – 8 = 3			
Check digit	3			

Therefore the complete product code is **12343**.

Errors in programs

- Syntax error
    - Occurs when the code does not follow the rules of the programming language, so the computer cannot interpret it
    - Detected when the compiler/interpreter try to translate the source code to machine code (during the syntax analysis stage of the program's translation)
    - Must be fixed before the program is executed as a program with syntax errors cannot run
- Runtime error
    - Detected during program execution — typically cause the program to crash/fail immediately
    - Can occur if a mistake is made in the processing algorithm or as a result of external effects not catered for by the program (e.g. lack of memory, unexpected data input)
    - Input error — occurs when the program receives incorrect data from the user
    - Type mismatch — occurs when a variable is used with an incompatible type in an operation
    - Index out of bounds
    - Undeclared variable
    - Arithmetic error (e.g. dividing by zero)
    - Infinite loop — freezes the program because the exit condition is never reached
    - Out of memory — occurs when the program uses more memory than is available
    - Memory leak — occurs when memory is allocated but not proper released
    - File I/O error — occurs when there are issues with file operations
    - Overflow — occurs when a value exceeds the maximum limit of a data type

        - Each computer has a well-designed range of values that it can represent. When the computer attempts to handle a number that is too large for it during execution, overflow error occurs.
    - Underflow — occurs when a value goes below the minimum limit of a data type
    - Stack overflow
- Logic error
    - A mistake made in the design of a program
    - Unlikely to generate an error message; program may run but will not produce the expected results
    - May go unnoticed as they do not necessarily disrupt the program's execution
    - Detected when a test case is run and it is observed that the actual output of the program is different from the expected one

Test cases

- **Purpose:** To validate the testing coverage of the application
    - Also helps to check that any data validation created is working as expected
- Consist of components such as input, purpose and expected output, which help to determine whether a feature of the application works as intended
- Designed in such a way that only one thing is tested at a time (i.e. atomic test case)
    - Do not overlap or complicate test cases!
- It is important to run a program through multiple test cases to detect runtime and logic errors.
- Should aim to cover a wide range of data (at least one of each type):

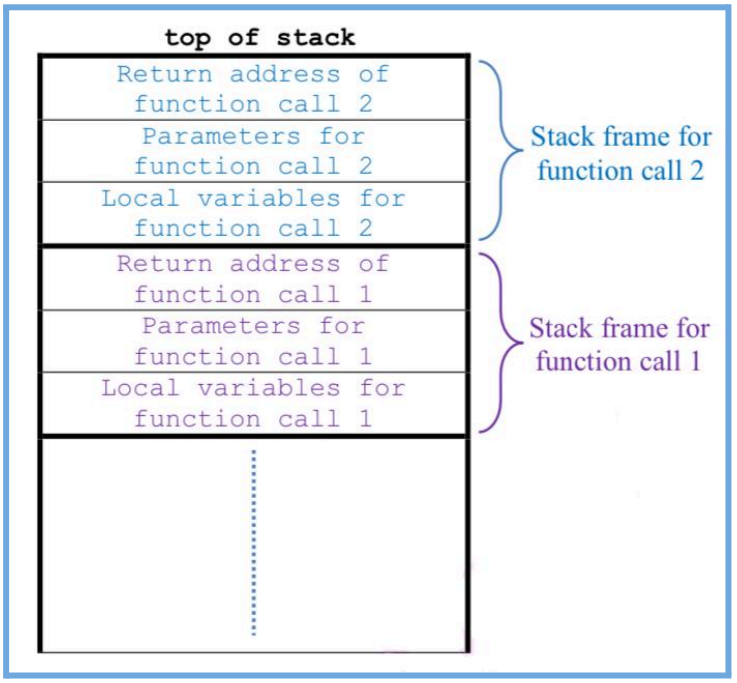
- *Normal — sensible, expected data that should be accepted and processed by the system*
  - e.g. in the middle of the accepted range
- *Abnormal (aka invalid, erroneous) — falls outside of what is acceptable and should be **rejected** by the system*
  - e.g. inputting a float instead of an integer
- *Extreme — the upper or lower limits of expectations that should be **accepted** by the system, or when special handling of the input data is required*
  - e.g. the use of '0' as an ending value in a program that calculates the average of some numbers
  - In systems that deal with text, the extreme values are defined by how long the text can be.

