Topic 8: Programming

IGCSE Computer Science 0478 | SenpaiCorner | Mr. Wei

Learning Objectives

8.1 Programming Concepts: variables, constants, basic data types (INTEGER, REAL, CHAR, STRING, BOOLEAN), input/output, sequence, selection (IF/CASE), iteration (FOR, WHILE, REPEAT...UNTIL), string handling, arithmetic/relational/logical operators, nested statements, procedures, functions, parameters, local/global variables, library routines (ROUND, RANDOM), maintainable programs.

8.2 Arrays: 1D and 2D arrays, indices, nested iteration.

8.3 File Handling: OPENFILE, READFILE, WRITEFILE, CLOSEFILE, reading and writing lines of text.

8.1 Programming Concepts

Variables and Constants

A **variable** is a named memory location that temporarily stores data which *can* change while the program runs. A **constant** is a named memory location that stores data which *cannot* change while the program runs; using constants makes code easier to update and more readable. **Pseudocode Example:**

```
DECLARE Score : INTEGER       // variable -- value can change
CONSTANT Pi <- 3.14159        // constant -- value is fixed
```

Variables vs Constants – Declaration Syntax

A variable uses `DECLARE ... : TYPE`. A constant uses `CONSTANT ...<- value`. Reassigning a constant is a logic error; the Cambridge mark scheme treats them as strictly different constructs.

Basic Data Types

Every declared variable must be assigned one of five data types:

Type	Stores	Notes	Example
`INTEGER`	Whole number	Positive or negative; no decimal point	`15`, `-3`
`REAL`	Decimal number	Contains a fractional part	`3.14`, `-0.5`
`CHAR`	One character	Enclosed in *single* quotes	`'A'`, `'%'`
`STRING`	Zero or more characters	Enclosed in *double* quotes	`"Hello"`
`BOOLEAN`	TRUE or FALSE	Only two possible values	`TRUE`

CHAR vs STRING – Quote Style

`CHAR` uses *single* quotes (`'A'`). `STRING` uses *double* quotes (`"Hello"`). Swapping the quote style is a recurring syntax error in exam pseudocode answers.

Input and Output

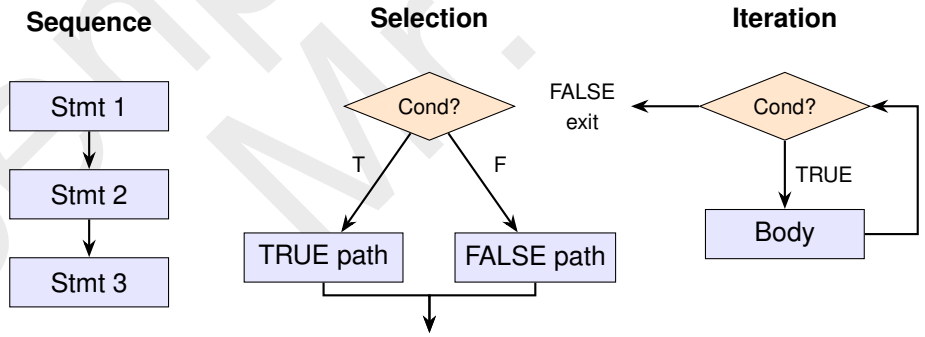
`OUTPUT` displays information to the user. `INPUT` receives data from the user and stores it in a variable.

Pseudocode Example:

```
OUTPUT "Please enter your age:"
INPUT Age
```

Sequence, Selection, and Iteration

These three constructs are the building blocks of all programs.



1. Sequence — statements execute one after another, from top to bottom, in order.

2. Selection — the program branches based on a condition.

- **IF statement:** used for specific conditions; an optional `ELSE` branch handles the alternative path.
- **CASE statement:** cleaner than nested IFs when one variable must be checked against multiple specific values.

