Variables and others

1. Introduction to programming techniques

Any computer program needs to be able to do the following:

- Set data values that will not change as the program runs.
- Keep track of data that does change as the program runs.
- Carry out calculations.
- Compare data.
- Allow the user to enter data.
- Display data to the user.

Every programming language does this using a common set of tools, called constructs. We will discuss some of the most common of these over the next few pages.

Variables and others

2. Variables

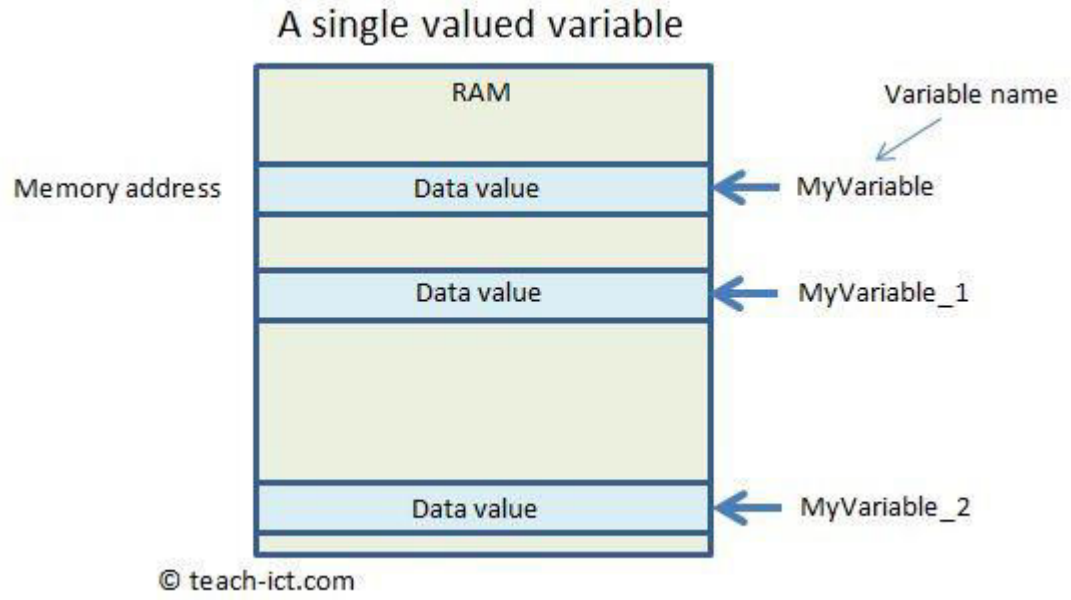
One of the tasks of a computer program is:-

To be able to keep track of changing data.

This is the purpose of a '**variable**'.

A **variable** is a location in memory that holds one or more values. It has a label or name to identify it and its values can be changed as the program runs.

The programmer gives each variable a name to identify it. This name is used by the program to find the memory location where the variable's data is stored, in order to read or modify it.



z = 10

The simplest type of variable is a single value, for example:

The line of code above is placing the value '10' into a variable called 'z' which is located somewhere in memory.

One of the handy things about having a *name* to a variable is that you do not need to care *where* in memory 'z' is located - the operating system is taking care of all that.

If you want to use 'z' a bit later in the code, then you use its name, like this

```
Line 10:    z = 10                              // assign 10 to z
Line 11:    another_variable = 1                // assign 1 to another_variable
Line 12:    new_variable = z + another_variable // do a calculation and store in new_variable
```

Line 12 is using the 'z' variable as part of a calculation.

Variables can get more complex than this. For example a variable might be a list of values (called an array):

y = {10, 16, 1}

This is discussed further in a section covering arrays.

Variables and others

3. Naming variables

It is good practice in programming to give variables a sensible name that will help explain its purpose.

What you should *not* do is put spaces into a name even if the language allows it, as it can be confusing and it is too easy to mistype it later.

Single letter

A single letter variable is often used to just count. For example the code below is using 'x' to count in a loop.

```
    Line 10:      FOR x = 1 TO 10       // x is being used to count
    Line 11:         ..... some code
    Line 12:      NEXT x               // increment x and go back to the
FOR line
```

Don't worry about what a loop is - that is covered later. But the point is that 'x' has no other use than to count and so does not need a complicated name.

