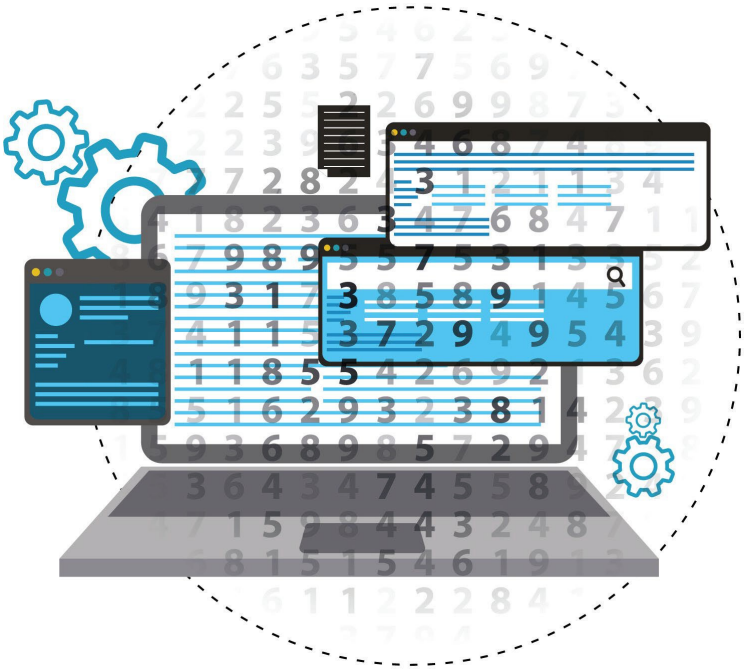
Cambridge Assessment International Education

Pseudocode Guide for Teachers

Cambridge International AS & A Level Computer Science 9618

For examination from 2021



Version 1.4

Cambridge Pathway

In order to help us develop the highest quality resources, we are undertaking a continuous programme of review; not only to measure the success of our resources but also to highlight areas for improvement and to identify new development needs.

We invite you to complete our survey by visiting the website below. Your comments on the quality and relevance of our resources are very important to us.

www.surveymonkey.co.uk/r/GL6ZNJB

Would you like to become a Cambridge International consultant and help us develop support materials?

Please follow the link below to register your interest.

www.cambridgeinternational.org/cambridge-for/teachers/teacherconsultants/

Contents

Introduction

How should teachers use this guide?

We advise teachers to follow this guide in their teaching and make sure that learners are familiar with the style presented here. This will enable learners to understand any pseudocode presented in examination papers more easily. It will also give them a structure to follow so that they can present their algorithms more clearly in pseudocode when required.

Learners should be encouraged to follow this guide in their examination answers or any other material they present for assessment. By definition, pseudocode is not a programming language with a defined, mandatory syntax. Any pseudocode presented by candidates will only be assessed for the logic of the solution presented – where the logic is understood by the Examiner, and correctly solves the problem addressed, the candidate will be given credit regardless of whether the candidate has followed the style presented here. However, candidates are required to write pseudocode for questions that require answers in pseudocode and not a programming language. Using a recommended style will, however, enable the candidate to communicate their solution to the Examiner more effectively.

1. Pseudocode in examined components

The following information sets out how pseudocode will appear within the examined components and is provided to allow you to give learners familiarity before the exam.

1.1 Font style and size

Pseudocode is presented in a monospaced (fixed-width) font such as `Courier New`. The size of the font will be consistent throughout.

1.2 Indentation

Lines are indented (usually by three spaces) to indicate that they are contained within a statement in a previous line. In cases where line numbering is used, this indentation may be omitted. Every effort will be made to make sure that code statements are not longer than a line of text unless this is absolutely necessary. Where necessary, continuation lines will be aligned to maximise readability.

1.3 Case

Keywords are in upper-case, e.g. `IF`, `REPEAT`, `PROCEDURE`. (Different keywords are explained in later sections of this guide.)

