DILUSO

Compiled by Sir Kwembeya

CAMBRIDGE
International Education

Grade thresholds – March 2025

Cambridge IGCSE™ Computer Science (0478)

Grade thresholds taken for Syllabus 0478 (Computer Science) in the March 2025 examination.

		Minimum raw mark required for grade:						
	Maximum raw mark available	A	B	C	D	E	F	G
Component 12	75	49	39	28	24	20	15	10
Component 22	75	55	45	36	30	24	19	14

Grade A* does not exist at the level of an individual component.

The overall thresholds for the different grades were set as follows.

Option	Maximum mark after weighting	Combination of components	A*	A	B	C	D	E	F	G
AY	150	12, 22	124	104	84	64	54	44	34	24

Question 10

Candidates were required to complete an extended program to meet a set of requirements given in a scenario based on adding and outputting membership details of a sports club. The program was to display a menu to allow three options:

- add a new member by entering a new membership code, chosen by the new member, with the program checking to make sure it had a length of six characters and that it had not already been used before.
- display a list of all members showing their names and membership codes.
- end the program.

There was a wide range of quality of responses, with most responses using either pseudocode or Python, but a small number of Java and VB.NET solutions were provided.

The full range of marks was awarded, with a high proportion of candidates achieving marks from the middle or higher mark bands. Stronger responses had closely matched the requirements stated in the scenario, ensuring that all points were fully covered.

Stronger responses had followed the remaining additional guidance at the end of the scenario. This included the comprehensive use of comments to explain the purpose of each part or sub-part of their solution and the use of appropriate messages to accompany all inputs and outputs.

The best responses correctly used all the data structures given in the scenario; in the way they were expected to be used as stated in their descriptions. These were the one-dimensional array `MemberID[]` to store unique membership codes, the two-dimensional array `Name[]` to store members first and last names and the variable `NewID` to allow a new membership code to be input.

0478/22 February/March 2023

11 The one-dimensional (1D) array `TeamName[]` contains the names of teams in a sports league. The two-dimensional (2D) array `TeamPoints[]` contains the points awarded for each match. The position of each team's data in the two arrays is the same. For example, the team stored at index 10 in `TeamName[]` and `TeamPoints[]` is the same.

The variable `LeagueSize` contains the number of teams in the league. The variable `MatchNo` contains the number of matches played. All teams have played the same number of matches.

The arrays and variables have already been set up and the data stored.

Each match can be played at home or away. Points are recorded for the match results of each team with the following values:

- 3 – away win
- 2 – home win
- 1 – drawn match
- 0 – lost match.

Write a program that meets the following requirements:

- calculates the total points for all matches played for each team
- counts the total number of away wins, home wins, drawn matches and lost matches for each team
- outputs for each team:
  - name
  - total points
  - total number of away wins, home wins, drawn matches and lost matches
- finds and outputs the name of the team with the highest total points
- finds and outputs the name of the team with the lowest total points.

You must use pseudocode or program code **and** add comments to explain how your code works.

You do **not** need to declare any arrays, variables or constants; you may assume that this has already been done.

All inputs and outputs must contain suitable messages.

You do **not** need to initialise the data in the arrays `TeamName[]` and `TeamPoints[]` or the variables `LeagueSize` and `MatchNo`

