Temasek Junior College

2023 JC2 H2 Computing

Programming Constructs

Section	1	Algorithms and Data Structures
Unit	1.1	Algorithmic Representation
Objectives	1.1.1 1.1.3	Use appropriate techniques or tools such as pseudo-code and flowchart to show program flow. Use a combination of various control structures.

Section	2	Programming
Unit	2.2	Programming Elements and Constructs
Objectives	2.2.3	Apply the fundamental programming constructs to control the flow of program execution: - Sequence - Selection - Iteration

1 Introduction

To design code that can compare and manipulate data, and to make decisions, programmers use different logical, relational and arithmetic operators via the programming constructs of **sequence**, **selection** and **iteration**. These constructs are key 'building blocks' in almost all programs and can be assembled in different ways to achieve the goal(s) of a program.

2 Sequence

The difference between a computer and a human is that a computer will do exactly as instructed. It will not misinterpret an instruction or "figure it out" like a human would. If a computer does something unexpected, it is probably due to its human programmer(s) not supplying sufficiently clear and precise instructions in the form of code. Hence computers need precise instructions. Equally important, these instructions must be given in the correct order.

Program code statements written one after another as a precise set of instructions to complete a (sub)-task is called a **sequence**. During program runtime, these statements will be executed in the order that they were written in.

3 Selection

There will be times when a program needs to execute a selected portion of the code only when certain conditions are met. Take for example a login page. An error message is to be displayed only when the user enters an invalid username and password pair. If the username and password pair is correct, an alternative sequence of code that directs the user to the landing page will be executed. In such cases, the **selection** programming construct should be used.

Selection is a programming construct that allows a program to run a specific sequence of code depending on whether a condition evaluates to `True` or `False`.

3.1 Selection Using IF-THEN and IF-THEN-ELSE Statements

`IF-THEN and IF-THEN-ELSE` statements are used to implement the selection programming construct.

<u>Example 1</u>
Write down the output of the program containing the following code snippet.

```
DECLARE test_score : INTEGER
test_score ← 71
IF test_score >= 60 THEN  // Start of selection construct
    OUTPUT "Pass"
ENDIF                     // End of selection construct
OUTPUT "End of program"
```

[Solution]

```
Pass
End of program
```

<u>Notes</u>

- In the above code snippet, the expression `test_score >= 60` is evaluated. If the expression evaluates to `True`, then the program will output `Pass`.

- The final line of code `OUTPUT "End of program"` is outside of the selection programming construct. It will thus execute regardless of whether `test_score >= 60` evaluates to `True` or `False`.

- Observe that the code is **written sequentially** to:
  - first assign the value of 71 to the variable `test_score`;
  - then evaluate the expression `test_score >= 60` to decide whether to output `Pass`;
  - lastly, output `End of Program`.

  This itself demonstrates the **sequence** programming construct, where the code statements are written one after another as a precise set of instructions.

In **Example 1**, no code has been written to specify the action(s) to take when the expression `test_score >= 60` evaluates to `False`. In this case, the execution will continue with the first statement that comes after the `IF` statement block.

The use of `ELSE` will allow the specification of an additional sequence of code statements to be executed when `test_score >= 60` evaluates to `False`.

<u>Example 2</u>
Write down the output of the program containing the following code snippet for the test cases of 70, 10 and 60.

```
DECLARE test_score : INTEGER

OUTPUT "Enter a test score"
INPUT test_score

IF test_score >= 60 THEN          // Start of selection construct
    OUTPUT "Pass"
ELSE
    OUTPUT "Room for improvement"
ENDIF                             // End of selection construct

OUTPUT "End of program"
```

<u>[Solution]</u>

`test_score`	Output
70	`Pass` `End of program`
10	`Room for improvement` `End of program`
60	`Pass` `End of program`

<u>Notes</u>
- In the above code snippet, an `ELSE` statement block is written after the `IF` statement block to output `"Room for improvement"`.

- When the expression `test_score >= 60` is evaluated to `False`, the execution of the code jumps immediately to the `ELSE` statement block, bypassing the code written within the `IF` statement block.

3.2 Multiple Conditions and Nested Selection

Nested selection is used to implement branching logic. This means that if a condition of a main selection block evaluates to `True`, then it leads to sub-conditions (in the form of nested `IF` statements), which are included inside the main condition.