Pseudocode Example:

```
// IF Statement
IF Score >= 50 THEN
    OUTPUT "Pass"
ELSE
    OUTPUT "Fail"
ENDIF

// CASE Statement
CASE OF Grade
    'A' : OUTPUT "Excellent"
    'B' : OUTPUT "Good"
    OTHERWISE OUTPUT "Invalid Grade"
ENDCASE
```

3. Iteration (Loops) — repeating a block of code until a condition is met.

- **FOR loop** (count-controlled): repeats a fixed number of times.
- **WHILE loop** (pre-condition): condition checked *before* each iteration — may run 0 times.
- **REPEAT... UNTIL loop** (post-condition): condition checked *after* each iteration — always runs at least once.

Pseudocode Example:

```
// FOR (Count-controlled)
FOR Counter <- 1 TO 10
    OUTPUT Counter
NEXT Counter

// WHILE (Pre-condition)
WHILE Number > 0 DO
    Number <- Number - 1
ENDWHILE

// REPEAT...UNTIL (Post-condition)
REPEAT
    INPUT Password
UNTIL Password = "Secret"
```

Choosing the Right Loop

Use **FOR** when the number of repetitions is known before the loop starts. Use **WHILE** when the condition may already be false on entry (loop may never run). Use **REPEAT... UNTIL** when at least one execution is required — e.g. prompting a user for input.

String Handling

String operations manipulate text values. In Cambridge pseudocode, the first character of a string is at index **1**.

- `LENGTH(string)` — returns the number of characters.
- `SUBSTRING(string, start, length)` — extracts a substring starting at position *start* for *length* characters.
- `UCASE(string)` — converts the entire string to uppercase.
- `LCASE(string)` — converts the entire string to lowercase.

Pseudocode Example:

```
Name <- "Cambridge"
OUTPUT LENGTH(Name)             // outputs 9
OUTPUT SUBSTRING(Name, 1, 3)    // outputs "Cam"
OUTPUT UCASE(Name)              // outputs "CAMBRIDGE"
```

SUBSTRING Third Parameter Is a Length – Not an End Index

The third argument is the *count of characters to extract*, not the end position. `SUBSTRING("Cambridge", 1, 3)` extracts 3 characters from position 1, giving `"Cam"`. Writing the end position (e.g. 3 meaning "stop at index 3") is a very common error.

Operators

Type	Symbols	Notes
Arithmetic	`+ - * / ^`	Standard maths; `^` is power
Integer arithmetic	`MOD DIV`	`DIV` = quotient; `MOD` = remainder
Relational	`= < <= > >= <>`	Returns BOOLEAN; <> means not equal
Logical	`AND OR NOT`	Combines Boolean expressions

DIV vs MOD – Which Is Which

`13 DIV 5 = 2` (how many whole times 5 goes into 13). `13 MOD 5 = 3` (the leftover). Exam questions frequently ask for the `MOD` result and students write the quotient instead — they are not the same.

Nested Statements

Placing one programming construct inside another (e.g. an IF inside a FOR loop) is called **nesting**. The Cambridge syllabus limits exam questions to a maximum of **three levels** of nesting.

Procedures, Functions, and Parameters

Subroutines group code into reusable, named blocks.

- **Procedure:** performs a task; does *not* return a value.

- **Function:** performs a task and *returns a single value* to the caller.
- **Parameters:** variables passed into the subroutine when it is called. The syllabus caps this at a maximum of three parameters.

Pseudocode Example:

```
// Defining a function
FUNCTION CalcArea(Length : INTEGER, Width : INTEGER) RETURNS INTEGER
    Area <- Length * Width
    RETURN Area
ENDFUNCTION

// Calling the function
TotalArea <- CalcArea(5, 10)
```

Variable scope:

- **Local variable:** declared inside a subroutine; exists only while that subroutine runs; invisible to the main program.
- **Global variable:** declared in the main program; accessible and modifiable from anywhere, including inside subroutines.

Scope – Local Variables Are Invisible Outside Their Subroutine

A variable declared inside a `FUNCTION` or `PROCEDURE` is local. The main program cannot read or modify it. Only *global* variables declared at the top of the main program persist across the entire execution.

