AQA Computer Science A-Level

4.1.1 Programming

Advanced Notes

Specification:

4.1.1.1 Data types:

Understand the concept of a data type.

Understand and use the following appropriately:

- integer
- real/float
- Boolean
- character
- string
- date/time
- pointer/reference
- records (or equivalent)
- arrays (or equivalent)

Define and use user-defined data types based on language-defined (built-in) data types.

4.1.1.2 Programming concepts:

Use, understand and know how the following statement types can be combined in programs:

- variable declaration
- constant declaration
- assignment
- iteration
- selection
- subroutine (procedure / function)

Use definite and indefinite iteration, including indefinite iteration with the condition(s) at the start or the end of the iterative structure. A theoretical understanding of condition(s) at either end of an iterative structure is required, regardless of whether they are supported by the language being used.

Use nested selection and nested iteration structures.

Use meaningful identifier names and know why it is important to use them

4.1.1.3 Arithmetic operations

Be familiar with and be able to use:

- addition
- subtraction
- multiplication
- real/float division
- integer division, including remainders
- exponentiation
- rounding
- truncation

4.1.1.4 Relational operations in a programming language

Be familiar with and be able to use:

- equal to
- not equal to
- less than
- greater than
- less than or equal to
- greater than or equal to

4.1.1.5 Boolean operations in a programming language

Be familiar with and be able to use:

- NOT
- AND
- OR
- XOR

4.1.1.6 Constants and variables in a programming language

Be able to explain the differences between a variable and a constant.

Be able to explain the advantages of using named constants.

4.1.1.7 String-handling operations in a programming language

Be familiar with and be able to use:

- length
- position
- substring
- concatenation
- character → character code
- character code → character
- string conversion operations

4.1.1.8 Random number generation in a programming language

Be familiar with, and be able to use, random number generation.

4.1.1.9 Exception handling

Be familiar with the concept of exception handling.

Know how to use exception handling in a programming language with which students are familiar.

4.1.1.10 Subroutines (procedures/functions)

Be familiar with subroutines and their uses.

Know that a subroutine is a named 'out of line' block of code that may be executed (called) by simply writing its name in a program statement.

Be able to explain the advantages of using subroutines in programs.

4.1.1.11 Parameters of subroutines

Be able to describe the use of parameters to pass data within programs.

Be able to use subroutines with interfaces.

4.1.1.12 Returning a value/values from a subroutine

Be able to use subroutines that return values to the calling routine.

4.1.1.13 Local variables in subroutines

Know that subroutines may declare their own variables, called local variables, and that local variables:

- exist only while the subroutine is executing
- are accessible only within the subroutine

Be able to use local variables and explain why it is good practice to do so.

4.1.1.14 Global variables in a programming language

Be able to contrast local variables with global variables.

4.1.1.15 Role of stack frames in subroutine calls

Be able to explain how a stack frame is used with subroutine calls to store:

- return addresses
- parameters
- local variables

4.1.1.16 Recursive techniques

Be familiar with the use of recursive techniques in programming languages (general and base cases and the mechanism for implementation).

Be able to solve simple problems using recursion.

Data Types

The way in which data is stored depends on what the data is. A data type is defined by the values it can take or the operations which can be performed on it.

In some situations, it might be possible to store one piece of data using various different data types. In this case, the programmer must decide which option is the best suited to solving a particular problem or which is the most memory-efficient.

For example, if a programmer needs to store a user's age in years, they could use a string or an integer. In this situation, using an integer would be the best option, because a person's age is only ever going to contain numerical digits.

Data type	Description
Integer	A whole number, positive or negative, including zero.
Real / Float	A positive or negative number which can have a fractional part.
Boolean	A value which is either true or false.
Character	A single number, letter or symbol.
String	A collection of characters.
Data / Time	A way of storing a point in time, many different formats are used.
Pointer / Reference	A way of storing memory addresses.
Records	A collection of fields, each of which could have a different data type. You can think of a record as a row from a table.
Arrays	A finite, indexed set of related elements each of which has the same data type.

