Chapter 6 - Storage devices

6.1 Introduction

Storage devices are **non-volatile** devices. That means that when the power is removed from them, for example, when you switch your computer off, they retain their contents (unlike RAM, which is **volatile** – it loses its contents). You can then retrieve the contents next time you switch your computer on. Storage devices can be used to hold operating systems, applications and files, amongst other types of software. They are simply **big suitcases** – used only for storage. We have already seen that when you want to **use** an application kept on a storage device, it has to be moved to RAM before you can start using it. This applies to the operating system, your files and any other category of software. For this reason, RAM is sometimes known as **Primary Memory** whereas storage devices are often referred to as **Secondary Storage** devices.

When comparing and contrasting different storage devices, we could use a number of criteria:

- Whether they were a magnetic or an optical storage media.
- How fast the media can be accessed.
- Whether data can be accessed directly or serially.
- How much data can be stored.
- What the media might typically be used for.
- How commonly used the media is.
- The cost of the media and the cost of the actual device used to read from or write to it.
- Whether the media is read-only or read-write.
- The convenience of carrying the storage device.
- The reliability of the device.
- How many read-write cycles a particular device can take before corrupting.

6.2 How the capacity of secondary storage media is measured - just to remind you

Data storage is measured in 'bytes'. When you talk about storage media, however, you quickly end up talking about thousands of bytes, millions of bytes, millions and millions of bytes and so on. There are lots of different ways of talking about these large numbers. We have met the following summary before:

- 1 Kilobyte (1 Kbyte) is 1024 bytes exactly, or 2^{10} bytes exactly, or about 1000 bytes, or about a thousand bytes.
- 1 Megabyte (1 Mbyte) is 1048576 bytes exactly, or 2^{20} bytes exactly, or about 1000000 bytes (a million bytes).
- 1 Gigabyte (1 Gbyte) is 1073741824 bytes exactly, 2^{30} bytes exactly, or about 1000000000 bytes, or about a thousand million bytes.

So 15 Kbytes is about 15 thousand bytes. 128 Mbytes is about 128 million bytes. 20 Gbytes is about 20 thousand million bytes. More often than not, you don't need to know the exact number of bytes, just an approximation!

6.3 Floppy disks

Floppy disks were once the most common way to store and transport files and software. This magnetic storage medium could typically hold only 1.44Mbytes of information – not even enough storage for a single high-quality MP3 song these days! Although all computers once came with a floppy disk drive, they have now all but disappeared from use.

6.4 Flash drives (USB pen drives)

A USB pen drive is a small, lightweight rewritable storage device that comes with an integrated USB interface. This means that you can carry it around easily and plug it into any computer, to save data to it or to read data from it. In addition, they use the mass storage standard for storing data. This means that no additional device drivers need to be loaded on to a computer so the device can work – the drivers are already there as part of the operating system.

They are rapidly becoming the personal storage device of choice because they can hold a large amount of data, which can be accessed quickly. They use solid state technology, which means they store their data using electronic technology and have no moving parts. This also means that they need very little power. Compared to a CD RW, which typically can take 1000 erase-write cycles, flash drives can take 500 000 (half a million) erase-write cycles and therefore the data is less likely to be lost.

When you want to remove a pen drive from a PC, you should always stop it properly before detaching it. In Windows, you would normally click the little 'Safely Remove Hardware' icon in the bottom right hand corner and then select the device and click Stop. If you don't do this, sooner or later, you will corrupt the device and lose all the data on it!

They have a wide range of uses. They can be used to store or backup large, personal files and especially audio files. They can be used by technicians and network managers to help recover PCs infected with viruses or to transfer configuration files. You can also boot directly from them. This means that you can start up a PC with an operating system, applications and a personal configuration already on your flash drive. A small problem is that they are easily left behind in a public PC and you may not always get them back. It is important to password protect or encrypt any data you would not want others to see.

6.5 Hard disks

A hard disk is a hard magnetic disk or set of disks, each with their own read/write head that moves over the disks as they spin at high speed to read from or write to device. Hard disks are 'direct access' devices - you can go straight to a file on an area of the disk by moving the read/write head to the correct areas without having to go through all the other files first. Compare that to e.g. a magnetic tape, where you have to go through other files to get to the one you want. Hard disks are often part of a computer system, where they are used to store the operating system, files and applications on a computer whilst it is switched off. They are also used as a backup device because they can be portable and because they can store huge amounts of data

A computer or a network server (the main computer that controls a network of computers) can be fitted with a second hard disk known as a **mirror hard disk**. As the name suggests, this second hard disk is used to keep an identical copy of the main hard disk. Then, if the main hard disk fails (and every hard disk, indeed every storage device, will fail sooner or later), you can use the mirror disk to recover your applications and data with the minimum of effort.

