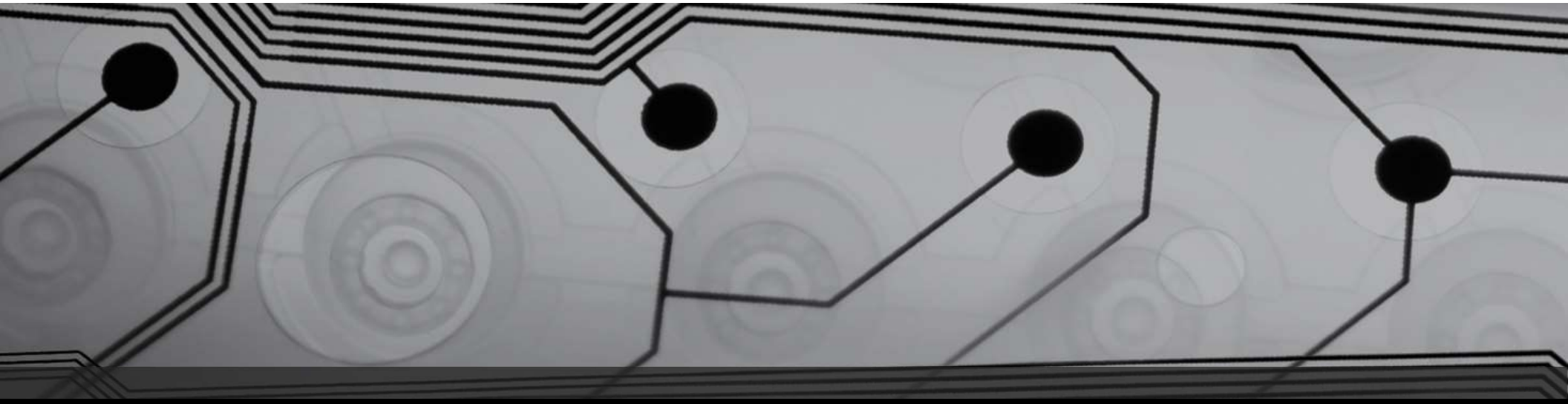
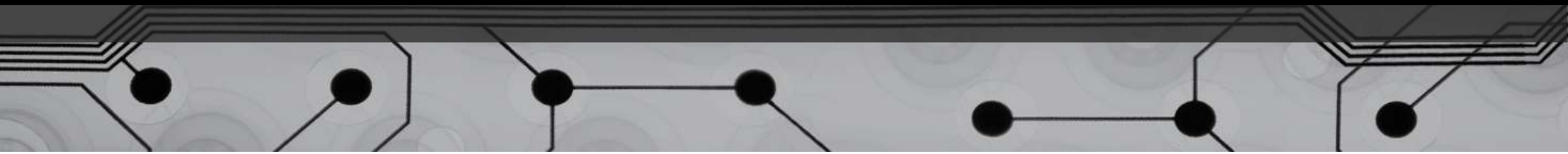


Cambridge IGCSE 0478 Computer Science

FRITZ EUGENE BANSAG



KNOWING WHAT YOU KNOW

- Go to https://joinmyquiz.com
  - Write your name and grade level
  - Join Code: _____

Cambridge 0478 Exam Components

Components at a glance

This table summarises the key information about each examination paper. You can find details and advice on how to approach each component in the 'About each paper' sub-section.

Component	Time and marks	Content/Skills assessed	Details	Percentage of qualification
Paper 1 Computer Systems	1 hour 45 minutes 75 marks	Questions will be based on Topics 1–6 of the subject content	Short-answer and structured questions All questions are compulsory No calculators are permitted Externally assessed	50%
Paper 2 Algorithms, Programming and Logic	1 hour 45 minutes 75 marks	Questions will be based on Topics 7–10 of the subject content	Short-answer and structured questions and a scenario-based question All questions are compulsory No calculators are permitted Externally assessed	50%

PSEUDOCODE IN EXAMINED COMPONENTS

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 by four spaces to indicate that they are contained within a statement in a previous line.

Where it is not possible to fit a statement on one line any continuation lines are indented by two spaces. 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 code, unless this is absolutely necessary.

Note that the `THEN` and `ELSE` clauses of an `IF` statement are indented by only two spaces (see Section 5.1). Cases in `CASE` statements are also indented by only two places (see Section 5.2).

PSEUDOCODE IN EXAMINED COMPONENTS

1.3 Case and italics

Keywords are in uppercase, 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 uppercase 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
     <Statements>
UNTIL <condition>
```

PSEUDOCODE IN EXAMINED COMPONENTS

1.4 Lines and 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.

1.4 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.

PSEUDOCODE IN EXAMINED COMPONENTS

1.4 Lines and 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.

1.4 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.

2. VARIABLES, CONSTANTS, & DATATYPES

2.1 Atomic type names

The following keywords are used to designate atomic 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`

2.2 Literals

Literals of the above data types are written as follows:

- Integers : 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`

2. VARIABLES, CONSTANTS, & DATATYPES

2.3 Identifiers

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

As in programming, 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 Assignments

The assignment operator is ←.

Assignments should be made in the following format:

```
<identifier> ← <value>
```

2. VARIABLES, CONSTANTS, & DATATYPES

2.4 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 an abstract 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

3.1 Using arrays

In the main pseudocode statements, only one index value is used for each dimension in the square brackets.

Example – using arrays

```
StudentNames[1]  ← "Ali"
NoughtsAndCrosses[2] ← '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 – assigning an array

```
SavedGame ← NoughtsAndCrosses
```

3. ARRAYS

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

Example – assigning an array

```
SavedGame ← NoughtsAndCrosses
```

A statement should **not**, however, 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
```

3. ARRAYS

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

Example – assigning an array

```
SavedGame ← NoughtsAndCrosses
```

A statement should **not**, however, 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. COMMON OPERATIONS

4.1 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"
```

Note that the syllabus for IGCSE (0478) gives `READ` and `PRINT` as examples for `INPUT` and `OUTPUT`, respectively.

4. COMMON OPERATIONS

4.2 Arithmetic operations

Standard arithmetic operator symbols are used:

- + Addition
- – Subtraction
- * Multiplication
- / Division

Care should be taken with the division operation: the resulting value should be of data type `REAL`, even if the operands are integers.

The integer division operators `MOD` and `DIV` can be used. However, their use should be explained explicitly and not assumed.

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.

4. COMMON OPERATIONS

4.2 Arithmetic operations

Standard arithmetic operator symbols are used:

- + Addition
- – Subtraction
- * Multiplication
- / Division

Care should be taken with the division operation: the resulting value should be of data type `REAL`, even if the operands are integers.

The integer division operators `MOD` and `DIV` can be used. However, their use should be explained explicitly and not assumed.

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.

4.3 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. SELECTION

5.1 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
    <statements>
ENDIF
```

IF statements with an else clause are written as follows:

```
IF <condition>
  THEN
    <statements>
  ELSE
    <statements>
ENDIF
```

Note that the THEN and ELSE clauses are only indented by two spaces. (They are, in a sense, a continuation of the IF statement rather than separate statements).

When IF statements are nested, the nesting should continue the indentation of two spaces. In particular, run-on THEN IF and ELSE IF lines should be avoided.

5. SELECTION

5.1 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
    <statements>
ENDIF
```

IF statements with an else clause are written as follows:

```
IF <condition>
  THEN
    <statements>
  ELSE
    <statements>
ENDIF
```

Note that the THEN and ELSE clauses are only indented by two spaces. (They are, in a sense, a continuation of the IF statement rather than separate statements).

When IF statements are nested, the nesting should continue the indentation of two spaces. In particular, run-on THEN IF and ELSE IF lines should be avoided.

5. SELECTION

Note that the `THEN` and `ELSE` clauses are only indented by two spaces. (They are, in a sense, a continuation of the IF statement rather than separate statements).

When `IF` statements are nested, the nesting should continue the indentation of two spaces. In particular, run-on `THEN IF` and `ELSE IF` lines should be avoided.

Example – nested `IF` statements

```
IF ChallengerScore > ChampionScore
     THEN
        IF ChallengerScore > HighestScore
        THEN
             OUTPUT ChallengerName, " is champion and highest scorer"
          ELSE
             OUTPUT PlayerlName, " 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
```

5. SELECTION

It is best practice to keep the branches to single statements as this makes the pseudocode more readable. Similarly single values should be used for each case. If the cases are more complex, the use of an `IF` statement, rather than a `CASE` statement, should be considered.

Each case clause is indented by two spaces. They can be seen as continuations of the `CASE` statement rather than new statements.

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 : Beep
ENDCASE
```

6. ITERATION

6.1 Count-controlled (FOR) loops

Count-controlled loops are written as follows:

```
FOR <identifier> ← <value1> TO <value2>
    <statements>
NEXT
```

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.

6. ITERATION

6.1 Count-controlled (FOR) loops

Count-controlled loops are written as follows:

```
FOR <identifier> ← <value1> TO <value2>
    <statements>
NEXT
```

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>
    <statements>
NEXT
```

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.

6. ITERATION

An increment can be specified as follows:

```
FOR <identifier> ← <value1> TO <value2> STEP <increment>
    <statements>
NEXT
```

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
```

6. ITERATION

An increment can be specified as follows:

```
FOR <identifier> ← <value1> TO <value2> STEP <increment>
    <statements>
NEXT
```

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
```

6. ITERATION

6.2 Post-condition (REPEAT UNTIL) loops

Post-condition loops are written as follows:

```
REPEAT
    <Statements>
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.

6. ITERATION

6.2 Post-condition (REPEAT UNTIL) loops

Post-condition loops are written as follows:

```
REPEAT
    <Statements>
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` statement

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

6. ITERATION

6.3 Pre-condition (WHILE) loops

Pre-condition loops are written as follows:

```
WHILE <condition> DO
    <statements>
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`.

6. ITERATION

6.3 Pre-condition (WHILE) loops

Pre-condition loops are written as follows:

```
WHILE <condition> DO
    <statements>
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 DO
  Number ← Number - 9
ENDWHILE
```

KNOWING WHAT YOU LEARNED

- Go to https://joinmyquiz.com
  - Write your name and grade level
  - Join Code: _____