Note

Knowledge of the **pointer / reference** data type is not required for AS level.

User-defined data types

User-defined data types are derived from existing data types in order to create a customised data structure. Creating and using user-defined data types allows a programmer to ensure that a solution is as memory efficient as possible.

Synoptic Link

In many ways, **user-defined data types** are similar to **classes** in **object-oriented programming**.

Classes are covered in **Programming Paradigms**

For example, a shop might use a user-defined data type called `Customer` to store information about their customers. The user-defined data type might have attributes like `Forename`, `Surname` and `EmailAddress`.

The way in which you use user-defined data types differs between programming languages. It's important that you know how to use them in your chosen language.

Programming Concepts

Programming languages support a variety of different statement types, some of which are explained in the table below.

Statement type	Description
Variable declaration	Creating a variable for the first time, giving it a name and sometimes a data type. This allocates a portion of the computer's memory to the variable.
Constant declaration	The same as variable declaration, but when creating a constant. The value of a constant does not change while the program is running.
Assignment	Giving a constant or variable a value.
Iteration	Repeating an instruction, this could be definite or indefinite (see below).
Selection	Comparing values and choosing an action based on those values.
Subroutine	A named block of code containing a set of instructions designed to perform a frequently used operation.

Definite and indefinite iteration

Iteration is the process of repeating a block of code. Examples of iteration include `for` loops and `while` loops.

Definite iteration is a type of iteration in which the number of repetitions required is known before the loop starts.

In contrast to definite iteration, indefinite iteration is used when the number of repetitions required is not known before the loop starts.

```
FOR Count ← 0 TO 63
    OUTPUT Count
ENDFOR
```

This is an example of definite iteration. The `for` loop will run 64 times before finishing.

```
WHILE Temperature = 18
    Temperature = GetTemp()
ENDWHILE
```

The `while` loop above uses indefinite iteration. The number of repetitions is not known before the loop begins.

Nested Structures

Selection structures and iteration structures can be nested.

This means that one structure is placed within another and can easily be identified by different levels of indentation in code.

For example, the pseudocode below consists of an `if` structure, containing further selection and iteration structures.

Whenever a new selection or iteration structure begins, the code moves to a higher level of indentation, making the code easier for humans to understand.

```
IF Colour = "RED" THEN
    WHILE Colour = "RED"
        Colour ← UpdateColour()
    ENDWHILE
ELSE
    IF Colour = "GREEN" THEN
        WHILE Colour = "GREEN"
            Colour ← UpdateColour()
        ENDWHILE
    ELSE
        Colour ← "RED"
    ENDIF
ENDIF
```

Synoptic Link

Indentation is a feature of **High level languages**.

Different types of programming languages are covered in the chapter **Fundamentals of computer systems**.

Meaningful Identifier Names

When declaring a constant, variable or subroutine, it's important to give it a sensible and meaningful identifier name. This makes it easier for others to understand what the purpose of the named object is within the program.

If a different programmer, who was unfamiliar with your program, were to read the code, they should be able to work out the purpose of a constant, variable or subroutine from its name.

Note

Certain programming languages forbid the use of some identifier names which are used as a part of the language itself.

Arithmetic Operations

The following operations can be applied to operands by your programming language. Different languages notate these operations differently, so ensure that you're familiar with your chosen language's approach.

Operation	Description	Example
Addition	When two values are added, the result is the sum of the two values.	128 + 42 = 170
Subtraction	When one value is subtracted from another, the result is the difference between the two numbers.	34 - 13 = 21
Multiplication	The product of two numbers is returned when multiplied.	64 * 2 = 128
Real / Float Division	When one value is divided by another, both a quotient and a remainder are returned.	12 / 8 = 1.5
Integer Division	Integer division returns just the whole number part of a division.	12 \ 8 = 1 *Or* 12 DIV 8 = 1
Modulo	Returns the remainder of an integer division.	12 MOD 8 = 4
Exponentiation	Raising one value to the power of another.	2 ^ 6 = 64
Rounding	Limiting the degree of accuracy of a number, for example, to a set number of significant figures.	3.14159 = 3.14 to 3 significant figures
Truncation	Removing the decimal part of a number. Truncation always returns the whole part of the number and never rounds up.	3.14159 truncated = 3

Relational Operations

You can make use of relational operators whenever you need to compare two values. They are used in iterative and selection structures as well as for base cases in recursion.