6.6 Magnetic tape

Magnetic tapes like hard drives can hold lots of data. They are used typically to backup data files on networks. Network servers often have a backup device fitted in the server that can hold 7 tapes, for example. These rotate automatically each day. The backup process is automated, so that in the middle of the night, data is backed-up on a new tape and the tape gets ejected from the device. The Network Administrator then removes the tape in the morning and puts it in a fire safe. Magnetic tape is not a direct access device. It is a serial access device. If you lost one of your files and asked the Network Manager to recover it from a backup tape, they would have to search through every file until they found the right one. They couldn't go straight to the file in question. For this reason, magnetic tapes are very slow devices, not suitable for fast access applications but ideal for applications where you won't probably need the data - like backup applications! They are also cheap to store data on compared to other types of media. You can store more bytes per penny!

6.7 CD ROM and CD RW

CD ROMs are ideal for distributing software because they hold a lot of data (650 Mbytes or more), are cheap, portable and most computers have a drive that can read CD ROMs. They are direct access media but they do not use magnetic technology! They are optical storage media. They store information on pits in the surface of the CD and then use a laser to scan over the pits to read the data. CD ROMs are read-only. If you want to write and overwrite to a CD then you need a CD RW. This kind of optical, direct access media is ideal for backing up personal files, especially those involving multimedia. It is suitable for this type of application because of their high storage capacity.

6.8 Digital Versatile Disk (DVD)

This optical, direct access, very fast media can hold a lot of data. You can buy 4.7 Gbyte disks in the shops. Compare this to the 650 Mbytes of a standard CD. Most PCs you buy today come with a DVD read-write drive. A DVD player can usually read CDs as well as DVDs. They are typically used for distributing multimedia, especially high quality video.

6.9 Blu-ray

This type of disk is becoming widely used, especially for the distribution of films. Compared to the 4.7 Gbytes DVDs hold, this type of disk can hold much more, up to 100 Gbytes of data and huge companies like Panasonic, Sony, Disney and Dell have added their weight of support to this format.

6.10 SD cards and micro SD cards

These small, solid state cards can hold very large amounts of data and are ideal for cameras, tablet PCs and mobile phones, to hold personal files, pictures, videos and music.

6.11 Cloud storage

Many companies these days do not back up their work onto a physical medium such as a tape or USB pen drive but use internet storage instead. When they want to back something up, the data may or may not be compressed first, to make it smaller. It is then sent over the Internet to a company, who stores it on their computers. This has quite a few advantages. You can set up the backups to automatically happen so no one will forget to do them. You don't need to buy expensive equipment to backup work and you can't lose backups. Of course, you need an Internet connection and some companies aren't happy about sending their valuable data to another company to look after for security reasons. However, this way of storing data, known as 'cloud storage' is becoming increasingly popular. Many cloud storage companies offer a certain amount of free storage to individuals and have integrated cloud storage so that you can work from files and folders directly, as if it were on your own computer. Even better, you can synchronise all of your files across different devices. This means that if you are using a laptop one minute, your phone the next and then your main computer, they will all always have the same, latest copy of your files and folders. This is really convenient for people who often work on different devices as it means less potential problems when working on different versions of the same file on different computers. Companies that you can use to try out cloud storage include Dropbox, Google Drive and Skydrive.

6.12 Compressing data

Data compression refers to the process of 'squashing' data so that you can store more of it in the same space on a storage device. This can be done by using a utility program from within your operating system or using applications such as WinZip. A lot of data sent over the Internet is also compressed because this reduces the amount of time (and cost) of data transfer. It is compressed by the sending computer and de-compressed by the receiving computer. This happens automatically, without the knowledge of the users.

Q1. Complete this table.

Name	Typical storage size	Cost?	SS, optical, magnetic?	Portable?	Direct access or serial?	Typical use?
USB pen drive						
Hard disk						
Magnetic tape						
CD & CD RW						
DVD						
Blu-ray						
SD / micro SD						
Cloud storage						

Q2. What are the benefits of solid state technology, as used in a pen drive?
Q3. What is meant by a WORM device? Use the Internet to find out.
Q4. What kind of technology does a hard drive use?
Q5. Explain what the difference is between serial access and direct access to files.
Q6. Why are floppy disks no longer in widespread use?
Q7. What is the unit of data storage?
Q8. What does SD stand for in SD cards?
Q9. Why is data sometimes compressed before being stored?
Q10. What are the benefits and drawbacks of cloud storage? Use the Internet to find out.