Identifiers are in mixed case (sometimes referred to as camelCase or Pascal case) with upper-case letters indicating the beginning of new words, for example `NumberOfPlayers`.

Meta-variables – symbols in the pseudocode that should be substituted by other symbols are enclosed in angled brackets `< >` (as in Backus-Naur Form). This is also used in this guide.

Example – meta-variables

```
REPEAT
   <statement(s)>
UNTIL <condition>
```

1.4 Lines and line numbering

Where it is necessary to number the lines of pseudocode so that they can be referred to, line numbers are presented to the left of the pseudocode with sufficient space to indicate clearly that they are not part of the pseudocode statements.

Line numbers are consecutive, unless numbers are skipped to indicate that part of the code is missing. This will also be clearly stated.

Each line representing a statement is numbered. However, when a statement runs over one line of text, the continuation lines are not numbered.

1.5 Comments

Comments are preceded by two forward slashes //. The comment continues until the end of the line. For multi-line comments, each line is preceded by //.

Normally the comment is on a separate line before, and at the same level of indentation as, the code it refers to. Occasionally, however, a short comment that refers to a single line may be at the end of the line to which it refers.

Example – comments

```
// this procedure swaps
// values of X and Y
PROCEDURE SWAP(BYREF X : INTEGER, Y : INTEGER)
    Temp ← X    // temporarily store X
    X ← Y
    Y ← Temp
ENDPROCEDURE
```

2. Variables, constants and data types

2.1. Data Types

The following keywords are used to designate some basic data types:

- `INTEGER` a whole number
- `REAL` a number capable of containing a fractional part
- `CHAR` a single character
- `STRING` a sequence of zero or more characters
- `BOOLEAN` the logical values `TRUE` and `FALSE`
- `DATE` a valid calendar date

2.2. Literals

Literals of the above data types are written as follows:

- Integer — Written as normal in the denary system, e.g. `5, -3`
- Real — Always written with at least one digit on either side of the decimal point, zeros being added if necessary, e.g. `4.7, 0.3, -4.0, 0.0`
- Char — A single character delimited by single quotes e.g. `'x', 'C', '@'`
- String — Delimited by double quotes. A string may contain no characters (i.e. the empty string) e.g. `"This is a string", ""`
- Boolean — `TRUE`, `FALSE`
- Date — This will normally be written in the format `dd/mm/yyyy`. However, it is good practice to state explicitly that this value is of data type `DATE` and to explain the format (as the convention for representing dates varies across the world).

2.3. Identifiers

Identifiers (the names given to variables, constants, procedures and functions) are in mixed case. They can only contain letters (`A–Z`, `a–z`), digits (`0–9`) and the underscore character (_). They must start with a letter and not a digit. Accented letters should not be used.

It is good practice to use identifier names that describe the variable, procedure or function they refer to. Single letters may be used where these are conventional (such as `i` and `j` when dealing with array indices, or `X` and `Y` when dealing with coordinates) as these are made clear by the convention.

Keywords identified elsewhere in this guide should never be used as variables.

Identifiers should be considered case insensitive, for example, `Countdown` and `CountDown` should not be used as separate variables.

2.4. Variable declarations

It is good practice to declare variables explicitly in pseudocode.

Declarations are made as follows:

```
DECLARE <identifier> : <data type>
```

Example – variable declarations

```
DECLARE Counter : INTEGER
DECLARE TotalToPay : REAL
DECLARE GameOver : BOOLEAN
```

2.5. Constants

It is good practice to use constants if this makes the pseudocode more readable, as an identifier is more meaningful in many cases than a literal. It also makes the pseudocode easier to update if the value of the constant changes.

Constants are normally declared at the beginning of a piece of pseudocode (unless it is desirable to restrict the scope of the constant).

Constants are declared by stating the identifier and the literal value in the following format:

```
CONSTANT <identifier> = <value>
```

Example – `CONSTANT` declarations

```
CONSTANT HourlyRate = 6.50
CONSTANT DefaultText = "N/A"
```

Only literals can be used as the value of a constant. A variable, another constant or an expression must never be used.