Example 3

Study the code snippet below:

```
01 DECLARE age : INTEGER
02 DECLARE salary : INTEGER
03 DECLARE guarantor : STRING
04
05 OUTPUT "Enter your age: "
06 INPUT age
07
08 IF age >= 21 THEN                      // Main condition
09
10     OUTPUT "Enter your salary: "
11     INPUT salary
12
13     IF salary > 15000 THEN             // Start nested selection
14         OUTPUT "You can rent the flat"
15     ELSE
16         OUTPUT "Not enough for rental"
17     ENDIF                              // End nested selection
18
19 ELSE
20
21     IF age >= 18 AND age < 21 THEN     // Main condition
22
23         OUTPUT "Any guarantor (Y/N):"
24         INPUT guarantor
25
26         IF guarantor = "Y" THEN        // Start nested selection
27             OUTPUT "Contact guarantor"
28         ELSE
29             OUTPUT "Guarantor needed"
30         ENDIF                          // End nested selection
31
32     ELSE                               // Main condition
33
34         OUTPUT "Can’t rent the flat"
35
36     ENDIF
37
38 ENDIF
```

Identify the lines which contain the main branch of the selection programming construct and the lines that contain nested selection. Give a short description of what each of these lines means.

[Solution]

Line `08` – Is age 21 or over?

…

Line `13` – Is salary over 15000?

…

Line `15` – No.

…

Line `19` – No.

…

Line `21` – Is age 18 or above but below 21?

…

Line `26` – Is guarantor available?

…

Line `28` – No.

…

Line `32` – No

…

Notes

- In pseudocode, not all nested `IF-THEN and IF-THEN-ELSE` statements belong to sub-conditions. Some may belong to the main logic branch. In line `21`, although the `IF` statement is nested within the `ELSE` in line `19`, it is part of the main logic branch that checks the age of the user, which is the main condition that determines the action the program will take i.e.
  - get user to input salary (age 21 or above) or
  - check availability of guarantor (below age 21 but age 18 or above) or
  - inform the user that he/she cannot rent a flat (below age 18).

- For multiple conditions belonging to the same logic branch, each subsequent condition will be nested within an `ELSE` statement e.g.

```
IF condition_1
    …
ELSE
    IF condition_2
        …
    ELSE
        IF condition_3
            …
        ELSE
            IF condition_4
                …
            ELSE
                …
            ENDIF
        ENDIF
    ENDIF
ENDIF
```

3.3 Selection Using `CASE` Statements

`CASE` statements can be used as an alternative method for writing selection programming constructs when handling a lot of related options.

The use of `CASE` statements can make the code clearer for another programmer to follow.

<u>Example 4</u>
Rewrite the following selection programming construct using `CASE` statements.

```
IF error_code = "400" THEN
    OUTPUT "Bad request"
ELSE
    IF error_code = "401" THEN
        OUTPUT "Unauthorised request"
    ELSE
        IF error_code = "403" THEN
            OUTPUT "Forbidden request"
        ELSE
            IF error_code = "404" THEN
                OUTPUT "Page not found"
                OUTPUT "Are you sure of the URL?"
            ELSE
                IF error_code = "408" THEN
                    OUTPUT "Timeout error"
                    OUTPUT "Unable to process request"
                ELSE
                    OUTPUT "Unknown error"
                    CALL Reload
                ENDIF
            ENDIF
        ENDIF
    ENDIF
ENDIF
```

[Solution]

```
CASE OF error_code
    400       : OUTPUT "Bad request"
    401       : OUTPUT "Unauthorised request"
    403       : OUTPUT "Forbidden request"
    404       : OUTPUT "Page not found"
                OUTPUT "Are you sure of the URL?"
    408       : OUTPUT "Timeout error"
                OUTPUT "Unable to process request"
    OTHERWISE : OUTPUT "Unknown error"
                CALL Reload
ENDCASE
```

Notes

- The values specified in the `CASE` statement are compared in the order which they are written until there is a match.

- The action(s) specified for the matched value is (are) executed and the `CASE` code block terminates, bypassing any remaining cases. Thereafter, the code statement immediately after the `CASE` block will be executed.

- An `OTHERWISE` statement can be used to handle actions that do not fall under any of the specified cases.