Solution

```
// Loop through each team to calculate total points and match statistics
FOR TeamIndex ← 0 TO LeagueSize - 1
    TotalPoints ← 0
    AwayWins ← 0
    HomeWins ← 0
    Draws ← 0
    Losses ← 0

    // Loop through each match played by the team
    FOR MatchIndex ← 0 TO MatchNo - 1
        Points ← TeamPoints[TeamIndex][MatchIndex]

        // Add to total points
        TotalPoints ← TotalPoints + Points

        // Count match result types
        IF Points = 3 THEN
            AwayWins ← AwayWins + 1
        ELSE IF Points = 2 THEN
            HomeWins ← HomeWins + 1
        ELSE IF Points = 1 THEN
            Draws ← Draws + 1
        ELSE IF Points = 0 THEN
            Losses ← Losses + 1
        ENDIF
    NEXT MatchIndex

    // Store total points for later comparison
    TeamTotalPoints[TeamIndex] ← TotalPoints

    // Output team statistics
    OUTPUT "Team Name: ", TeamName[TeamIndex]
    OUTPUT "Total Points: ", TotalPoints
    OUTPUT "Away Wins: ", AwayWins
    OUTPUT "Home Wins: ", HomeWins
    OUTPUT "Draws: ", Draws
    OUTPUT "Losses: ", Losses
    OUTPUT "----------------------------"
NEXT TeamIndex

// Find team with highest and lowest total points
MaxPoints ← TeamTotalPoints[0]
MinPoints ← TeamTotalPoints[0]
MaxTeam ← TeamName[0]
MinTeam ← TeamName[0]

FOR TeamIndex ← 1 TO LeagueSize - 1
    IF TeamTotalPoints[TeamIndex] > MaxPoints THEN
        MaxPoints ← TeamTotalPoints[TeamIndex]
        MaxTeam ← TeamName[TeamIndex]
    ENDIF

    IF TeamTotalPoints[TeamIndex] < MinPoints THEN
        MinPoints ← TeamTotalPoints[TeamIndex]
        MinTeam ← TeamName[TeamIndex]
    ENDIF
NEXT TeamIndex

// Output highest and lowest scoring teams
```

```
OUTPUT "Team with highest total points: ", MaxTeam, " (", MaxPoints, "
points)"
OUTPUT "Team with lowest total points: ", MinTeam, " (", MinPoints, "
points)"
```

0478/22 May/June 2023

12 A two-dimensional (2D) array `Account[]` contains account holders' names and passwords for a banking program.

A 2D array `AccDetails[]` has three columns containing the following details:

- column one stores the balance – the amount of money in the account, for example 250.00
- column two stores the overdraft limit – the maximum total amount an account holder can borrow from the bank after the account balance reaches 0.00, for example 100.00
- column three stores the withdrawal limit – the amount of money that can be withdrawn at one time, for example 200.00

The amount of money in a bank account can be negative (overdrawn) but **not** by more than the overdraft limit.
For example, an account with an overdraft limit of 100.00 must have a balance that is greater than or equal to –100.00

Suitable error messages must be displayed if a withdrawal cannot take place, for example if the overdraft limit or the size of withdrawal is exceeded.

The bank account ID gives the index of each account holder's data held in the two arrays.
For example, account ID 20's details would be held in:
`Account[20,1]` and `Account[20,2]`
`AccDetails[20,1]` `AccDetails[20,2]` and `AccDetails[20,3]`

The variable `Size` contains the number of accounts.

The arrays and variable `Size` have already been set up and the data stored.

Write a program that meets the following requirements:

- checks the account ID exists and the name and password entered by the account holder match the name and password stored in `Account[]` before any action can take place
- displays a menu showing the four actions available for the account holder to choose from:
    1. display balance
    2. withdraw money
    3. deposit money
    4. exit
- allows an action to be chosen and completed. Each action is completed by a procedure with a parameter of the account ID.

You must use pseudocode or program code **and** add comments to explain how your code works.
All inputs and outputs must contain suitable messages.

You only need to declare any local arrays and local variables that you use.

You do **not** need to declare and initialise the data in the global arrays `Account[]` and `AccDetails[]` and the variable `Size`