Chapter 7 - An introduction to program design

7.1 Algorithms

The first thing that needs to be done when solving a problem using a programming language is to describe how to solve it! This will then lead the programmer into producing efficient code that will perform the task. There are a number of ways to describe a problem that we wish to write a program for. However we decide to describe a problem, it is known as an '**algorithm**'. An algorithm is simply a list of instructions that describe how to do a particular task. All that needs to happen is that the programmer analyses a task and writes down (or draws) step-by-step how the solution is to solve the problem. A programmer may also decide to write algorithms to describe the problem in the first place, and then write algorithms to describe the solution. Whichever approach is taken, there is some good news when writing algorithms. There are no rules dictating what you have to do and how you have to do it! You can produce an algorithm by writing a list of sentences. You can produce an algorithm by drawing a diagram, for example, a flow chart. You can produce an algorithm by writing instructions using pseudo-code. As long as your argument is structured and logical you can write or draw what you like. However, a popular way of representing algorithms is to use **pseudo-code**.

7.2 Describing a solution using algorithms

When writing algorithms, there are two broad questions to ask. These are 'How can the problem be solved?' and 'What special cases need to be taken into account in the code?' Special cases are situations such as what happens if a record cannot be found, what happens if a file is empty and what happens if the wrong data type is entered? Dealing with special cases is initially down to experience. The more practice you have of writing algorithms and programs, the easier it gets!

7.3 Pseudo-code

Pseudo-code is a cross between English and the keywords used in programming languages. It also makes use of the programming constructs found in programming languages (sequence, selection, iteration) but it has no strict rules that you have to follow. It is used to effectively describe solutions in a near-programming language type of way! This means that the pseudo-code can then be given to a programmer expert in **any** language and they should be able to quickly convert the pseudo-code into that language. This is the big advantage of pseudo-code. It is independent of any particular programming language and can be quickly converted into any language!

7.3.1 An example of pseudo-code

Suppose you need to write an algorithm to find a record in a file. A first attempt at an algorithm might look like this (Don't worry if you don't understand the code. We are only at the beginning of a long path)!

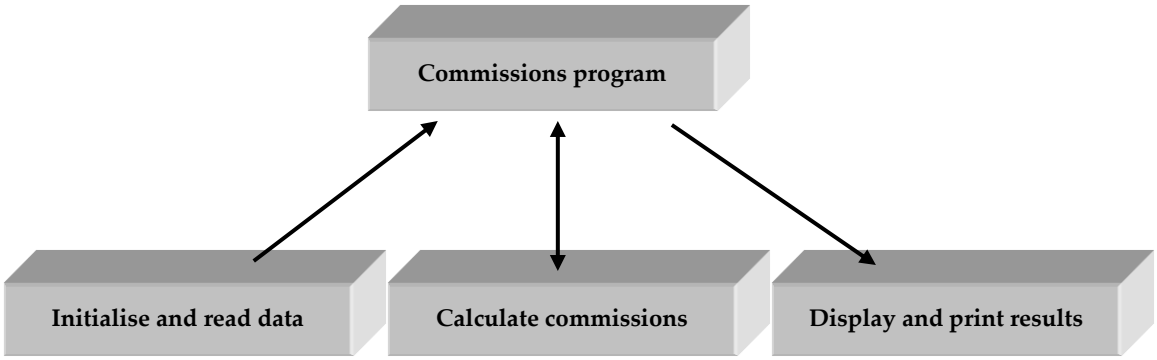
```
RecordFound=FALSE
EndOfFile=FALSE
READ Key field of record you want to find
READ First record in file
WHILE ((EndOfFile=FALSE) AND (RecordFound=FALSE)) DO
BEGIN
        COMPARE record you are looking for with record from file
        IF same THEN
                RecordFound=TRUE
                PRINT "Record found"
        ELSE
                IF EndOfFile Then
                        EndOfFile=TRUE
                        PRINT "End of file reached. Record not found."
                ELSE
                        READ next record
                ENDIF
        ENDIF
END
ENDWHILE
```

7.4 Top-down programming design for procedural languages

When faced with any complex problem, finding a solution may well appear a daunting task at first. However, with the right systematic approach, complex problems can be reduced to many far simpler problems. Each of these simpler problems can then be solved. They can then be re-combined back up to the original problem. Before you know it, that unsolvable, complicated task that you had to find a solution for has been solved! This approach has been around for years and is suitable for any problem that is going to be solved using a procedural programming language. This is a type of programming language that makes a lot of use of self-contained blocks of code that then interact with each other. It is not the best approach to use for other types of languages that you will learn about, such as object-oriented languages. The approach to use for this type of language will be discussed in much more detail later in the book.