2.6. Assignments

The assignment operator is ← .

Assignments should be made in the following format:

```
<identifier> ← <value>
```

The identifier must refer to a variable (this can be an individual element in a data structure such as an array or a user defined data type). The value may be any expression that evaluates to a value of the same data type as the variable.

Example – assignments

```
Counter ← 0
Counter ← Counter + 1
TotalToPay ← NumberOfHours * HourlyRate
```

3. Arrays

Syllabus requirements

The Cambridge International AS & A Level syllabus (9618) requires candidates to understand and use both one-dimensional and two-dimensional arrays.

3.1. Declaring arrays

Arrays are considered to be fixed-length structures of elements of identical data type, accessible by consecutive index (subscript) numbers. It is good practice to explicitly state what the lower bound of the array (i.e. the index of the first element) is because this defaults to either 0 or 1 in different systems. Generally, a lower bound of 1 will be used.

Square brackets are used to indicate the array indices.

A One-dimensional array is declared as follows:

```
DECLARE <identifier>:ARRAY[<lower>:<upper>] OF <data type>
```

A two-dimensional array is declared as follows:

```
DECLARE <identifier>:ARRAY[<lower1>:<upper1>,<lower2>:<upper2>] OF <data type>
```

Example – array declarations

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

3.2. Using arrays

Array index values may be literal values or expressions that evaluate to a valid integer value.

Example – Accessing individual array elements

```
StudentNames[1] ← "Ali"
NoughtsAndCrosses[2,3] ← 'X'
StudentNames[n+1] ← StudentNames[n]
```

Arrays can be used in assignment statements (provided they have same size and data type). The following is therefore allowed:

Example – Accessing a complete array

```
SavedGame ← NoughtsAndCrosses
```

A statement should **not** refer to a group of array elements individually. For example, the following construction should not be used.

```
StudentNames [1 TO 30] ← ""
```

Instead, an appropriate loop structure is used to assign the elements individually. For example:

Example – assigning a group of array elements

```
FOR Index ← 1 TO 30
   StudentNames[Index] ← ""
NEXT Index
```

4. User-defined data types

Syllabus requirements

The AS & A Level (9618) syllabus requires candidates to understand that data structures that are not available in a particular programming language need to be constructed from the data structures that are built-in within the language. User-defined data types need to be defined. The syllabus requires candidates to use and define non-composite data types such as enumerated and pointer and composite data types such as record, set, class/object. Abstract Data Types (ADTs) stack, queue, linked list, dictionary and binary tree are also defined as composite data types.

4.1. Defining user-defined data types

Non-composite data type – Enumerated

A user-defined non-composite data type with a list of possible values is called an enumerated data type. The enumerated type should be declared as follows:

```
TYPE <identifier> = (value1, value2, value3, ...)
```

Example – declaration of enumerated type

This enumerated type holds data about seasons of the year.

```
TYPE Season = (Spring, Summer, Autumn, Winter)
```

Non-composite data type – Pointer

A user-defined non-composite data type referencing a memory location is called a pointer. The pointer should be declared as follows:

```
TYPE <identifier> = ^<data type>
```

The ^ shows that the variable is a pointer and the data type indicates the type of the data stored in the memory location.

Example – declarations of pointer type

```
TYPE TIntPointer = ^INTEGER
TYPE TCharPointer = ^CHAR
```

Declaration of a variable of pointer type does not require the ^ (caret) symbol to be used.

Example – declaration of a pointer variable

```
DECLARE MyPointer : TIntPointer
```

Composite data type

A composite data type is a collection of data that can consist of different data types, grouped under one identifier. The composite type should be declared as follows:

```
TYPE <identifier1>
   DECLARE <identifier2> : <data type>
   TECLARE <identifier3> : <data type>
   ...
ENDTYPE
```

Example – declaration of composite type

This user-defined data type holds data about a student.

```
TYPE Student
    DECLARE LastName : STRING
    DECLARE FirstName : STRING
    DECLARE DateOfBirth : DATE
    DECLARE YearGroup : INTEGER
    DECLARE FormGroup : CHAR
ENDTYPE
```

4.2. Using user-defined data types

When a user-defined data type has been defined it can be used in the same way as any other data type in declarations.