Longer names

If the variable has a specific purpose, then give it a relevant name. For example, the code below is asking for an age to be input, so naming the variable 'age' makes it obvious what the it contains

```
    Line 10:      age = input("Please enter your age:")
    Line 11:      output "Your age is " + age
```

Using capitals

Some variables are better to have two words in the name to describe it. In which case capitalise the first letter of each word. For example

```
    Line 10:      FirstName = "Laura"
    Line 11:      SecondName = "Bradshaw"
```

Using capitals makes it much easier to read than firstname or secondname.

Alternative method

If you don't like to use capitals then a good alternative is to use underscore to separate the words in the name. For example

```
    Line 10:      first_name = "Laura"
    Line 11:      second_name = "Bradshaw"
```

Variables and others

4. Constants

Another task of a computer program is to:-

Set data values that will not change as the program runs.

This is the purpose of a '**constant**'.

A **constant** is a location in memory that holds a value. It has a label or name to identify it and its value *cannot* be changed as the program runs.

The key difference between a variable and a constant is that the value held within a constant cannot be altered once the program is running. In order to change the value, the source code itself would have to be modified by the programmer.

Naming a constant

It is good practice in programming to give each constant a sensible name that helps explain its purpose.

A constant is often declared in capital letters to highlight the fact that it is *not* a variable.

If it makes sense to use two or more words in the name, then use underscore to separate them, like this pseudocode :-

```
Constant CHRISTMAS_DAY = "25th December"
```

Variables and others

5. Why use constants?

Legibility

Constants attach a label to a particular value, which is useful when it comes time to look back at the code and figure out what those values meant. For example,

```
Total_Cost = Price * 1.2
```

It is not immediately obvious what the '1.2' is for, or why it is being used. But if you make it into a constant:

```
Constant TAX = 1.2

Total_Cost = Price * TAX
```

Now anyone who reads the code knows that the added value is the tax.

Constants are a convenient way for the programmer to understand the purpose of an unchanging value within the source code.

Single editing location

An advantage of using a constant is that it only needs to be declared in one place.

For example, if you were using a constant called 'TAX' then once set up, the same constant can be applied in multiple places throughout the program:

```
AppleCost = ApplePrice * 1.2
OrangeCost = OrangePrice * 1.2
BananaCost = BananaPrice * 1.2
KiwiCost = KiwiPrice * 1.2
```

If the tax value changed from 1.2 to 1.3, you would have to manually go through and change each entry. This is a waste of time and you could easily miss one, causing the program to give you incorrect data. If you use a constant, however,

```
Constant TAX = 1.3

AppleCost = ApplePrice * TAX
OrangeCost = OrangePrice * TAX
BananaCost = BananaPrice * TAX
KiwiCost = KiwiPrice * TAX
```

it is as simple as changing a single value.

Cannot be changed

As you know, a constant cannot be changed and this, in itself, is advantageous. It means that it cannot accidentally be changed whilst the program is running, as can a variable.

Variables and others

6. Operators

Two of the tasks of a computer program mentioned on the first page are to:-

Carry out calculations.
Compare data

These are some of the most basic actions performed by a CPU. They are so central to how programming works that they are each represented by just one character, called an '**operator**'.

There are several types of operator:

- Arithmetic
- Boolean
- Assignment
- Compound Assignment

Arguments

An operator always acts on at least one item. This item is called the '**argument**' of the operator.

For example the addition operator '+' has two arguments such as

1+2

The arguments are 1 and 2 in this case.

Arithmetic Operators

You are already familiar with many simple arithmetic operations: addition, subtraction, division and multiplication. The symbols used to represent these operations in a program are mostly the same ones you see in a maths textbook:-

+ for addition
- for subtraction
***** for multiplication
/ for division

A maths expression is worked out in a fixed order of operators - BODMAS is the acronym for this rule

- Brackets first
- Order (numbers involving powers or square roots)
- Division
- Multiplication
- Addition
- Subtraction

So an expression such as

$$x = 1 + (3 - 1) - 2^3 + 10/3$$

can be worked out in the correct order

Variables and others

7. Arithmetic Operators cont.

On the last page we discussed common arithmetic operators that you are probably already familiar with, using symbols that you've seen before. There are a few less common arithmetic operations that you will also need to learn.