In top-down programming, a programmer takes a task and then breaks it down into smaller tasks. They then take each smaller task and break it down further into sub-tasks. They continue to do this until each sub-task is simple enough to understand and program and, ideally, each sub-task performs only one job. The sub-tasks are then programmed as self-contained modules of code. When a problem is broken down into modules, the relationship between one module and the next is clearly defined. If we break down one module into three modules, for example, the relationship between the three modules is clearly defined so that together, they perform in exactly the same way that one big module of code would have performed.

Consider a module that calculates a salesman's commission. You could have one module that does this, but because it performs a number of different tasks, it will be split into three modules. This is shown in the next diagram.



Breaking a problem down using top-down techniques.

The first module is now responsible for initialising the program and reading in values. That is all it does. Its relationship with the main program is that it passes sales figures out to the program. The next module is responsible for doing the calculations. Its relationship with the main program is that it reads in sales figures and passes back commissions due. The third module is the display and print module. Its relationship with the main program is that it reads in commissions due.

The above design could be improved further. We have already said that ideally modules should perform only one function. The 'Initialise and read in data' module could be split into an 'Initialise' module and a 'Read data' module. The 'Display and Print' module could also be split into two modules, one called ‘Display’, which will be in charge of displaying results on a VDU, and one called 'Print' which will be responsible for printing out results. Once modules have been identified, they can then be written. Remember, each module is a block of code that should be self-contained, perform one very specific task and will usually have an interface with the main program.

7.4.1 Different names for ‘modules of code’

Do note that different programming languages call modules of code by different names although they all (very broadly speaking) mean the same thing. One language might, for example, call a self-contained block of code a ‘module’ of code. Another one might talk about ‘procedures’ and ‘functions’. A third one might use the word ‘subroutine’.

Look at the next program. It has the name ‘commission’. Three procedures are written. The actual program is at the end of the code and is simply made up of calls to the procedures that have been written. If you looked at the 'shape' of the final code for the problem just described, it might look something like this:

```
Program commission
Procedure InitialiseAndRead        //this procedure is responsible for initialising variables and reading in data.
Begin
        -------
        -------
End
Procedure Calculate        //this procedure is responsible for calculations.
Begin
        -------
        -------
End
Procedure DisplayAndPrint        //this procedure is responsible for displaying and printing the output.
Begin
        -------
        -------
End
Begin                              //this is the actual program. It is made up of calls to the various procedures.
        InitialiseAndRead
        Calculate
        DisplayAndPrint
End
```

7.4.2 The benefits of top-down programming design

Programs written using a top-down approach produce modules of code. We sometimes refer to this approach as ‘**modular design**’ or ‘**modular programming**’. Whether the modules are called procedures, functions, objects or anything else, they have some distinct advantages when compared to writing one big block of code.

1) It helps get the job done more efficiently because modules of code can be worked on at the same time by different programmers. In addition, it helps because easier modules can be given to less experienced programmers while the harder ones can be given to more experienced ones.
2) It helps program testing because it is easier to debug lots of smaller self-contained modules than one big program.
3) Splitting up a problem into modules helps program readability because it is easier to follow what is going on in smaller modules than a big program.
4) It improves a company's efficiency because self-contained modules can be re-used. They can be put into a library of modules. When a module is needed to, for example, display some values, you don't need to get the programmers to write and test a module for this again. You just re-use a module from the library. Over time, this will save a company a lot of time and money.
5) It improves a Project Manager's ability to monitor the progress of a program. Modules can be 'ticked off the list' as they are done and this will demonstrate some progress. This is far harder for a Project Manager to do if the program has not been split up into modules.
6) It is good for future program maintenance. If a program needs to be changed for any reason, it may be possible simply to remove one module of code and replace it with another.

Whether modules of code are produced using the top-down or bottom-up method, they must be tested.