Variables of a user-defined data type can be assigned to each other. Individual data items are accessed using dot notation.

Example – using user-defined data types

This pseudocode uses the user-defined types `Student`, `Season` and `TIntPointer` defined in the previous section.

```
DECLARE Pupil1 : Student
DECLARE Pupil2 : Student
DECLARE Form : ARRAY[1:30] OF Student
DECLARE ThisSeason : Season
DECLARE NextSeason : Season
DECLARE MyPointer : TIntPointer

Pupil1.LastName ← "Johnson"
Pupil1.Firstname ← "Leroy"
Pupil1.DateOfBirth ← 02/01/2005
Pupil1.YearGroup ← 6
Pupil1.FormGroup ← 'A'

Pupil2 ← Pupil1

FOR Index ← 1 TO 30
    Form[Index].YearGroup ← Form[Index].YearGroup + 1
NEXT Index

ThisSeason ← Spring
MyPointer ← ^ThisSeason
NextSeason ← MyPointer^ + 1
// access the value stored at the memory address
```

5. Common operations

5.1. Input and output

Values are input using the INPUT command as follows:

```
INPUT <identifier>
```

The identifier should be a variable (that may be an individual element of a data structure such as an array, or a custom data type).

Values are output using the OUTPUT command as follows:

```
OUTPUT <value(s)>
```

Several values, separated by commas, can be output using the same command.

Example – `INPUT` and `OUTPUT` statements

```
INPUT Answer
OUTPUT Score
OUTPUT "You have ", Lives, " lives left"
```

5.2. Arithmetic operations

Standard arithmetic operator symbols are used:

+ Addition

- Subtraction

* Multiplication

/ Division (The resulting value should be of data type REAL, even if the operands are integers.)

DIV Integer division**:** Used to find the quotient (integer number before the decimal point) after division.

MOD or Modulus: The remainder that is left over when one number is divided by another.

Multiplication and division have higher precedence over addition and subtraction (this is the normal mathematical convention). However, it is good practice to make the order of operations in complex expressions explicit by using parentheses.

5.3. Relational operations

The following symbols are used for relational operators (also known as comparison operators):

>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
=	Equal to
<>	Not equal to

The result of these operations is always of data type `BOOLEAN`.

In complex expressions it is advisable to use parentheses to make the order of operations explicit.

5.4. Logic operators

The only logic operators (also called relational operators) used are `AND`, `OR` and `NOT`. The operands and results of these operations are always of data type `BOOLEAN`.

In complex expressions it is advisable to use parentheses to make the order of operations explicit.

5.5. String functions and operations

Syllabus requirements

The AS & A Level (9618) syllabus specifically requires candidates to know string manipulation functions in their chosen programming language. Pseudocode string manipulation functions will always be provided in examinations. Some basic string manipulation functions are given here.

Each function returns an error if the function call is not properly formed.

`RIGHT(ThisString : STRING, x : INTEGER) RETURNS STRING`
returns rightmost `x` characters from `ThisString`

Example: **`RIGHT("ABCDEFGH", 3)`** returns **`"FGH"`**

`LENGTH(ThisString : STRING) RETURNS INTEGER`
returns the integer value representing the length of `ThisString`

Example: **`LENGTH("Happy Days")`** returns **`10`**

`MID(ThisString : STRING, x : INTEGER, y : INTEGER) RETURNS STRING`
returns a string of length `y` starting at position `x` from `ThisString`

Example: **`MID("ABCDEFGH", 2, 3)`** returns **`"BCD"`**

`LCASE(ThisChar : CHAR) RETURNS CHAR`
returns the character value representing the lower-case equivalent of `ThisChar`
If `ThisChar` is not an upper-case alphabetic character, it is returned unchanged.