- In **Example 4**, the cases being considered are single values. In fact, each case can also consist of a set of values or a range of values e.g.

```
DECLARE days : INTEGER
DECLARE month_number: INTEGER

OUTPUT "Enter month"
INPUT month_number

CASE OF month_number
    1,3,5,7,8,10,12 : days ← 31   // set of values
    4,6,9,11        : days ← 30   // set of values
    2               : days ← 28   // single value
ENDCASE
```

and

```
DECLARE score : INTEGER
DECLARE pass_fail : STRING

OUTPUT "Enter score"
INPUT score

CASE OF score
    45 to 100 : pass_fail ← "pass"      // range of values
    40 to 44  : pass_fail ← "sub-pass"  // range of values
    0 to 39   : pass_fail ← "fail"      // range of values
ENDCASE

OUTPUT pass_fail
```

Technical Note
For 9569 H2 Computing, the version of Python used is 3.6.4. `CASE` statements need to be written using the `if-elif-else` conditionals.

Beyond the confines of the syllabus, Python 3.10 and above comes with the `match` keyword, which allows for code to be written in a manner similar to `CASE` statements. See https://docs.python.org/3/tutorial/controlflow.html#match-statements

4 Iteration

What happens when a process needs to be repeatedly executed within a program? Must the code defining the process be written repeatedly? Fortunately, repetitive coding can be avoided with the use of the **iteration programming construct**, where the code to be repeated can be specified within a **loop**. This can be for a certain number of repetitions or controlled by a condition at the start or end of the **loop**.

> **Iteration** is the repeated execution of a block of code until a certain condition is satisfied. Each repetition is considered as one iteration.

There are two main types of **loop**:

1) **Count-controlled loop** (also known as **definite loop**)
   - A block of code is repeated a specified number of times e.g. `FOR` loops.

2) **Condition-controlled loop** (also known as **indefinite loop**)
   - A block of code is repeated based on whether a specified condition evaluates to `True` or `False` e.g. `WHILE`, `DO-WHILE` and `REPEAT-UNTIL` loops.

4.1 Count-Controlled Loops

4.1.1 `FOR` Loops

`FOR` loops are count-controlled i.e. the number of times to repeat a block of code written within a `FOR` loop needs to be explicitly specified.

Example 5
Write a count-controlled loop for displaying the integers 1 to 4 sequentially.

[Solution]

```
DECLARE count : INTEGER

FOR count ← 1 TO 4  // upper limit will be included
    OUTPUT count
NEXT count
```

Notes
- The output will be as follows:
```
1
2
3
4
```

- `count` is initialised as `1`. The sequence is stepped through in intervals of size `1` and the code in the loop is repeated four times. There is no fifth execution since the value of `5` exceeds the specified upper limit of assignment `4`.

- In pseudocode, the upper limit value of `4` is included in the assignment as opposed to that of Python. To get the same output in Python, use

```
for count in range(1, 5):  # upper limit not included
    print(count)
```

A `FOR` loop can also be used to iterate through individual items of a collection via indexing. The `LENGTH` keyword can be used to determine the maximum value of the index.

<u>Example 6</u>
Write a count-controlled loop to display the characters of the word "Python"

[Solution]

```
DECLARE word : STRING
DECLARE index : INTEGER

word ← "Python"

FOR index ← 1 TO LENGTH(word)
    OUTPUT word[index]
NEXT index
```

<u>Notes</u>
- The output is as follows:

```
P
y
t
h
o
n
```

- In some texts, the pseudocode `.LENGTH()` is used instead e.g. `word.LENGTH()`.

<u>Example 7</u>
Write a count-controlled loop to display the items in the following array:

```
subjects ← ["Computing", "Math", "Physics", "Econs", "GP", "PW"]
```

[Solution]

```
DECLARE subjects : ARRAY[1 : 6] OF STRING
DECLARE index : INTEGER

subjects ← ["Computing", "Math", "Physics", "Econs", "GP", "PW"]

FOR index ← 1 TO LENGTH(subjects)
    OUTPUT subjects[index]
NEXT index
```

<u>Technical Note</u>
Python is a zero-based language. Hence the index of the first item in a collection is `0`.

As a rule of thumb, pseudocode is usually one-based. The index of the first item in a collect is thus `1`.

Despite so, always study a given piece of pseudocode carefully to determine if it is one-based or zero-based.

4.1.2 Stepping through a Sequence

There may be times when the increment (or decrement) needs to be of a different magnitude other than `1`. In this case, a **step value** needs to be specified.

Example 8

Write a count-controlled loop to display the multiples of 3 within 3 inclusive and 10 inclusive.