Operation	Example
Equal to	12 = 12
Not equal to	16 <> 413 16 != 413
Less than	75 < 422
Greater than	19 > 18
Less than or equal to	6 >= 22 95 >= 95
Greater than or equal to	20 >= 126 44 >= 44

Synoptic Link

Base cases and **recursion** is covered in greater detail later in this document.

Boolean Operations

As explained earlier in this document, a Boolean data type is one whose value can only ever be true or false. There are a series of operations that can be performed on Boolean values.

Operation	Description	Example
NOT	The opposite of a Boolean value	`NOT 1 = 0`
AND	The product of two Boolean values	`1 AND 1 = 1` `0 AND 1 = 0`
OR	The sum of two Boolean values	`1 OR 0 = 1` `1 OR 1 = 1`
XOR	True if strictly one of two values is true	`1 XOR 1 = 0` `1 XOR 0 = 1`

Synoptic Link

Boolean operations form an important part of **logic gates**.

Logic gates are covered in **fundamentals of computer systems**.

Constants and Variables

When a program needs to store data, it usually does so using one of two types of data item: constants or variables.

As their name suggests, variables can change their value during the execution of a program, whereas a constant's value cannot change once assigned.

Constants can be used for storing data that doesn't need to change such as a value for *pi* or the number of days in a year. Using constants allows values to be given identifier names which makes code easier for a human to understand.

Synoptic Link

Named constants should be given **meaningful identifier names** to ensure that their purpose can be understood.

Furthermore, should a constant value be required multiple times throughout a program, using a constant makes changing that value much easier as it only needs to be updated in one place.

Using hard-coded values	Using constants

`HoursWorked ← USERINPUT`
`PAY ← 14 * HoursWorked`
`OUTPUT PAY`
	`HourlyRate ← 14`
	`HoursWorked ← USERINPUT`
	`PAY ← HourlyRate * HoursWorked`
	`OUTPUT PAY`

The pseudocode examples above show two different approaches to the same problem. One approach uses hard-coded values whereas the other uses constants.

The code which makes use of constants is easier to understand as it clearly specifies that 14 refers to an hourly rate. In the example which uses hard-coded values, it's difficult to understand why `HoursWorked` is being multiplied by 14.

String-handling operations

As discussed earlier in this document, a string is a collection of characters. Thanks to their composition, strings can have various functions applied to them.

Function	Description
Length	Returns the number of characters in a specified string.
Position	Returns the position of a specified character within a string.
Substring	Given a starting position and a length, returns a portion of a string.
Concatenation	Joining two or more strings together to form a new, longer string.
Character to character code	Returning the character code which corresponds to a specified character.
Character code to character	Returning the character represented by a given character code.
String to integer	Converting a string to an integer.
String to float	Converting a string to a float.
Integer to string	Converting an integer to a string.
Float to string	Converting a float to a string.
Date / time to string	Converting a date / time data type to a string.
String to date / time	Converting a string to a date / time data type.

Synoptic Link

Characters are linked to **character codes** by **information coding systems**.

Information coding systems are covered in **fundamentals of data representation**.

Note

AMP **is a substring of** ***EXAMPLE*** **with a starting position of 2 and a length of 3.**

Random number generation

Most high level programming languages have the ability to generate random numbers.

A built-in function takes a seed value and uses a series of mathematical operations to arrive at a number. However, a computer can never generate a truly random number and as such, computer-generated random numbers are said to be pseudorandom.