Example: **`LCASE('W')`** returns **`'w'`**

`UCASE(ThisChar : CHAR) RETURNS CHAR`
returns the character value representing the upper-case equivalent of `ThisChar`
If `ThisChar` is not a lower-case alphabetic character, it is returned unchanged.

Example: **`UCASE('h')`** returns **`'H'`**

In pseudocode, the operator `&` is used to concatenate (join) two strings.

Example: **"Summer" & " " & "Pudding"** produces **"Summer Pudding"**

Where string operations (such as concatenation, searching and splitting) are used in a programming language, these should be explained clearly, as they vary considerably between systems.

Where functions in programming languages are used to format numbers as strings for output, their use should also be explained.

5.6. Numeric functions

```
INT(x : REAL) RETURNS INTEGER
```
returns the integer part of `x`

Example: **`INT(27.5415)`** returns **`27`**

```
RAND(x : INTEGER) RETURNS REAL
```
returns a random real number in the range `0` to `x` (not inclusive of `x`)

Example: **`RAND(87)`** may return **`35.43`**

6. Selection

6.1. IF statements

`IF` statements may or may not have an `ELSE` clause.

`IF` statements without an `ELSE` clause are written as follows:

```
IF <condition> THEN
   <statement(s)>
ENDIF
```

`IF` statements with an `ELSE` clause are written as follows:

```
IF <condition> THEN
    <statement(s)>
ELSE
    <statement(s)>
ENDIF
```

Note, due to space constraints, the `THEN` and `ELSE` clauses may only be indented by two spaces rather than three. (They are, in a sense, a continuation of the `IF` statement rather than separate statements).

Example – nested `IF` statements

```
IF ChallengerScore > ChampionScore THEN
    IF ChallengerScore > HighestScore THEN
       OUTPUT ChallengerName, " is champion and highest scorer"
    ELSE
       OUTPUT ChallengerName, " is the new champion"
    ENDIF
ELSE
    OUTPUT ChampionName, " is still the champion"
    IF ChampionScore > HighestScore THEN
       OUTPUT ChampionName, " is also the highest scorer"
    ENDIF
ENDIF
```

6.2. CASE statements

CASE statements allow one out of several branches of code to be executed, depending on the value of a variable.

CASE statements are written as follows:

```
CASE OF <identifier>
   <value 1> : <statement1>
               <statement2>
               ...
   <value 2> : <statement1>
               <statement2>
               ...
   ...
ENDCASE
```

An OTHERWISE clause can be the last case:

```
CASE OF <identifier>
   <value 1> : <statement1>
               <statement2>
               ...
   <value 2> : <statement1>
               <statement2>
               ...
   OTHERWISE : <statement1>
               <statement2>
               ...

ENDCASE
```

Each value may be represented by a range, for example:

```
 <value1> TO <value2> : <statement1>
                        <statement2>
                        ...
```

Note that the CASE clauses are tested in sequence. When a case that applies is found, its statement is executed and the CASE statement is complete. Control is passed to the statement after the ENDCASE. Any remaining cases are not tested.

If present, an OTHERWISE clause must be the last case. Its statement will be executed if none of the preceding cases apply.

Example – formatted CASE statement

```
INPUT Move
CASE OF Move
   'W' : Position ← Position - 10
   'S' : Position ← Position + 10
   'A' : Position ← Position - 1
   'D' : Position ← Position + 1
   OTHERWISE : CALL Beep
ENDCASE
```

7. Iteration (repetition)

7.1. Count-controlled (FOR) loops

Count-controlled loops are written as follows:

```
FOR <identifier> ← <value1> TO <value2>
    <statement(s)>
NEXT <identifier>
```

The identifier must be a variable of data type `INTEGER`, and the values should be expressions that evaluate to integers.