[Solution]

```
DECLARE i : INTEGER

FOR i ← 3 to 10 STEP 3
    OUTPUT i
NEXT i
```

Notes

- The output will be as follows:
  ```
  3
  6
  9
  ```

- `i` is initialised as `3`. The sequence is stepped through at intervals of `3` and the code in the loop is repeated thrice. There is no fourth execution as the value of `12` exceeds the specified upper limit of assignment `10`.

Example 9

Write a count-controlled loop to display all even numbers from 1 to 10 in descending order.

[Solution]

```
DECLARE i : INTEGER

FOR i ← 1 to 10 STEP -2
    OUTPUT i
NEXT i
```

Notes

- For a sequence in descending order, use a negative step value. The range of values to be assigned to the counting variable (`i` in this case) in the `FOR` loop should still be specified in ascending order (`1` to `10` in this case, not `10` to `1`).

- The output will be as follows:
  ```
  10
  8
  6
  4
  2
  ```

- `i` is initialised as `10`. The sequence is stepped through at intervals of `-2` and the code in the loop is repeated five times. There is no sixth execution as the value of `0` is smaller than the specified lower limit of assignment `1`.

4.2 Condition-Controlled Loops

Instead of pre-defining the number of repetitions explicitly, a process can also be repeated for as long as a condition is met, or until a condition is met. The number of repetitions to run the code is thus dependent on the number of times it takes for the condition to evaluate to `True`.

In this case, the code to be repeated will have to be placed in a **condition-controlled loop**.

4.2.1 Pre-Condition-Controlled Loops

In a **pre-condition-controlled loop**, the condition will be evaluated at the start of the loop. The code placed inside the loop will be executed only when the condition is evaluated to `True`.

4.2.1.1 `WHILE` Loops

The `WHILE` loop can be used to set-up a pre-condition-controlled loop. When doing so, an **initial value** must **always** be assigned to the variable before the `WHILE` loop.

Example 10
Write a pre-condition-controlled loop to check whether the password input by a user is correct. The user will be prompted to re-enter the password until it is correct.

[Solution]

```
CONSTANT password = "1234"

DECLARE password_guess : STRING

OUTPUT "Enter your password"
INPUT password_guess          // assign user input as initial value

WHILE password_guess <> password
    OUTPUT "Incorrect password, please enter your password again"
    INPUT password_guess
ENDWHILE

OUTPUT "Welcome to the site"
```

Notes
- The first user input is assigned as the initial value of `password_guess`.

- The code in the `WHILE` loop will be executed when the values of `password_guess` and `password` do not match. The eventual number of repetitions will be dependent on the number of times the user takes to enter the correct password.

- Once the condition is met, the code written within the `WHILE` loop is bypassed and the next line of code after the loop will be executed. In this case the message `"Welcome to the site"` will be displayed upon a match between `password` and `password_guess`.

Example 11
Write a pre-condition-controlled loop to perform a countdown from 5 to 0.

[Solution]

```
DECLARE i : INTEGER

i ← 5

WHILE i >= 0
    OUTPUT i
    i ← i - 1
ENDWHILE
```

Notes

- The output is as follows:
```
5
4
3
2
1
0
```

- The value of `5` is assigned as the initial value of `i`.

- The code in the `WHILE` loop will be executed as long as `i >= 0`. For each iteration, the value of `i` is decreased by 1 until `i = -1`, where the loop is terminated. Hence the code in loop will be executed six times.

Technical Note
In pseudocode, changes in values are written as an assignment statement in the form
- `i ← i + 1`
- `i ← i - 2`
- `i ← i * 3`

etc.

This is as opposed to Python where they are written in the form
- `i = i + 1` or `i += 1`
- `i = i - 2` or `i -= 2`
- `i = i * 3` or `i *= 2`

etc.

In some texts, a `WHILE` loop is also known as a `WHILE-DO` loop. There is no difference in syntax except for the additional `DO` keyword placed after the condition e.g.

```
DECLARE i : INTEGER

i ← 5

WHILE i >= 0 DO  // A DO keyword is placed after the condition
    OUTPUT i
    i ← i - 1
ENDWHILE
```

4.2.1.2 Flags

It is important that a while loop always terminates.

Besides including the condition in the `WHILE` statement to trigger termination when it is no longer met or when it is first met, a **flag** can also be used.

A **flag** is typically a Boolean variable than can take on one of two values: `True` or `False`. It is often used to indicate whether a certain condition or an event has occurred, or to trigger a specific action when being set to a certain value.