Synoptic Link

C#, Java, Pascal, Delphi, Python and VB.Net are all examples of **High level languages.**

Different types of programming languages are covered in **Fundamentals of computer systems.**

It's important that you make yourself familiar with random number generation in your chosen programming language.

Exception handling

When an error occurs in program code, an "exception" is said to be thrown. This could be caused by using the wrong data type, attempting to divide by zero or attempting to access a non-existent element in an array to name a few examples.

Volatile

Data which is not stored permanently and will therefore be lost should the program crash.

Once an exception has been thrown, the computer has to handle the exception to avoid crashing. It does this by pausing execution of the program and saving the current volatile state of the program on the system stack before running a section of code called a catch block.

This code will prevent the program from crashing and might inform the user that an error has occurred. Once the exception has been handled, the program uses the system stack to restore its previous state before resuming execution.

Subroutines

A subroutine is a named block of code containing a set of instructions designed to perform a frequently used operation. Using subroutines reduces repetition of code and hence makes code more compact and easier to read.

Synoptic Link

Functions are a type of **subroutine.**

Functions are covered in more detail later in this document.

Both functions and procedures are types of subroutine and can be called by writing their name in a program statement. While both functions and procedures can return a value, functions are required to whereas procedures may not.

Parameters of subroutines

Parameters are used to pass data between subroutines within programs. Specified within brackets after a subroutine call, parameters hold pieces of information that the subroutine requires to run.

```
Length ← USERINPUT
Width ← USERINPUT
OUTPUT CalculateArea(Length, Width)

SUBROUTINE CalcualteArea(x, y)
     RETURN x * y
ENDSUBROUTINE
```

The subroutine `CalculareArea` in the pseudocode above takes two parameters, `Length` and `Width`. It then returns the product of the two values.

The actual value passed by a parameter is called an argument. If a rectangle with sides of height 4 and width 6 was input into `CalculateArea`, the parameters `Length` and `Width` would have arguments 4 and 6 respectively.

Returning values from a subroutine

A subroutine can return a value. One that always returns a value is called a function, but don't think that procedures can't return a value, they can (but don't always).

Subroutines that return values can appear in expressions and be assigned to a variable or parameter.

```
Length ← USERINPUT
Width ← USERINPUT
Area ← CalculateArea(Length, Width)
OUTPUT Area

SUBROUTINE CalcualteArea(x, y)
     RETURN x * y
ENDSUBROUTINE
```

For example, in the pseudocode above, the variable `Area` is assigned to the subroutine `CalculateArea`. The value taken by the variable will be the value returned by the subroutine.

Local variables in subroutines

A local variable is a variable that can only be accessed from the subroutine within which it is declared. They only exist in the computer's memory when their parent subroutine is executing. This makes local variables a more memory efficient way of storing data than using global variables, which are discussed below.

Global variables

In contrast to local variables, global variables can be accessed from any part of a program and exist in memory for the entire duration of the program's execution.

Local variables can be given the same identifier name as global variables, although this is generally considered bad practice. When the local variable's value is changed, the global variable's value remains the same.

The role of stack frames in subroutine calls

Stack frames are used by computers to store return addresses, parameters and local variables for each subroutine call that occurs during the execution of a program.

If one subroutine calls another, nesting is said to occur. Each subroutine call will be pushed onto the computer's call stack in the form of a stack frame before the subroutine's code begins to execute. When the nested subroutine finishes executing, the stack frame is popped from the call stack and the computer uses the information to return to execution of the previous subroutine.

```
1     Name ← USERINPUT
2     OUTPUT Greeting(Name)
3
4     SUBROUTINE Greeting(Name)
5           TimeOfDay ← GetTimeOfDay()
6           RETURN "Good " + TimeOfDay + Name
7     ENDSUBROUTINE
8
9     SUBROUTINE GetTimeOfDay()
10          IF Time < 12:00 THEN
11                RETURN "morning"
12          ELSE
13                RETURN "afternoon"
14          ENDIF
15    ENDSUBROUTINE
```

Synoptic Link

Push and **pop** are terms used with **stacks**.