The variable is assigned each of the integer values from `value1` to `value2` inclusive, running the statements inside the `FOR` loop after each assignment. If `value1` = `value2` the statements will be executed once, and if `value1` > `value2` the statements will not be executed.

It is good practice to repeat the identifier after `NEXT`, particularly with nested `FOR` loops.

An increment can be specified as follows:

```
FOR <identifier> ← <value1> TO <value2> STEP <increment>
    <statement(s)>
NEXT <identifier>
```

The increment must be an expression that evaluates to an integer. In this case the `identifier` will be assigned the values from `value1` in successive increments of `increment` until it reaches `value2`. If it goes past `value2`, the loop terminates. The `increment` can be negative.

Example – nested FOR loops

```
Total ← 0
FOR Row ← 1 TO MaxRow
   RowTotal ← 0
   FOR Column ← 1 TO 10
      RowTotal ← RowTotal + Amount[Row, Column]
   NEXT Column
   OUTPUT "Total for Row ", Row, " is ", RowTotal
   Total ← Total + RowTotal
NEXT Row
OUTPUT "The grand total is ", Total
```

7.2. Post-condition (REPEAT) loops

Post-condition loops are written as follows:

```
REPEAT
    <statement(s)>
UNTIL <condition>
```

The condition must be an expression that evaluates to a Boolean.

The statements in the loop will be executed at least once. The condition is tested after the statements are executed and if it evaluates to `TRUE` the loop terminates, otherwise the statements are executed again.

Example – `REPEAT UNTIL` loop

```
REPEAT
   OUTPUT "Please enter the password"
   INPUT Password
UNTIL Password = "Secret"
```

7.3. Pre-condition (WHILE) loops

Pre-condition loops are written as follows:

```
WHILE <condition>
    <statement(s)>
ENDWHILE
```

The condition must be an expression that evaluates to a Boolean.

The condition is tested before the statements, and the statements will only be executed if the condition evaluates to `TRUE`. After the statements have been executed the condition is tested again. The loop terminates when the condition evaluates to `FALSE`.

The statements will not be executed if, on the first test, the condition evaluates to `FALSE`.

Example – `WHILE` loop

```
WHILE Number > 9
   Number ← Number - 9
ENDWHILE
```

8. Procedures and functions

Syllabus requirements

The definition and use of procedures and functions is explicitly required in the AS & A Level (9618) syllabus. Any pseudocode functions used in an examination will be defined.

8.1. Defining and calling procedures

A procedure with no parameters is defined as follows:

```
PROCEDURE <identifier>
    <statement(s)>
ENDPROCEDURE
```

A procedure with parameters is defined as follows:

```
PROCEDURE <identifier>(<param1> : <data type>, <param2> : <data type>...)
    <statement(s)>
ENDPROCEDURE
```

The <identifier> is the identifier used to call the procedure. Where used, param1, param2 etc. are identifiers for the parameters of the procedure. These will be used as variables in the statements of the procedure.

Procedures defined as above should be called as follows, respectively:

```
CALL <identifier>

CALL <identifier>(Value1, Value2, ...)
```

These calls are complete program statements.

When parameters are used, Value1, Value2... must be of the correct data type and in the same sequence as in the definition of the procedure.
Unless otherwise stated, it should be assumed that parameters are passed by value. (See section 8.3).

Example – definition and use of procedures with and without parameters

```
PROCEDURE DefaultSquare
   CALL Square(100)
ENDPROCEDURE

PROCEDURE Square(Size : INTEGER)
   FOR Side ← 1 TO 4
      CALL MoveForward(Size)
      CALL Turn(90)
   NEXT Side
ENDPROCEDURE

IF Size = Default THEN
   CALL DefaultSquare
ELSE
   CALL Square(Size)
ENDIF
```

8.2. Defining and calling functions

Functions operate in a similar way to procedures, except that in addition they return a single value to the point at which they are called. Their definition includes the data type of the value returned.