Exponential

This means raising a number up to the power of another number. The operator symbol for this usually ^ or sometimes EXP

A statement such as

$$x=2^{3}=23$$

means 2 raised to the power of three which returns the answer 8 (2 x 2 x 2). Exponentiation always raises the number on the left by the power of the number on the right.

DIV (Quotient)

DIV is an operator that calculates how many times one number can be divided by another, discarding remainders.

For example:

$$x = 20 \text{ DIV } 3$$

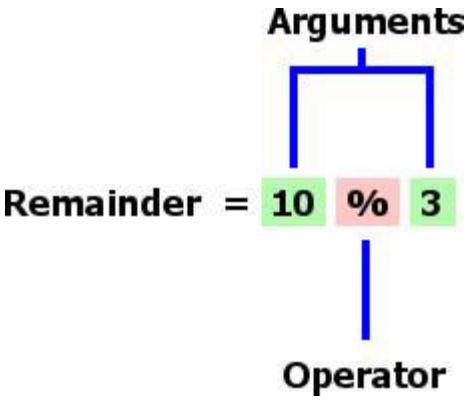
DIV works out how many times the argument on the left (20) can be divided by the argument on the right (3), and returns the result.

In this case x = 6.

MOD (Modulus)

The "Modulus" operation does the same thing as DIV, but, instead of returning the number of divisions, it returns the *remainder*. The symbol for this is often % or sometimes MOD.

For example:



or without labels:

```
Remainder = 10 % 3
```

Since 3 goes into 10 three times with **1** left over, the modulus operation returns "**1**" as the answer.

Variables and others

8. Boolean Operators

A boolean operator carries out a logical operation on its arguments.

Examples of boolean operators include:

AND - returns TRUE only if both arguments are true
OR - returns TRUE if either arguments are true
NOT - return the inverse of the boolean argument
XOR - Exclusive OR. Returns true if either argument is true but not both.

Boolean operators return **True** or **False**.

They are widely used in program flow control by placing then in conditional statements, for example

```
IF (x AND y) THEN
          ..... Do this if true
END IF
```

There are no completely standard symbols for boolean operators in programming, but popular ones are:

&& for AND

|| for OR

! for NOT.

Variables and others

9. Assignment Operators

As a reminder, there are several types of operator:

- Arithmetic
- Boolean
- Assignment
- Compound Assignment

Assignment operators *assign* values.
The simplest assignment operator is "=". It copies the value of its *right* argument into the argument on its *left.*

```
MyVariable = 2
```

This assigns the value 2 to the variable named 'MyVariable'.

The assignment operator can also copy a value from one variable to another. For example

```
MyVariable = AnotherVariable
```

Because of the right-to-left rule, that statement is going to copy the value of the right variable AnotherVariable into the variable on the left MyVariable - not the other way around.

Because of the assigment operator, you can write code to carry out calculations and assign the result to a single variable, like this

```
MyVariable = AnotherVariable + 3 + Age
```

In the code above the + operator is adding up the arguments on the right hand side of =, then the = operator assigns the result to the argument on its left - MyVariable.

Variables and others

9. Compound assignment operators

Some operations are so common that operators have been created to carry out two of them at once. For example a common operation might be something like:

```
a = a + b
```

This updates the variable a by adding b to it. But because there are so many programs that do this kind of thing, a shortcut has been set up. Instead of typing out the full operation to add two things together, you can use the combined "+=" operator instead:

```
a += b
```

This makes the source code a bit more concise.

The compound assignment operators include:-

```
a += b  (a = a + b)
a -= b (a = a - b)
a *= b (a = a*b )
a /= b (a = a/b)
```

Variables and others

11. Input

Another of the tasks of a computer program that we mentioned on the first page was to:-

Allow the user to enter data

The process of entering data into a program is called 'inputting', and while the exact command words to do this vary from language to language, most approach it in similar ways.

For example in Python you might use the **raw_input()** command.

Input commands asks the user to enter data and gives the program access to what they enter. That data can then be assigned to a variable (using an assignment operator), or used for other tasks.