Library Routines

Built-in functions provided by the language:

- `MOD(a, b)` — remainder of `a` divided by `b`.
- `DIV(a, b)` — integer quotient of `a` divided by `b`.
- `ROUND(identifier, places)` — rounds a REAL number to the specified number of decimal places.
- `RANDOM()` — returns a random REAL number between 0 and 1 inclusive.

Maintainable Programs

A maintainable program is one that other developers (and your future self) can read, debug, and update with minimal effort. The three required techniques are:

- **Meaningful identifiers** — e.g. `PlayerScore` instead of `x`.
- **Comments** — e.g. `// Calculate total including tax` to explain non-obvious logic.

- **Subroutines** — procedures and functions break the program into modular, reusable units.

Maintainability Mark Points – Three Distinct Answers

If asked "how could this program be made more maintainable?" the target answers are meaningful identifiers, comments, and use of procedures/functions. Each is a *separate* mark point. Listing only one earns one mark even on a 3-mark question.

Student Registration Module

A school registration program uses a **constant** for minimum age and **variables** (`STRING`, `INTEGER`) to collect student details via `INPUT` and `OUTPUT`. A **function** (`IsValid`) takes the password as a **parameter**, using **string handling** (`LENGTH`), **relational operators**, and an **IF statement** (**selection**) to return a `BOOLEAN`. The main program uses a **REPEAT...UNTIL** loop (**iteration**) to re-prompt the user until the password is valid. Finally, **meaningful identifiers** and **comments** ensure the code remains **maintainable**. **Pseudocode Example:**

```
CONSTANT MinAge <- 11
DECLARE StudentName, Password : STRING
DECLARE Age : INTEGER
DECLARE Valid : BOOLEAN

// Function to check password security (Subroutine & Local Scope)
FUNCTION IsValid(Pwd : STRING) RETURNS BOOLEAN
    IF LENGTH(Pwd) >= 8 THEN
        RETURN TRUE
    ELSE
        RETURN FALSE
    ENDIF
ENDFUNCTION

// Main sequence (Sequence & I/O)
OUTPUT "Enter student name:"
INPUT StudentName

// Password validation loop (Iteration & Nested Function Call)
REPEAT
    OUTPUT "Enter a password (min 8 chars):"
    INPUT Password
    Valid <- IsValid(Password)
UNTIL Valid = TRUE

OUTPUT "Registration complete for ", UCASE(StudentName)
```

8.2 Arrays

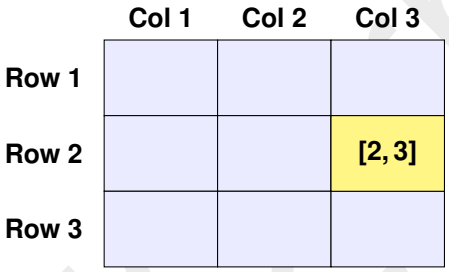
An **array** is a data structure that stores multiple items of data of the *same data type* under a single identifier.

1D and 2D Arrays

- **1D array:** a single list accessed by one index (e.g. `StudentNames[5]`).
- **2D array:** a table/grid accessed by two indices — row then column (e.g. `Grid[2, 3]`).

Indices typically start at **1**. Always define the bounds explicitly in the declaration.

```
DECLARE StudentNames   : ARRAY[1:30] OF STRING
DECLARE TicTacToeBoard : ARRAY[1:3, 1:3] OF CHAR
```



Reading a 2D Array Index

Given `DECLARE Grid : ARRAY[1:3, 1:3] OF INTEGER`: the statement `Grid[2, 3] <- 99` stores 99 in **Row 2, Column 3** (the yellow cell above). To retrieve it: `OUTPUT Grid[2, 3]` prints `99`. The first index is always the row; the second is always the column.

Array Out of Bounds

If declared as `ARRAY[1:30]`, valid indices are **1 through 30 inclusive**. Accessing index 0 or index 31 is an out-of-bounds error. Note that Cambridge pseudocode indices start at 1, not 0.

Reading and Writing Arrays with Iteration

Loops are the standard way to read from or write to arrays efficiently. A 1D array requires one loop; a 2D array requires two *nested* loops — one for rows and one for columns.