Solution:

```
// Prompt user to enter account ID
INPUT "Enter your account ID: " → AccountID

// Check if AccountID is valid
IF AccountID < 0 OR AccountID ≥ Size THEN
    OUTPUT "Error: Invalid account ID."
    STOP
ENDIF

// Prompt for name and password
INPUT "Enter your name: " → InputName
INPUT "Enter your password: " → InputPassword

// Verify credentials
IF Account[AccountID, 1] ≠ InputName OR Account[AccountID, 2] ≠
InputPassword THEN
    OUTPUT "Error: Name or password incorrect."
    STOP
ENDIF

// Display menu and loop until user exits
REPEAT
    OUTPUT "Choose an action:"
    OUTPUT "1. Display balance"
    OUTPUT "2. Withdraw money"
    OUTPUT "3. Deposit money"
    OUTPUT "4. Exit"
    INPUT "Enter your choice (1-4): " → Choice

    IF Choice = 1 THEN
        CALL DisplayBalance(AccountID)
    ELSE IF Choice = 2 THEN
        CALL WithdrawMoney(AccountID)
    ELSE IF Choice = 3 THEN
        CALL DepositMoney(AccountID)
    ELSE IF Choice = 4 THEN
        OUTPUT "Thank you. Goodbye!"
    ELSE
        OUTPUT "Invalid choice. Please select 1-4."
    ENDIF
UNTIL Choice = 4

// Procedure to display balance
PROCEDURE DisplayBalance(ID)
    OUTPUT "Your current balance is: ", AccDetails[ID, 1]
ENDPROCEDURE

// Procedure to withdraw money
PROCEDURE WithdrawMoney(ID)
    INPUT "Enter amount to withdraw: " → Amount

    // Check withdrawal limit
    IF Amount > AccDetails[ID, 3] THEN
        OUTPUT "Error: Withdrawal exceeds limit of ", AccDetails[ID, 3]
        RETURN
    ENDIF

    // Check overdraft limit
    NewBalance ← AccDetails[ID, 1] - Amount
    IF NewBalance < -AccDetails[ID, 2] THEN
        OUTPUT "Error: Withdrawal exceeds overdraft limit."
        RETURN
    ENDIF

    // Update balance
    AccDetails[ID, 1] ← NewBalance
    OUTPUT "Withdrawal successful. New balance: ", NewBalance
ENDPROCEDURE

// Procedure to deposit money
PROCEDURE DepositMoney(ID)
    INPUT "Enter amount to deposit: " → Amount
    AccDetails[ID, 1] ← AccDetails[ID, 1] + Amount
    OUTPUT "Deposit successful. New balance: ", AccDetails[ID, 1]
ENDPROCEDURE
```

0478/22 October/November 2023

11 A wood flooring company stores the names of up to 100 customers in a one-dimensional (1D) array `Customers[]`. A two-dimensional (2D) array `Quotations[]` stores details of each customer's quotation:

- length of room (one decimal place)
- width of room (one decimal place)
- area of wood required (rounded up to next whole number)
- choice of wood index (whole number)
- price of wood required in dollars (two decimal places).

The floor measurements (room length and room width) are taken in metres. All floors are rectangles and room measurements must be between 1.5 and 10.0 inclusive.

The index of any customer's data is the same in both arrays. For example, a customer named in index 4 of `Customers[]` corresponds to the data in index 4 of `Quotations[]`

The wood choices available are:

Index	Wood type	Price per square metre ($)
1	Laminate	29.99
2	Pine	39.99
3	Oak	54.99

The data are stored in two 1D arrays named `WoodType[]` and `Price[]`. The index of the wood type and price in their arrays share the same index number.

Write a program that meets the following requirements:

- input a new customer's name, room length and room width
- check that each measurement is valid
- output an error message and require the measurement to be re-entered until it is valid
- calculate the area of the room by multiplying together the length of the room and the width of the room
- input the choice of wood and find its price per square metre
- calculate the price of the wood needed
- store all data in the relevant array
- output the customer's quotation to include: the name of the customer, the choice of wood and the calculated price of the wood required
- continue to accept the next customer.

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; you may assume that this has already been done.

You will need to initialise `WoodType[]` and `Price[]`

All inputs and outputs must contain suitable messages.