Many input commands, including raw_input, also allow the programmer to include a prompt for whoever is entering the data. For example

```
FirstName = raw_input('Enter your name')
```

This code prompts the user to enter their name. Whatever they type in is then assigned to the variable FirstName

Variables and others

13. Operator summary

Operator Summary	
+	Addition
-	Subtraction
*	Multiplication
/	Division
^ or exp	Exponential - raising a number by the power of another number
% or MOD	Modulus - the remainder when a number is divided by another number

DIV	Quotient - the whole number of times a number can be divided by another number
==	a == b checks if the two arguments are equal - it returns TRUE or FALSE
=	assigns the right argument to the left argument e.g a = b assigns value of b to a
>	greater than
<	less than
>=	greater than or equal to
<=	less than or equal to
!= or <>	Is not equal to

+=	Compound operator e.g. a += b is the same as stating a = a + b
-=	Compound operator e.g. a -= b is the same as stating a = a - b
*=	Compound operator e.g. a*=b is the same as stating a = a * b
/=	Compound operator e.g. a/=b is the same as stating a = a / b

Arrays

1. Introduction to Arrays

There are many different ways to organise data, and each way is called a "data structure". One such data structure is called an 'array'.

For example, say you wanted to to store the names of players in a football team. The team has 11 players.

One way, is to store the names as independent variables,

Like this:

```
player1 = "John Smith"

player2 = "Harry Potter"

player3 = "Jimmy Carr"
  ... and so on
```

So there would be eleven variables. The problem with this is it does not show that there is *relationship* between the names - that they are all part of the

same football team. This is why an array data store is useful - it allows related data item to be stored as a group.

Arrays

2. One dimensional array

An alternative to storing all data as separate variables is to use an array.

The array is given a single name and is associated with each of the entries in it. For example:

```
football_team = ("John Smith", "Harry Potter", "Jimmy Carr", ... )
```

Football_team is an example of a **one-dimensional array**. In this case it is the names of a football squad. It will have 11 items in it.

Now let us think about storing a collection of 5 numbers. This is written in pseudocode:

```
MyArray = (3,56,21,5,7)
```

Each item has an **index** (position in the array). The first item (or **element**) is at position zero. Arrays usually begin with position zero, not one.

An **array** is a collection of related data. Each piece of data within the array is an **element**. The position of each element within the array is pointed to be its **index**. Arrays usually begin at index 0, not index 1.

In order to access an element of an array, you use its index

Like this

```
MyArray = (3,56,21,5,7)
print MyArray[1]
```

This will print out '56' because that is the element at index 1.

Updating an array

You can also change the value of an element by using its index, like this

```
MyArray = (3,56,21,5,7)
MyNumber = 11
MyArray[4] = MyNumber
```

This will replace whatever was in position 4 in the MyArray array with "11". So we are replacing the number 7 with the number 11

Arrays

3. Static arrays

There are two ways of determining the length of an array. The first is to define exactly how long the array can be, and not allow it to get any bigger or smaller while the program runs. This is called a *static array*.

A **static array** has a defined length which does not change.

The length of a static array is declared in advance, like this

```
array MyList[5]
```

This is declaring to the program that the array named MyList requires enough memory to store five items. Even when it is empty of any elements or values, it is still given the same amount of memory.

Declaring static arrays whenever you can is useful to compilers, as it helps the compiler set aside the correct amount of memory in the final executable file. It will also run faster than a dynamic array.

Arrays

4. Dynamic arrays

A *dynamic array* does not have a definite size as the program runs. It is declared by using a pair of empty brackets. A dynamic array grows and shrinks as items are added or removed from it.

Like this

```
array MyList[]
```

The length of a **dynamic array** can change as a program runs.

Dynamic arrays are very useful in situations where you do not know before hand how many items it will need to contain. For example, an user may be asked to enter a list of their friends. There is no way to predict the size of the list.

The disadvantage of a dynamic array is they run slightly slower than a static array and they use an unpredictable amount of memory - if they use too much, then the program crashes.

Getting the length of an array

Most programming languages allow you to get the length of an array while the program is running, whether that array is static or dynamic. This is useful because the length can be used as part of an iteration loop, like this

```
array MyList[5]
FOR x = 0 TO MyList.length - 1
  print MyList(x);
NEXT x
```

The syntax of the command varies from language to language, but it is often done by attaching a .length property to the array name

Like this

```
array MyList[5]
what_is_length = MyList.length
```

In this case what_is_length variable now contains 5 because that is the length that was returned by MyList.length.

Arrays

5. Arrays and iteration

The arrays we've been looking at so far have been filled (*initialised*) with values by just stating them as a list

Like this

```
MyList = (2,6,7,3)
```

This is fine for very small lists. But what if you needed to load ten thousand items into the array? It is not practical to just type them as a line of code.