<u>Example 12</u>
Write a pre-condition-controlled loop with a flag to indicate that counting down from 5 to 0 has been completed.

[Solution]

```
01 DECLARE i : INTEGER
02 DECLARE countdown_complete : BOOLEAN
03
04 i ← 5
05 countdown_complete ← False
06
07 WHILE NOT countdown_complete
08
09     OUTPUT i
10     i ← i - 1
11
12     IF i = -1 THEN
13         countdown_complete ← True
14     ENDIF
15
16 ENDWHILE
```

<u>Notes</u>

- `countdown_complete` is being used as a flag to indicate the completion of the counting down process. Intuitively, it is assigned an initial value of `False`, since counting down has yet to commence.

- Within the `WHILE` loop, an `IF` statement checks the value of `i`. When `i = -1`, countdown has successfully reached `0` and completed. The value of `countdown_complete` is reassigned as `True` to indicate that the counting down process has completed.

- In line `07`, the statement used to commence the loop is `WHILE NOT countdown_complete`.
    - o When `countdown_complete = False`, the statement will translate to `WHILE NOT False` i.e. `WHILE True` and the code in the loop executes.
    - o Conversely, when `countdown_complete = True`, the statement will translate to `WHILE NOT True` i.e. `WHILE False`, and the loop terminates.

- While the use of a flag might seem rather trivial at this juncture (since the loop can be terminated by just having the actual condition written in the `WHILE` statement), the use of flags has the potential to improve the efficiency and readability of codes, especially in the more complex and complicated ones.

4.2.1.3 Sentinel Values

Termination of a while loop can also be done so with the use of a **sentinel value**.

A sentinel value is a special value typically chosen to be distinct from the normal values expected. They can be used to mark the end of a sequence of data, indicate a boundary condition, represent special conditions or error states etc.

The choice of what sentinel value to choose is the decision of the programmer, but it should be a value that is unlikely to be part of the sequence, condition or process under consideration.

<u>Example 13</u>
Write a pre-condition-controlled loop that allows entry of test scores repeatedly until `-1` is entered as a sentinel value to terminate the loop.

[Solution]

```
DECLARE mark : INTEGER

OUTPUT "Enter a mark or -1 to terminate entry"
INPUT mark

WHILE mark <> -1
    OUTPUT "Enter a mark or -1 to terminate entry"
    INPUT mark
ENDWHILE
```

<u>Notes</u>
- The program will repeatedly request for user input until the sentinel value of `-1` is entered.

4.2.2 Post-Condition-Controlled Loops

The code in a **post-condition-controlled loop** is repeated as long as the condition evaluates to `True`. As opposed to a pre-condition-controlled loop, the condition is evaluated at the end of the loop instead of the start of the loop. It follows that the code in the loop will **run at least once**, before the condition is evaluated the first time.

4.2.2.1 `DO-WHILE` Loops

A `DO-WHILE` loop is an example of a post-condition-controlled loop.

In a `DO-WHILE` loop, the code in the loop will be executed **at least once**. Execution is then repeated for as many times as the condition at the end of the loop evaluates to `True`.

<u>Example 14</u>
Rewrite the solution to **Example 11** using a post-condition-controlled `DO-WHILE` loop.

[Solution]

```
DECLARE i : INTEGER

i ← 5

DO
    OUTPUT i
    i ← i - 1
WHILE i >= 0
```

<u>Example 15</u>
Given that `i` was initialised as `-2` instead in **Example 14**, write down the output.

[Solution]

```
-2
```

<u>Notes</u>

- The code will now look as follows

  ```
  DECLARE i : INTEGER

  i ← -2

  DO
      OUTPUT i
      i ← i - 1
  WHILE i >= 0
  ```

- Even though the value of `-2` is smaller than that of `0` i.e. the condition `i >= 0` will evaluate to `False` initially, the code in the loop will still run once as the condition is checked only at the end of the loop (`DO` first… `WHILE` condition evaluates to `True`, `DO` again…, otherwise terminate).

4.2.2.2 `REPEAT-UNTIL` Loops

In a `REPEAT-UNTIL` loop, the code in the loop will be executed **at least once** and repeated until the condition at the end of the loop is met. This is as opposed to a `DO-WHILE` loop, where repetition only occurs when the condition at the end of the loop evaluates to `True`.

Example 16
Re-write the `DO-WHILE` loop in **Example 14** using a `REPEAT-UNTIL` loop.