- Test plan — a comprehensive set of test cases that covers all types of data and scenarios
  - Includes expected outputs for each test case
  - Presented in a table format
  - e.g.

Input (examples)	Purpose	Expected output	Actual output
10, 25, 65, 88	Normal data	Accepted	*(record exactly what was output by the program, for example a screenshot of a sample run of the program)*
-1, 101, 200, -55	Abnormal data	Rejected	
0, 100	Extreme data	Accepted	
-1, 101	Boundary data	Rejected	

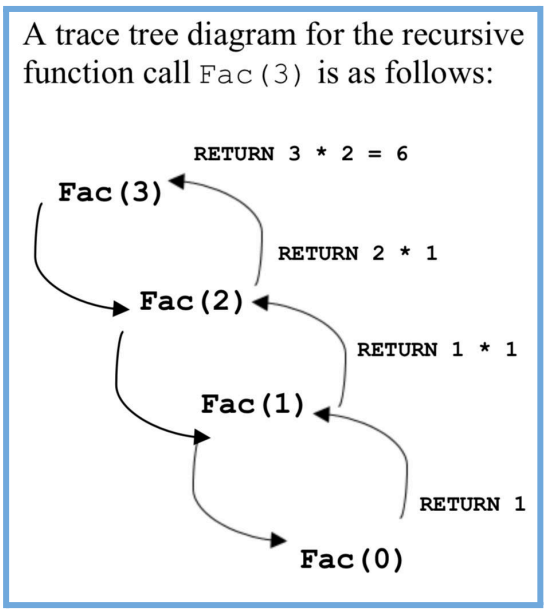
Call stack

- Stores information about active function calls during the program's runtime
- Composed of stack frames, where each frame corresponds to a function call that has not yet completed

    - Each stack frame stores:
        - Return address — the instructions to return to when the function finishes
        - Parameters — the input values passed to the function
            - **Note:** “Parameter” refers to a variable, while “argument” refers to the actual value taken on by the variable when the function is run.
        - Local variables — variables created within the function
- When a function calls another function (or itself, in the case of recursion), a new stack frame (with a return address, parameters and local variables) is pushed onto the top of the stack, with the parameters pushed in reverse order to which they were declared.
- Once a function finishes executing, its stack frame is popped from the stack, and control returns to the stored return address.
- 

- aka machine stack, execution stack, stack

Recursion

- *The process by which a function calls itself as part of its execution*
- A recursive function is defined in terms of itself, and for input values other than the stopping condition, it calls itself.
- A recursive function has 2 key components:
    - Stopping condition(s) (base case) — prevents infinite recursion by ensuring that the function eventually stops calling itself and begins to "unwind" so that it can return a result
        - Must be reached after a finite number of calls
    - A recursive call — where the recursion actually occurs
- In general, but not always, recursive functions replace loops in non-recursive functions.
- If recursion continues indefinitely without reaching a base case (either due to the absence of a base case or the recursion being too deep), the stack will eventually run out of space, resulting in a stack overflow and thus a program crash.
- 

Recursion vs iteration

Feature	Recursion	Iteration
Simplicity	More elegant for problems that naturally break into smaller subproblems (e.g. tree traversal)	Can be more complex for problems that involve recursive patterns
Memory usage	Can use a large amount of memory due to call stack growth, especially for deep recursion	Fixed memory usage; does not use extra memory for function calls
Execution speed	May be slower due to overhead from function calls and maintaining the call stack	Generally faster as it avoids the overhead of recursive function calls
Time complexity	May be less efficient due to overhead from multiple function calls	Generally more time-efficient for simple problems as there is no overhead from function calls
Risk of errors	Risk of stack overflow is the base case is missing or if the recursion depth is too large	No risk of stack overflow; risks are usually related to incorrect loop conditions or logic errors

Ease of debugging	More challenging to trace due to repeated function calls and deeper call stack	Easier to debug since it involves straightforward loop constructs
Proof of correctness	Easier to prove mathematically using induction, especially when the problem follows a recursive structure	Harder to prove (but still possible with loop invariants or inductive reasoning)
Code size	Typically shorter and more elegant, especially for naturally recursive problems	May require more lines of code, especially for problems that can be described recursively
Use cases	Best for problems that can be divided into smaller, similar subproblems (e.g. factorial, Fibonacci, binary search)	Best for repetitive tasks that don't require breaking down problems into smaller subproblems (e.g. summing numbers in a list)

@star.notes