Or what about if you were storing input values that the user enters as the program runs?

The answer is to get the program itself to fill the list, using an iteration loop like a FOR or a WHILE

The same trick can be used to get things back out of a list. For example, when you want to print every item in a long array, you could use a FOR loop like in this pseudocode:

```
01 FOR x = 0 TO MyList.length - 1
02     PRINT MyList[x]
03 NEXT x
```

Arrays

6. Inputting into an array

Let's consider a problem:

> Write a program that requests the user to input a list of their favourite animals. The user types in 'end' to indicate that the list is complete. This data needs to be stored and then printed out once the input is complete.

Solution

As the data is a list, an array should be used. And as the list is open-ended, a dynamic array should be used.

The first lines of pseudocode are:

```
01  array favourite_animals[]
02  count = 0
```

Line 1 declares a dynamic array and the second line initialises a variable named count which will be used with the array.

A DO - UNTIL loop will be used to keep asking the user for their input

```
03 DO

04   animal = INPUT("Enter animal name or type end")
05   IF animal != 'end' THEN
06      favourite_animals[count] = animal
07      INCREMENT count
08   END IF
09 UNTIL animal == 'end'
```

Comments:

- Line 3 is the start of the DO loop
- Line 4 asks for user input and stores it in animal
- Line 5 is checking if end has been entered, if not then the array is updated
- Line 6 copies the value of animal into the array favourite_animals at index count
- Line 7 increments the index counter count
- Line 8 is the end of the IF block

- Line 7 the UNTIL checks to see if the word end has been typed in yet, if not loop back to the DO

Once end has been entered, the loop is finished. Now the pseudocode below will print out the animal names

```
08 PRINT "Your favourite animals are\n"
09 FOR x = 0 TO favourite_animals.length - 1
10     PRINT favourite_animals[x] + "\n"
11 NEXT x
```

Comments:

- Line 8 prints out the heading, including a newline character to force a new line
- Line 9 the FOR is traversing the array from beginning to end
- Line 10 prints out the animal name from the array and concatenates a newline character as well
- Line 11 loops back to the FOR

Once all the array has been printed, then program exits the FOR loop.

Complete program

This is the complete program

```
01 array favourite_animals[]
02 count = 0
03 DO
04   animal = input("Enter animal name or type end")
05   favourite_animals[count] = animal
06   increment count
07 UNTIL animal != 'end'
08 PRINT "Your favourite animals are\n"
09 FOR x = 0 TO favourite_animals.length - 1
10     PRINT favourite_animals[x] + "\n"
11 NEXT x
```

Arrays

7. Two dimensional array

So far, we have discussed how to use a one dimensional array to hold a list of items. Each element of the array is accessed by a single index value, for example MyArray[3]

These are one-dimensional arrays. But you can also have a **two dimensional array**, also known as a **matrix**. Where one-dimensional arrays are lists, two dimensional arrays are tables.

	Column 0	Column 1	Column 2
Row 0	45	34	65
Row 1	56	23	43
Row 2	67	13	21

To locate an item in this table *two pieces* of information needs to be provided - the row number and the column number. For instance, the value "23" is at row 1 and column 1.

Handling data in this way is so common that many computer languages support the 2-dimensional array.

The two dimensional (2D) array needs two indexes to locate any single item, just like the table above.

The 2D array looks like

```
My2DArray [3,4]
```

The pseudocode above is declaring a 2D array made up of 3 rows and 4 columns. A 2D array can be used to store tabular data very efficiently.

2D array is ideal for storing any data that has been arranged in a grid - for example the computer screen you are reading right now is made up of pixels, and any pixel can be identified by its vertical and horizontal location.

Arrays

8. Array functions

There are a number of common actions that can be done on arrays. These include

- append - add an item to the end of an array
- insert - insert an item at a specific location in the array, all the data above is moved up one location
- delete - remove an item from a location, then all the data moves down one place.
- split - split an array at a given point and create two new arrays.
- slice - take a chunk of data out of an array to form a new array, the original array remains intact

Many computer language have built-in commands that can carry out these types of actions.

If the action is not available in that language, then a procedure can be written to carry it out.