Stacks are covered in more detail in fundamentals of data structures.

Stack Frame Example

When the pseudocode above is run, the subroutine `GetTimeOfDay` is called from within the subroutine `Greeting` and nesting occurs.

The first stack frame to be pushed onto the call stack is for the subroutine `Greeting`

:

Call Stack

Subroutine Name	Return Address	Parameters	Local Variables
Greeting	Line 2	Name = “Sarah”	Null

When the subroutine `GetTimeOfDay` is called, another stack frame is pushed onto the call stack and is placed on top of the frame representing `Greeting`:

Call Stack

Subroutine Name	Return Address	Parameters	Local Variables
GetTimeOfDay	Line 5	Null	Null

Subroutine Name	Return Address	Parameters	Local Variables
Greeting	Line 2	Name = “Sarah”	Null

When the subroutine `GetTimeOfDay` completes, its corresponding stack frame is popped from the call stack:

Call Stack

Subroutine Name	Return Address	Parameters	Local Variables
Greeting	Line 2	Name = “Sarah”	Null

The computer now takes the stack frame at the top of the stack and goes to the specified return address. Any parameters and local variables are restored.

Once the subroutine `Greeting` has completed, the final stack frame is popped from the call stack, leaving it empty.

Call Stack

Recursive techniques

A recursive subroutine is one which is defined in terms of itself. This means that somewhere within the recursive subroutine, there is a call to the subroutine itself.

Any recursive subroutine must meet have a stopping condition (called a base case) which must be met at some point in the execution of the program.

Base case

The terminating situation in recursion that does not use recursion to produce a result.

If an algorithm calls itself, but doesn't have a base case, it will never terminate. This will cause a stack overflow as more and more stack frames are pushed onto the call stack.

Synoptic Link

A **stack overflow** occurs when too many items are pushed onto a **stack**.

Stacks are covered in more detail in **fundamentals of data structures**.

The pseudocode below is an example of a recursive algorithm which can be used to calculate the factorial of a number, passed to the subroutine as a parameter.

```
1    SUBROUTINE Factorial(Value)
2         IF Value = 0 THEN
3              RETURN 1
4         ELSE
5              RETURN Value * Factorial(Value - 1)
6         ENDIF
7    ENDSUBROUTINE
```

The algorithm can be called recursive because it calls itself on line 5 (shown in bold). The algorithm's base case is when `Value = 0`. In this case, the algorithm doesn't use recursion to return a result.

When a problem can be solved recursively, it can often also be solved using iteration. While iterative solutions are often easier to program, recursive solutions can be more compact in code.

For example, the two pseudocode algorithms below represent two possible approaches to the binary search algorithm. The first approach uses iteration while the second uses recursion.

Synoptic Link

Binary search is a type of **searching algorithm**.

Binary search is explained in **searching algorithms** under **fundamentals of algorithms**.

Iterative Binary Search Example

```
SUBROUTINE BinarySearch(Array, ToFind)
     Low ← 0
     High ← Length(Array)
     Middle ← 0
     WHILE Low <= High
          Middle ← (Low + High) / 2
          IF Array(Middle) > ToFind THEN
               High ← Middle - 1
          ELSEIF Array(Middle) < ToFind THEN
               Low ← Middle + 1
          ELSE
               RETURN Middle
          ENDIF
     ENDWHILE
ENDSUBROUTINE
```

Recursive Binary Search Example

```
SUBROUTINE BinarySearch(Low, High, Array, ToFind)
     Middle ← (Low + High) / 2
     IF ToFind < Array(Middle) THEN
          RETURN BinarySearch(Low, Middle, Array, ToFind)
     ELSEIF ToFind > Array(Middle) THEN
          RETURN BinarySearch(Middle, High, Array, ToFind)
     ELSEIF ToFind = Array(Middle) THEN
          Return Middle
     ENDIF
ENDSUBROUTINE
```