Solution:

```
// Initialize wood types and prices
WoodType[1] ← "Laminate"
WoodType[2] ← "Pine"
WoodType[3] ← "Oak"

Price[1] ← 29.99
Price[2] ← 39.99
Price[3] ← 54.99

CustomerIndex ← 0  // Start index for storing customer data
REPEAT
    // Input customer name
    INPUT "Enter customer name: " → Customers[CustomerIndex]
    // Input and validate room length
    REPEAT
        INPUT "Enter room length (1.5 to 10.0 metres): " → Length
        IF Length < 1.5 OR Length > 10.0 THEN
            OUTPUT "Error: Length must be between 1.5 and 10.0 metres."
        ENDIF
    UNTIL Length ≥ 1.5 AND Length ≤ 10.0
    // Input and validate room width
    REPEAT
        INPUT "Enter room width (1.5 to 10.0 metres): " → Width
        IF Width < 1.5 OR Width > 10.0 THEN
            OUTPUT "Error: Width must be between 1.5 and 10.0 metres."
        ENDIF
    UNTIL Width ≥ 1.5 AND Width ≤ 10.0

    // Calculate area and round up
    Area ← Length * Width
    RoundedArea ← CEILING(Area)

    // Input wood choice
    REPEAT
        INPUT "Enter wood choice (1 = Laminate, 2 = Pine, 3 = Oak): " →
WoodChoice
        IF WoodChoice < 1 OR WoodChoice > 3 THEN
            OUTPUT "Error: Invalid wood choice. Please enter 1, 2, or 3."
        ENDIF
    UNTIL WoodChoice ≥ 1 AND WoodChoice ≤ 3

    // Get price per square metre
    UnitPrice ← Price[WoodChoice]

    // Calculate total price
    TotalPrice ← RoundedArea * UnitPrice
    // Store quotation details
    Quotations[CustomerIndex, 1] ← Length
    Quotations[CustomerIndex, 2] ← Width
    Quotations[CustomerIndex, 3] ← RoundedArea
    Quotations[CustomerIndex, 4] ← WoodChoice
    Quotations[CustomerIndex, 5] ← TotalPrice
    // Output quotation
    OUTPUT "Quotation for ", Customers[CustomerIndex]
    OUTPUT "Wood type: ", WoodType[WoodChoice]
    OUTPUT "Total price: $", TotalPrice

    // Ask if another customer should be entered
    INPUT "Do you want to enter another customer? (Yes/No): " → Response
    CustomerIndex ← CustomerIndex + 1
UNTIL Response = "No"
```

0478/22 February/March 2024

11 Students in a class are recording the amount of time in minutes spent in front of a screen for each day of the week.

The one-dimensional (1D) array `StudentName[]` contains the names of the students in the class.

The two-dimensional (2D) array `ScreenTime[]` is used to input the number of minutes on each day spent in front of a screen.

The position of each student's data in the two arrays is the same. For example, the student stored at index 10 in `StudentName[]` and `ScreenTime[]` is the same.

The variable `ClassSize` contains the number of students in the class.

Write a program that meets these requirements:

- allows all the students to enter their daily minutes of screen times for the past week
- calculates the total number of minutes of screen time for each student in the week
- counts, for each student, the number of days with more than 300 minutes of screen time
- calculates the average weekly minutes of screen time for the whole class
- finds the student with the lowest weekly minutes of screen time
- outputs for each student:
  - name
  - total week's screen time in hours and minutes
  - number of days with more than 300 minutes of screen time
- outputs the average weekly minutes of screen time for the whole class
- outputs the name of the student with the lowest weekly screen time.

You must use pseudocode or program code **and** add comments to explain how your code works. All inputs and outputs must contain suitable messages.

Assume that the array `StudentName[]` and the variable `ClassSize` already contain the required data.

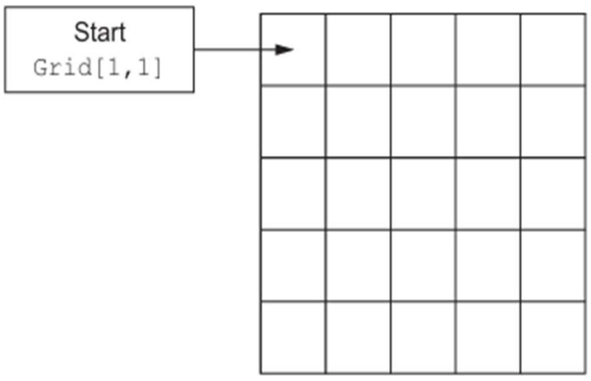
You do **not** need to declare any arrays or variables; you may assume that this has already been done.