Pseudocode Example:

```
// Writing to a 1D array
FOR Index <- 1 TO 30
    StudentNames[Index] <- "Empty"
NEXT Index

// Writing to a 2D array
FOR Row <- 1 TO 3
    FOR Col <- 1 TO 3
        TicTacToeBoard[Row, Col] <- 'X'
    NEXT Col
NEXT Row
```

Nested Loops and 2D Arrays – Counting Iterations

The *outer* loop iterates over rows; the *inner* loop iterates over columns. The inner loop completes *all* its iterations for every single step of the outer loop. For a 3×3 board this means $3 \times 3 = 9$ total iterations of the inner body. Examiners use trace tables to test this understanding.

Student Grades Tracker

A teacher tracks test scores using a **1D array** for student names and a **2D array** for grades (rows = students, columns = subjects). Both use explicit bounds starting at **1** to prevent **out-of-bounds** errors. **Nested loops** efficiently initialize the data: the outer loop sets the name and iterates through students, while the inner loop resets all subject scores for that specific student.

Pseudocode Example:

```
DECLARE Names  : ARRAY[1:30] OF STRING
DECLARE Grades : ARRAY[1:30, 1:5] OF INTEGER
DECLARE Row, Col : INTEGER

FOR Row <- 1 TO 30
    Names[Row] <- "Unknown"
    FOR Col <- 1 TO 5
        Grades[Row, Col] <- 0
    NEXT Col
NEXT Row
```

8.3 File Handling

Data stored in variables and arrays lives in RAM and is **lost** when a program closes. To persist data across runs it must be written to an external file (e.g. a `.txt` file).

File Commands

Command	Purpose
`OPENFILE "name" FOR READ`	Opens an existing file for reading
`OPENFILE "name" FOR WRITE`	Creates a new file or *overwrites* the existing file for writing
`READFILE "name", Variable`	Reads the next line from the file into a variable
`WRITEFILE "name", Value`	Writes a value as a new line in the file
`CLOSEFILE "name"`	Closes the file and flushes all changes — always required

Pseudocode Example:

```
DECLARE LineOfText : STRING

// Writing to a file
OPENFILE "Scores.txt" FOR WRITE
WRITEFILE "Scores.txt", "Player 1: 50"
CLOSEFILE "Scores.txt"

// Reading from a file
OPENFILE "Scores.txt" FOR READ
READFILE "Scores.txt", LineOfText
OUTPUT LineOfText
CLOSEFILE "Scores.txt"
```

WRITE Mode Overwrites – It Does Not Append

Opening a file `FOR WRITE` *creates a new file or replaces the existing one entirely*. All previous contents are destroyed. If retention of existing data is needed, read it first and then rewrite everything including the old data.

Forgetting CLOSEFILE – A Standalone Mark Point

Every `OPENFILE` must be paired with a `CLOSEFILE`. Omitting it causes data corruption and fails to release system resources. Cambridge mark schemes award `CLOSEFILE` as a *separate* mark point — always include it.

Score Logger Application

A game saves the player's score at the end of each round and reloads it on the next startup. The **write phase** opens `"Scores.txt" FOR WRITE`, calls `WRITEFILE` with the score value, then calls `CLOSEFILE`. The **read phase** opens the same file `FOR READ`, calls `READFILE` into a `STRING` variable, outputs it with `OUTPUT`, then calls `CLOSEFILE`. The key constraint: both phases must open and close the file independently — you cannot read and write simultaneously in the same `OPENFILE` block.

Pseudocode Example:

```
// --- Write Phase ---
DECLARE currentScore : INTEGER
currentScore <-- 2500  // Example round score

OPENFILE "Scores.txt" FOR WRITE
WRITEFILE "Scores.txt", currentScore
CLOSEFILE "Scores.txt"

// --- Read Phase ---
DECLARE savedScore : STRING

OPENFILE "Scores.txt" FOR READ
READFILE "Scores.txt", savedScore
OUTPUT "Last saved score was: " & savedScore
CLOSEFILE "Scores.txt"
```