Arrays

9. Summary

- An array is a type of data store or structure
- An array is used to store a number of related items
- An array is made up of elements
- The position of an element in an array is defined by its index
- An array that uses a single index to locate an element is called a one-dimensional array
- An array is usually made up of a single data type - number, string etc
- A static array has its size declared before being used and is not changed
- A dynamic array has a variable size depending on how many elements it contains at any point
- Iteration loops are used to traverse an array
- An array can be used to store input values from an user
- An array that needs two indexes to locate an element is called a two-dimensional array
- A two dimensional array is useful to store tabular data
- There are a number of standard functions that can be carried out on an array

Sequence Selection Iteration

1. Introduction

Although there are many computer languages, most are built from the same basic "building blocks". These building blocks are called *programming constructs*, and there are three of them that we have to consider for this module.

These are:

- Sequence - the order in which things happen
- Selection - choosing between possible actions
- Iteration - repeating actions

Sequence Selection Iteration

2. Sequence

Sequence in a computer language means that programming statements are set out one after another. Unless the computer is told otherwise, it will carry out each instruction in turn.

Therefore a 'sequential language' such as Python or 'C' sets out source code instructions one after another.

In pseudocode, each instruction is set out like this

```
Do this
Next do this
Then do this
....
```

Once translated into a sequential programming language, it may look more like this:-

```
x = 3
MyVariable = 34
New_Variable = x + MyVariable
```

Usually the *order* of a sequence is important. For example the pseudocode below cannot work because New_Variable is trying to
use MyVariable and x before they are given a value

```
New_Variable = x + MyVariable
x = 3
MyVariable = 34
```

Sequence Selection Iteration

3. Selection: Comparisons

Other than for the most basic tasks, most programs involve having to make choices. Maybe you want to do one thing if a condition is true and another if it is not. The process of determining which choice to take is called '**selection**'.

Comparison operators

Selection always involves a *comparison* operation, where you check whether certain conditions are met. The comparison operators are

Meaning	Symbol
Greater than	>
Less than	<
Greater than or equal to	>=
Less than or equal to	<=
Not equal to	!=
Are the same	==

The last one (==) is often confused with the equal sign =. They are not the same thing.

If you use =, you are *setting* (aka *assigning*) one value to be equal to another.
If you use ==, you are *comparing* one value to another to see if they are equal.

For example if you say "chalk = cheese", then you are telling the computer that chalk is cheese. If you say "chalk == cheese", you are asking the computer to check whether chalk is cheese.

Comparison - Boolean operators

This is a **logical** comparion, the result is either logical TRUE or logical FALSE.

The comparison is laid out for the AND operator as

```
Expression 1 AND Expression 2
```

The OR operator is similar with

```
Expression 1 OR Expression 2
```

The NOT operator only uses one expression

```
NOT (Expression)
```

The boolean operators are

Meaning	Symbol
Both sides must be true for the result to be true	AND
Either or both sides are true for the result to be true	OR
The result is the opposite of the expression. If expression is true, the result is false. If the expression is false, the result is true	NOT

For example (Expression 1 AND Expression 2) the comparison leads to a TRUE result only if both expressions are also TRUE, otherwise the comparison leads to FALSE

And for the OR operator, the statement (Expression 1 OR Expression 2) is TRUE if either one or both is TRUE. If both expressions are FALSE then the comparison is also FALSE

Combine comparisons

You can combine size operators with a boolean operator to make two comparisons in the same statement, like this

```
(a < 2) AND (b > 4)
```

The boolean AND operator means that both expressions need to be true for the evaluation to be true.

Sequence Selection Iteration

4. Selection: Conditionals

Conditional statements use the comparison operations described on the last page. They instruct a program to perform different actions depending on the results of that comparison.

IF

The most basic conditional statement is IF. In pseudocode, you will see IF statements written like this:

```
IF (expression) THEN
    ... Do this
END IF
```

In the OCR exam reference language it is

```
if ... then
elseif ... then
else ...
endif
```

The start of the statement is marked with an IF, and the end of it with an END IF. You can instruct the program to do any number of things between the two.

Do you notice the `expression` between IF and THEN? The expression is a comparison that can only be true or false. If the expression is true then the source code between the IF and END IF is run. If the expression is false then the program jumps to the next statement after the END IF.

Example pseudocode

```
YourAge = input("Enter your age in years")

IF YourAge <= 16 THEN
     PRINT 'You are still attending school'
END IF
```

The first line is asking the user to enter their age.