Solution:

```
// Initialize variables
TotalClassMinutes ← 0
LowestMinutes ← 999999
LowestStudent ← ""

// Loop through each student
FOR StudentIndex ← 0 TO ClassSize - 1
    WeeklyTotal ← 0
    Over300Days ← 0

    // Input screen time for each day of the week
    FOR Day ← 0 TO 6
        INPUT "Enter screen time in minutes for ",
StudentName[StudentIndex], " on day ", Day + 1, ": " →
ScreenTime[StudentIndex][Day]
        WeeklyTotal ← WeeklyTotal + ScreenTime[StudentIndex][Day]

        // Count days with more than 300 minutes
        IF ScreenTime[StudentIndex][Day] > 300 THEN
            Over300Days ← Over300Days + 1
        ENDIF
    NEXT Day

    // Convert weekly total to hours and minutes
    Hours ← WeeklyTotal DIV 60
    Minutes ← WeeklyTotal MOD 60

    // Output student summary
    OUTPUT "Student: ", StudentName[StudentIndex]
    OUTPUT "Total weekly screen time: ", Hours, " hours and ", Minutes, "
minutes"
    OUTPUT "Days with more than 300 minutes: ", Over300Days
    OUTPUT "------------------------------"

    // Add to class total
    TotalClassMinutes ← TotalClassMinutes + WeeklyTotal

    // Check for lowest screen time
    IF WeeklyTotal < LowestMinutes THEN
        LowestMinutes ← WeeklyTotal
        LowestStudent ← StudentName[StudentIndex]
    ENDIF
NEXT StudentIndex

// Calculate and output class average
AverageMinutes ← TotalClassMinutes DIV ClassSize
OUTPUT "Average weekly screen time for the class: ", AverageMinutes, "
minutes"

// Output student with lowest screen time
OUTPUT "Student with the lowest weekly screen time: ", LowestStudent, " (",
LowestMinutes, " minutes)"
```

0478/22 May/June 2024

11 A one-player game uses the two-dimensional (2D) array `Grid[]` to store the location of a secret cell to be found by the player in 10 moves. Each row and column has 5 cells.



At the start of the game:

- The program places an 'X' in a random cell (**not** in `Grid[1,1]`) and empties all the other cells in the grid.
- The player starts at the top left of the grid.
- The player has 10 moves.

During the game:

- The player can move left, right, up or down by one cell and the move must be within the grid.
- "You Win" is displayed if the player moves to the cell with 'X' and has played 10 moves or less.
- "You Lose" is displayed if the player has made 10 moves without finding the 'X'.

Write a program that meets these requirements.

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; you may assume that this has already been done.

All inputs and outputs must contain suitable messages.

Solution:

```
// Initialize grid size
GridSize ← 5

// Randomly place 'X' in a cell that is not Grid[1][1]
REPEAT
    SecretRow ← RANDOM(1, GridSize)
    SecretCol ← RANDOM(1, GridSize)
UNTIL SecretRow ≠ 1 OR SecretCol ≠ 1

// Initialize grid with empty cells
FOR Row ← 1 TO GridSize
    FOR Col ← 1 TO GridSize
        Grid[Row][Col] ← ""
    NEXT Col
NEXT Row
```

```
// Place 'X' in the secret cell
Grid[SecretRow][SecretCol] ← "X"

// Set starting position
PlayerRow ← 1
PlayerCol ← 1
Moves ← 0
Found ← FALSE

// Game loop: allow up to 10 moves
WHILE Moves < 10 AND Found = FALSE
    OUTPUT "Move ", Moves + 1, ": You are at position (", PlayerRow, ",",
PlayerCol, ")"
    INPUT "Enter move direction (up/down/left/right): " → Direction

    // Update position based on direction
    IF Direction = "up" THEN
        IF PlayerRow > 1 THEN
            PlayerRow ← PlayerRow - 1
        ELSE
            OUTPUT "Invalid move: You can't move up."
        ENDIF
    ELSE IF Direction = "down" THEN
        IF PlayerRow < GridSize THEN
            PlayerRow ← PlayerRow + 1
        ELSE
            OUTPUT "Invalid move: You can't move down."
        ENDIF
    ELSE IF Direction = "left" THEN
        IF PlayerCol > 1 THEN
            PlayerCol ← PlayerCol - 1
        ELSE
            OUTPUT "Invalid move: You can't move left."
        ENDIF
    ELSE IF Direction = "right" THEN
        IF PlayerCol < GridSize THEN
            PlayerCol ← PlayerCol + 1
        ELSE
            OUTPUT "Invalid move: You can't move right."
        ENDIF
    ELSE
        OUTPUT "Invalid input. Please enter up, down, left, or right."
        CONTINUE
    ENDIF

    // Check if player found the secret cell
    IF Grid[PlayerRow][PlayerCol] = "X" THEN
        Found ← TRUE
        OUTPUT "You Win! You found the secret cell in ", Moves + 1, "
moves."
    ENDIF

    // Increment move count
    Moves ← Moves + 1
ENDWHILE

// If player didn't find the cell in 10 moves
IF Found = FALSE THEN
    OUTPUT "You Lose! You did not find the secret cell in 10 moves."
ENDIF
```