A function with no parameters is defined as follows:

```
FUNCTION <identifier> RETURNS <data type>
    <statement(s)>
ENDFUNCTION
```

A function with parameters is defined as follows:

```
FUNCTION <identifier>(<param1> : <data type>,
                      <param2> : <data type>…) RETURNS <data type>
    <statement(s)>
ENDFUNCTION
```

The keyword `RETURN` is used as one of the statements within the body of the function to specify the value to be returned. Normally, this will be the last statement in the function definition, however, if the `RETURN` statement is in the body of the function its execution is immediate and any subsequent lines of code are omitted.

Because a function returns a value that is used when the function is called, function calls are not complete program statements. The keyword `CALL` should not be used when calling a function. Functions should only be called as part of an expression. When the `RETURN` statement is executed, the value returned replaces the function call in the expression and the expression is then evaluated.

Example – definition and use of a function

```
FUNCTION Max(Number1 : INTEGER, Number2 : INTEGER) RETURNS INTEGER
   IF Number1 > Number2 THEN
      RETURN Number1
   ELSE
      RETURN Number2
   ENDIF
ENDFUNCTION

OUTPUT "Penalty Fine = ", Max(10, Distance*2)
```

8.3. Passing parameters by value or by reference

To specify whether a parameter is passed by value or by reference, the keywords `BYVAL` and `BYREF` precede the parameter in the definition of the procedure. If there are several parameters passed by the same method, the `BYVAL` or `BYREF` keyword need not be repeated.

Example – passing parameters by reference

```
PROCEDURE SWAP(BYREF X : INTEGER, Y : INTEGER)
   Temp ← X
   X ← Y
   Y ← Temp
ENDPROCEDURE
```

If the method for passing parameters is not specified, passing by value is assumed. How this should be called and how it operates has already been explained in Section 8.1.

Parameters should not be passed by reference to a function.

9. File handling

9.1. Handling text files

Text files consist of lines of text that are read or written consecutively as strings.

A file must be opened in a specified mode before any file operations are attempted. This is written as follows:

```
OPENFILE <file identifier> FOR <file mode>
```

The file identifier may be a literal string containing the file names, or a variable of type `STRING` that has been assigned the file name.

The following file modes are used:

- `READ` for data to be read from the file
- `WRITE` for data to be written to the file. A new file will be created and any existing data in the file will be lost.
- `APPEND` for data to be added to the file, after any existing data.

A file should be opened in only one mode at a time.

Data is read from the file (after the file has been opened in `READ` mode) using the `READFILE` command as follows:

```
READFILE <file identifier>, <variable>
```

The `variable` should be of data type `STRING`. When the command is executed, the next line of text in the file is read and assigned to the variable.

The function `EOF` is used to test whether there are any more lines to be read from a given file. It is called as follows:

```
EOF(<file identifier>)
```

This function returns `TRUE` if there are no more lines to read (or if an empty file has been opened in `READ` mode) and `FALSE` otherwise.

Data is written into the file (after the file has been opened in `WRITE` or `APPEND` mode) using the `WRITEFILE` command as follows:

```
WRITEFILE <file identifier> , <data>
```

Files should be closed when they are no longer needed using the `CLOSEFILE` command as follows:

```
CLOSEFILE <file identifier>
```

Example – handling text files

This example uses the operations together, to copy all the lines from `FileA.txt` to `FileB.txt`, replacing any blank lines by a line of dashes.

```
DECLARE LineOfText : STRING
OPENFILE "FileA.txt" FOR READ
OPENFILE "FileB.txt" FOR WRITE
WHILE NOT EOF("FileA.txt")
   READFILE "FileA.txt", LineOfText
   IF LineOfText = "" THEN
      WRITEFILE "FileB.txt", " ----------------------------  "
   ELSE
      WRITEFILE "FileB.txt", LineOfText
   ENDIF
ENDWHILE
CLOSEFILE "FileA.txt"
CLOSEFILE "FileB.txt"
```

9.2. Handling random files

Random files contain a collection of data, normally as records of fixed length. They can be thought of as having a file pointer which can be moved to any location or address in the file. The record at that location can then be read or written.