The IF uses the expression YourAge <= 16 to determine if the PRINT statement should be run or not.

i.e. if 16 years old or younger then the expression is true and the print statement is carried out.

Once you translate the pseudocode into actual source code, the formatting of IF statements will depend on the exact syntax of the language you are using. But the way they they are used remains the same.

Sequence Selection Iteration

5. IF and ELSE

IF is the most basic conditional statement, but it can be built upon to allow for more complex types of selection. For example

IF .... ELSE ....

Let's say you wanted a program to print one thing if one condition is true, and another if it isn't. You *could* do this by using two separate IF statements, one for true and one for false.

But a more efficient way is to combine the two conditions together. This is done using ELSE.

For example

```
YourAge = input("Enter your age in years")
IF YourAge <= 16 THEN
     PRINT 'You are still attending school'
ELSE
     PRINT 'You are more than 16 years old'
END IF
```

This code will do one thing if your age is 16 or less, and another if your age is over 16.

Just as with basic IF statements, you can put as many commands between the start of an ELSE statement and the END IF as you want to.

Nested IF

The nested IF allows a number of expressions to be evaluated one after another. If any (or even all) are true then that block of code is exectuted. A nested IF uses the ELSE IF statement.

For example

The grade of a student depends on the marks gained in an exam. The grade bands are

- Grade C if the marks are less than or equal to 20%
- Grade B if marks are more than 20% and less than or equal to 60%
- Grade A if marks are more than 60%

The pseudocode below makes use of nested IFs to work out the grade

```
Mark = input("Enter your exam result as a percentage")
IF Mark <= 20 THEN
     PRINT 'You obtained a C grade'
ELSE IF Mark > 20 AND Mark <= 60
     PRINT 'You obtained a B grade'
ELSE
     PRINT 'You obtained an A grade'
END IF
```

Notice you don't need to test for the A grade because all other possible grades have been accounted for by the earlier IFs, so, by the time the program reaches that point, only A grades remain.

In the OCR exam reference language the statements are

```
if ... then
elseif ... then
else  ...
endif
```

Sequence Selection Iteration

6. CASE

IF is a very versatile type of conditional statement. But once you begin having to nest multiple IF statements inside one another in order to match a complicated set of conditions, the code can start to become very awkward to read and difficult to follow.

Some programming languages solve this issue by introducing a different type of conditional statement: SWITCH / CASE.

SWITCH / CASE allows the programmer to set up a list of possible options, and jump straight to the one that matches the current conditions.

Like this:

```
Number_of_Pets = input ('How many pets do you own?')


SWITCH Number_of_Pets

  CASE  0: print "You should get a pet."
  CASE  1: print "What type of pet do you have?"
  CASE  2: print "I hope your two pets like each other."
```

```
DEFAULT
      print "That is a lot of pets!"
END SWITCH
```

Just like an IF statement, SWITCH / CASE accepts an **expression** (in this case, "Number_of_Pets") and evaluates it. But CASE can handle multiple options with a single check of the conditions.

If there is no match then the DEFAULT statement is run.

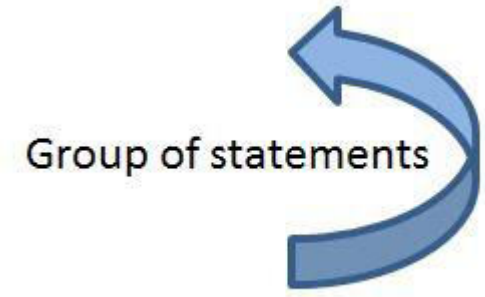
The CASE statement is good for selecting from a list of choices.

In the OCR exam reference language the format is

```
switch ... :
   case ... :
   case ... :
   default :
endswitch
```


7. Iteration

The third type of program construct is called iteration.



An algorithm often needs to perform the same task a number of times. This number might be set in advance, or it might have to repeat the task until a particular condition is met. This repetition of a task is called **iteration**, and there are three main ways to approach it.

- The WHILE loop
- FOR loop
- REPEAT (or DO) loop

These are discussed on the next three pages.

8. WHILE loops

A WHILE loop is given a condition to check. It will keep looping until that condition is *no longer* met. The while loop is a kind of invite to loop forever. You must be really sure that the '*no longer'* condition is eventually met or it is stuck in the loop forever.