0478/22 October/November 2024:

12 A one-dimensional (1D) array `Rooms[]` contains the names of up to 20 rooms in a house.
A two-dimensional (2D) array `Dimensions[]` is used to store the length, width and area of each room.

The position of any room's data is the same in both arrays. For example, the data in index 5 of `Dimensions[]` belongs to the room in index 5 of `Rooms[]`

The variable `Number` stores the number of rooms for which data is to be input. There must be at least 3 rooms but no more than 20.

Write a program that meets the following requirements:

- allows the number of rooms for which data is required to be input, stored and validated
- allows the name of the room and the length and width of the room, in metres, to be entered and stored
- allows the area of each room to be calculated as length multiplied by width and stored as square metres rounded to two decimal places
- calculates the average size of all the rooms by area, in square metres, rounded to two decimal places
- finds the largest room and smallest room by area
- outputs the names of all rooms with their dimensions and area
- outputs the names of the largest room and smallest room by area
- outputs the total area of the house and the average size of all the rooms by area.

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; you may assume that this has already been done.

All inputs and outputs must contain suitable messages.

Solution:

```
// Input and validate number of rooms
REPEAT
    INPUT "Enter the number of rooms (between 3 and 20): " → Number
    IF Number < 3 OR Number > 20 THEN
        OUTPUT "Error: You must enter a number between 3 and 20."
    ENDIF
UNTIL Number ≥ 3 AND Number ≤ 20

TotalArea ← 0
LargestArea ← -1
SmallestArea ← 999999
LargestRoom ← ""
SmallestRoom ← ""

// Input room data and calculate area
FOR Index ← 0 TO Number - 1
    INPUT "Enter name of room " + (Index + 1) + ": " → Rooms[Index]
    INPUT "Enter length of " + Rooms[Index] + " in meters: " → Length
    INPUT "Enter width of " + Rooms[Index] + " in meters: " → Width

    Area ← Length * Width
    Area ← ROUND(Area, 2)  // Round to two decimal places

    Dimensions[Index][0] ← Length
    Dimensions[Index][1] ← Width
    Dimensions[Index][2] ← Area

    TotalArea ← TotalArea + Area

    // Check for largest room
    IF Area > LargestArea THEN
        LargestArea ← Area
        LargestRoom ← Rooms[Index]
    ENDIF

    // Check for smallest room
    IF Area < SmallestArea THEN
        SmallestArea ← Area
        SmallestRoom ← Rooms[Index]
    ENDIF
NEXT Index
// Calculate average area
AverageArea ← ROUND(TotalArea / Number, 2)

// Output room details
OUTPUT "Room Details:"
FOR Index ← 0 TO Number - 1
    OUTPUT "Room: ", Rooms[Index]
    OUTPUT "Length: ", Dimensions[Index][0], " m"
    OUTPUT "Width: ", Dimensions[Index][1], " m"
    OUTPUT "Area: ", Dimensions[Index][2], " square meters"
    OUTPUT "-------------------------------"
NEXT Index

// Output summary
OUTPUT "Largest room: ", LargestRoom, " with area ", LargestArea, " square
meters"
OUTPUT "Smallest room: ", SmallestRoom, " with area ", SmallestArea, "
square meters"
OUTPUT "Total area of the house: ", TotalArea, " square meters"
OUTPUT "Average room size: ", AverageArea, " square meters"
```

0478/22 February/March 2025

10 A sports club uses a six-character alphanumeric membership code to identify each member of the club.

The one-dimensional (1D) array `MemberID[]` is used to store the unique membership codes for club members.