[Solution]

```
DECLARE i : INTEGER

i ← 5

REPEAT
    OUTPUT i
    i ← i - 1
UNTIL i < 0
```

Notes
- When `i` decreases to `-1`, the condition `i < 0` is finally met and the loop terminates.

Example 17
Rewrite the solution to **Example 10** using a post-condition-controlled `REPEAT-UNTIL` loop.

[Solution]

```
CONSTANT password = "1234"

DECLARE password_guess : STRING

REPEAT
    OUTPUT "Enter your password"
    INPUT password_guess
UNTIL password_guess = password

OUTPUT "Welcome to the site"
```

Technical Note
Python does not have built-in `DO-WHILE` and `REPEAT-UNTIL` loops. Nevertheless, such a structure can be mimic using a `while True` with a nested `if…break` statement e.g.

```
i = 5

while True:

  print(i)
  i = i - 1

  if i < 0:
      break
```

This will allow the code in the loop to run at least once.

4.2.2.3 Use of Flags and Sentinel Values in Post-Condition-Controlled Loops

As with the pre-condition-controlled loops, flags and sentinel values can also be used in post-condition-controlled loops.

Example 18

Re-write the code in **Example 12** using a `REPEAT-UNTIL` loop.

[Solution]

```
DECLARE i : INTEGER
DECLARE countdown_complete : BOOLEAN
i ← 5
countdown_complete ← False
REPEAT
    OUTPUT i
    i ← i - 1
    IF i = -1 THEN
        countdown_complete ← True
    ENDIF
UNTIL countdown_complete ← True
```

Notes

- Compare the code above with that in **Example 16**. Both are `REPEAT-UNTIL` loops performing the same purpose. While the use of a flag might seem rather trivial at this juncture (since the loop can be terminated by just having the actual condition written in the `UNTIL` statement), the use of flags has the potential to improve the efficiency and readability of codes, especially in the more complex and complicated ones.

Example 19

Re-write the code in **Example 13** using a `DO-WHILE` loop.

[Solution]

```
DECLARE mark : INTEGER
DO
    OUTPUT "Enter a mark or -1 to terminate entry"
    INPUT mark
WHILE mark <> -1
```

Notes

- For this instance, the value of `-1` is used as the sentinel value for terminating the loop.

4.2.3 Nested Loops

The nesting of loops is a common programming concept which involves using one loop inside another loop. Nested loops are very useful for iterating over the contents of multi-dimensional data structures such as nested lists and two-dimensional arrays. It is also a powerful technique for implementing complex code such as sorting algorithms.

When using nested loops, the same or different type of loop can be nested within the outer loop e.g. a count-controlled `FOR` loop can be nested within another count-controlled `FOR` loop.

<u>Example 20</u>
Display the first 5 multiples of 1, followed by that of 2 and then 3.

[Solution]

```
01 DECLARE num : INTEGER
02 DECLARE multiplier : INTEGER
03 DECLARE result : INTEGER
04
05 FOR num ← 1 TO 3
06     FOR multiplier ← 1 TO 5
07         result ← num * multiplier
08         OUTPUT result
09     NEXT multiplier
10 NEXT num
```

<u>Notes</u>

- The output is as follows:
  ```
  1
  2
  3
  4
  5
  2
  4
  6
  8
  10
  3
  6
  9
  12
  15
  ```

- The statements in the **outer** loop (lines `06` and `09`) are controlled by line `05 FOR num ← 1 TO 3`.

- The statements in the **inner** loop (lines `07` and `08`) are controlled by line `06 FOR multiplier ← 1 TO 5`.

- In each iteration, the statements in the inner loop will repeat 5 times.

A loop of a different type can also be nested within the outer loop e.g. a condition-controlled `WHILE` loop can be nested within a count-controlled `FOR` loop.

<u>Example 21</u>
Write a program to guess five pre-determined numbers between 1 inclusive and 100 inclusive that are arranged in ascending order.

```
DECLARE to_guess : ARRAY [1 : 5] OF INTEGER
DECLARE guess : INTEGER
DECLARE i : INTEGER

to_guess ← [11, 33, 44, 88, 100]

FOR i ← 1 TO 5

    guess ← -1  // initialise guess using a sentinel value

    WHILE guess <> to_guess[i]
        OUTPUT "Enter your guess"
        INPUT guess
    ENDWHILE

NEXT i

OUTPUT "Congratulations on making the correct guesses!"
```