Random files are opened using the `RANDOM` file mode as follows:

```
OPENFILE <file identifier> FOR RANDOM
```

As with text files, the file identifier will normally be the name of the file.

The `SEEK` command moves the file pointer to a given location:

```
SEEK <file identifier>, <address>
```

The address should be an expression that evaluates to an integer which indicates the location of a record to be read or written. This is usually the number of records from the beginning of the file. It is good practice to explain how the addresses are computed.

The command `GETRECORD` should be used to read the record at the file pointer:

```
GETRECORD <file identifier>, <variable>
```

When this command is executed, the record that is read is assigned to the variable which must be of the appropriate data type for that record (usually a user-defined type).

The command `PUTRECORD` is used to write a record into the file at the file pointer:

```
PUTRECORD <file identifier>, <variable>
```

When this command is executed, the data in the variable is inserted into the record at the file pointer. Any data that was previously at this location will be replaced.

Example – handling random files

The records from positions 10 to 20 of a file `StudentFile.Dat` are moved to the next position and a new record is inserted into position 10. The example uses the user-defined type `Student` defined in Section 4.1.

```
DECLARE Pupil : Student
DECLARE NewPupil : Student
DECLARE Position : INTEGER

NewPupil.LastName ← "Johnson"
NewPupil.Firstname ← "Leroy"
NewPupil.DateOfBirth ← 02/01/2005
NewPupil.YearGroup ← 6
NewPupil.FormGroup ← 'A'

OPENFILE "StudentFile.Dat" FOR RANDOM
FOR Position ← 20 TO 10 STEP -1
    SEEK "StudentFile.Dat", Position
    GETRECORD "StudentFile.Dat", Pupil
    SEEK "StudentFile.Dat", Position + 1
    PUTRECORD "StudentFile.Dat", Pupil
NEXT Position

SEEK "StudentFile.Dat", 10
PUTRECORD "StudentFile.Dat", NewPupil

CLOSEFILE "StudentFile.dat"
```

10. Object-oriented Programming

10.1. Methods and Properties

Methods and properties can be assumed to be public unless otherwise stated. Where the access level is relevant to the question, it will be explicit in the code using the keywords `PUBLIC` or `PRIVATE`.

Example code:

```
PRIVATE Attempts : INTEGER
Attempts ← 3

PUBLIC PROCEDURE SetAttempts(Number : INTEGER)
   Attempts ← Number
ENDPROCEDURE

PRIVATE FUNCTION GetAttempts() RETURNS INTEGER
   RETURN Attempts
ENDFUNCTION
```

Methods will be called using object methods, for example:

```
Player.SetAttempts(5)
OUTPUT Player.GetAttempts()
```

10.2. Constructors and Inheritance

Constructors will be procedures with the name `NEW`.

```
CLASS Pet
   PRIVATE Name : STRING
   PUBLIC PROCEDURE NEW(GivenName : STRING)
      Name ← GivenName
   ENDPROCEDURE
ENDCLASS
```

Inheritance is denoted by the `INHERITS` keyword; superclass/parent class methods will be called using the keyword `SUPER`, for example:

```
CLASS Cat INHERITS Pet
   PRIVATE Breed: INTEGER
   PUBLIC PROCEDURE NEW(GivenName : STRING, GivenBreed : STRING)
      SUPER.NEW(GivenName)
      Breed ← GivenBreed
   ENDPROCEDURE
ENDCLASS
```

To create an object, the following format is used:

```
<object name> ← NEW <class name>(<param1>, <param2> ...)
```

For example:

```
MyCat ← NEW Cat("Kitty", "Persian")
```

Index of symbols and keywords

Cambridge Assessment International Education
The Triangle Building, Shaftesbury Road, Cambridge, CB2 8EA, United Kingdom
t: +44 1223 553554
e: info@cambridgeinternational.org www.cambridgeinternational.org