The two-dimensional (2D) array `Name[]` is used to store the names of the club members. The first and last name of each member will be stored in separate array elements.

The system can store details for a maximum of 1000 members.

The position of any member's data is the same in both arrays. For example, the data in index 2 of `MemberID[]` belongs to the member in index 2 of `Name[]`

The variable `NewID` is used to input a new membership code.

Write a program that meets the following requirements:

- Provide a menu that offers the choices: inputting a new member, outputting a list of membership codes and first and last names, or stopping.
- Input and validate a response to the menu.
- When inputting a new member, input a new membership code and check that it contains six characters:
  - If the new code is six characters, check it against all the previously stored membership codes to make sure it is unique.
  - If the code is **not** unique, a new code must be entered and checked.
  - If the code is unique, it is stored in the first available space in the appropriate array and the new member is required to enter their first name and last name, which are also stored in the corresponding location of the appropriate array.
- When outputting a list of membership codes and names, output for each member: their membership code, first name and last name.
- The program will continue until the stop option on the menu is selected.

You must use pseudocode or program code **and** add comments to explain how your code works.

You do **not** need to declare any arrays, variables or constants; assume this has already been done.

You do **not** need to initialise the data in the arrays.

You do need to initialise any variables or constants used if appropriate.

All inputs and outputs must contain suitable messages.

Solution:

```
// Initialize variables
Count ← 0  // Number of members currently stored
Choice ← ""  // Menu choice

// Main program loop
REPEAT
    // Display menu
    OUTPUT "Menu:"
    OUTPUT "1. Input new member"
    OUTPUT "2. Output list of members"
    OUTPUT "3. Stop"
    INPUT "Enter your choice (1-3): " → Choice
    // Validate menu choice
    WHILE Choice ≠ "1" AND Choice ≠ "2" AND Choice ≠ "3"
        OUTPUT "Invalid choice. Please enter 1, 2, or 3."
        INPUT "Enter your choice (1-3): " → Choice
    ENDWHILE

    IF Choice = "1" THEN
        // Input new member
        IF Count = 1000 THEN
            OUTPUT "Maximum number of members reached. Cannot add more."
        ELSE
            REPEAT
                INPUT "Enter a new 6-character membership code: " → NewID
                IF LENGTH(NewID) ≠ 6 THEN
                    OUTPUT "Error: Membership code must be exactly 6
characters."
                ELSE
                    // Check for uniqueness
                    Unique ← TRUE
                    FOR Index ← 0 TO Count - 1
                        IF MemberID[Index] = NewID THEN
                            Unique ← FALSE
                            OUTPUT "Error: Membership code already exists.
Enter a different code."
                            BREAK
                        ENDIF
                    NEXT Index
                ENDIF
            UNTIL LENGTH(NewID) = 6 AND Unique = TRUE

            // Store membership code
            MemberID[Count] ← NewID

            // Input and store names
            INPUT "Enter first name: " → Name[Count][0]
            INPUT "Enter last name: " → Name[Count][1]

            OUTPUT "Member successfully added."
            Count ← Count + 1
        ENDIF

    ELSE IF Choice = "2" THEN
        // Output list of members
        IF Count = 0 THEN
            OUTPUT "No members to display."
        ELSE
            OUTPUT "List of Members:"
            FOR Index ← 0 TO Count - 1
                OUTPUT "Membership Code: ", MemberID[Index]
                OUTPUT "First Name: ", Name[Index][0]
                OUTPUT "Last Name: ", Name[Index][1]
                OUTPUT "------------------------------"
            NEXT Index
        ENDIF
    ENDIF

UNTIL Choice = "3"

OUTPUT "Program stopped. Goodbye!"
```

THE END