In pseudocode, WHILE loops begin with the keyword WHILE and end with the keyword END WHILE. The code to be repeated is often indented to make it easy to tell apart from the rest of the algorithm.

```
count = 1
WHILE count < 10
    PRINT count
    INCREMENT count by 1
END WHILE
```

In this example, the algorithm sets a variable called count to 1. Then the WHILE loop is set to test if count is less than 10. If it is then the PRINT and the INCREMENT pseudocode is applied.
It is important that the algorithm sets a starting value for the variable count because the while loop needs something to evaluate. This is called 'initialising' the variable.
There is no limit to the number of times a WHILE loop can potentially run. So it is important that the condition that the loop is checking will eventually become false. Otherwise the loop carries on forever. In the example above the count value is increased every time it goes around the loop, so eventually it reaches 10 and the loop ends.

Here is another example. The pseudocode below is asking the user for their password. The WHILE loop will not end until the correct password is entered.

```
correctPassword = 'hello'
enteredPassword = ''
WHILE enteredPassword != correctPassword
    DISPLAY 'Please enter your password'
    INPUT enteredPassword
END WHILE
```

In the OCR exam reference language the format is

```
while ...
```

Sequence Selection Iteration

9. Iteration DO loops

A WHILE (*expression*) loop checks *expression* first and afterwards decides whether to carry out a specified task.
But what if you want to ensure that a task is carried out at least once, regardless of *expression*? The way to do this is with a DO UNTIL loop

DO ... UNTIL will keep on looping around until the condition is true.

```
DO
      ... some code
UNTIL (expression) Exit loop if true, otherwise loop
```

DO example

Consider the pseudocode below

```
count = 1
DO
    PRINT count
    INCREMENT count
UNTIL count == 10
```

In this example a variable called count is set to 1. Then it enters the DO construct and runs the code up to the UNTIL and evaluates whether count has reached 10

If it has reached 10 the loop ends otherwise it goes back around.

It is important that the variable count has an initial value, otherwise you cannot be sure what the UNTIL is going to do. It is also important the the statement eventually becomes true, otherwise the loop carries on forever. In this case the count value is incremented every time it goes around the loop and so eventually it reaches 10.

As you can see, WHILE and DO loops are very similar. The key difference is that DO will always carry out its task at least once before checking whether to repeat, whereas WHILE checks first and only then carries out its task.

In the OCR exam reference language the format is

```
do

until ...
```

Sequence Selection Iteration

10. FOR loops

Loops repeat tasks. A **FOR** loop runs for a set number of times. A **WHILE** loop will continue to run until a certain condition is no longer met.

Much like WHILE and REPEAT, FOR loops are used to carry out the same task a number of times. But unlike the other types of loop, FOR loops only run for a set number of times, determined by a 'counter'.

In pseudocode, FOR loops start with the *FOR* keyword and a 'counter' variable. You can call the counter variable anything you like, but usually a very short name like 'i' or 'a' is easier to read.
They end with *NEXT (counter name)*. Every time the loop goes around, the computer automatically increments the counter by 1. So no specific program line is needed to do the incrementing. Once the count is complete, it exits the loop.

For example

```
FOR a = 1 TO 10
     PRINT 'I am looping'
     DISPLAY a
NEXT a
```

The counter variable in this case is called the single letter 'a'. Its starting value is 1 and the final value is 10.

The PRINT and the DISPLAY pseudocode is carried out

then the 'NEXT a' increments the counter

and the algorithm jumps back to the FOR line to test the counter once more.

Once the counter reaches its end point (in this case, 10), the loop ends.

An extension that many languages support is to increment by more than 1 each time, in which case a STEP statement is added to declare the step size.

In the OCR exam reference language the format is

```
for ... to ...

next ...
```

And for steps of more than one it is

```
for ... to ... step ...

next ...
```

Sequence Selection Iteration

11. Summary

- There are three basic constructs in a computer language - sequence, selection and iteration
- Sequence means carrying out instructions one after another
- Selection means a condition is evaluated to determine the next statement to run
- Selection always involves a comparison that results in true or false
- The IF ... ENDIF construct is a selection statement
- More complicated selection statements include nested IF and CASE constructs
- Iteration means loop around a group of statements until some condition is met
- The three iteration constructs are WHILE, REPEAT and FOR
- The iteration expression must eventually evaluate to false to avoid an infinite loop.