7.5 Unit testing

Whenever a module of code (or subroutine, or unit, or function, or procedure, or whatever other name you want to give to the block of self-contained code) is written, it needs to be tested. Testing can be done using black or white box testing, as discussed in the chapter ‘Testing your programs and quality control’. Whether you use black box or white box testing, testing an individual module of code is known as ‘unit testing’.

7.6 Integration testing

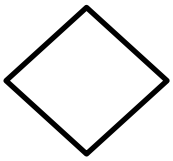
Once modules of code are tested using unit testing, they need to be combined and tested together. It is possible that two fully tested modules (tested individually using unit testing) will produce a problem when they are combined and asked to work together. Bringing modules together and checking that there are no unintentional problems is known as ‘integration testing’.

7.7 Flowcharts

Another way of describing algorithms is to use a flowchart. This is a diagram that uses a set of symbols and lines of flow that link the symbols. If you have used popular control programs such as Logicator, for example, then you will be familiar with flow diagrams. The following symbols are the most common ones used in flowcharts, although there are others and if you read around this topic, you may find slight variations in their use.

The start and end of a flowchart.

Processing of data or calculations.



Making a decision.

A flow of data.

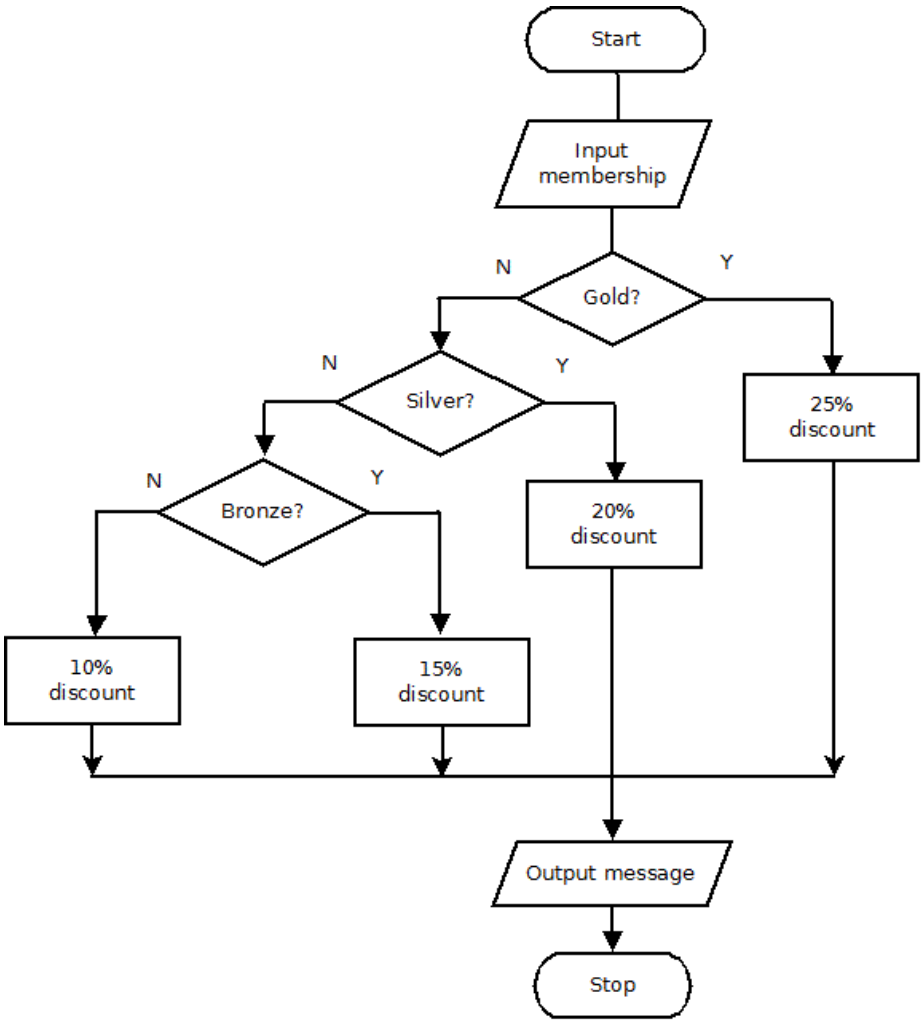
Input or output of data.

A document or report.

A subroutine.

7.7.1 An example of a flowchart

Here is an example of a flowchart that describes the discount given for buying products in a club's shop. The discount given depends upon your membership type.

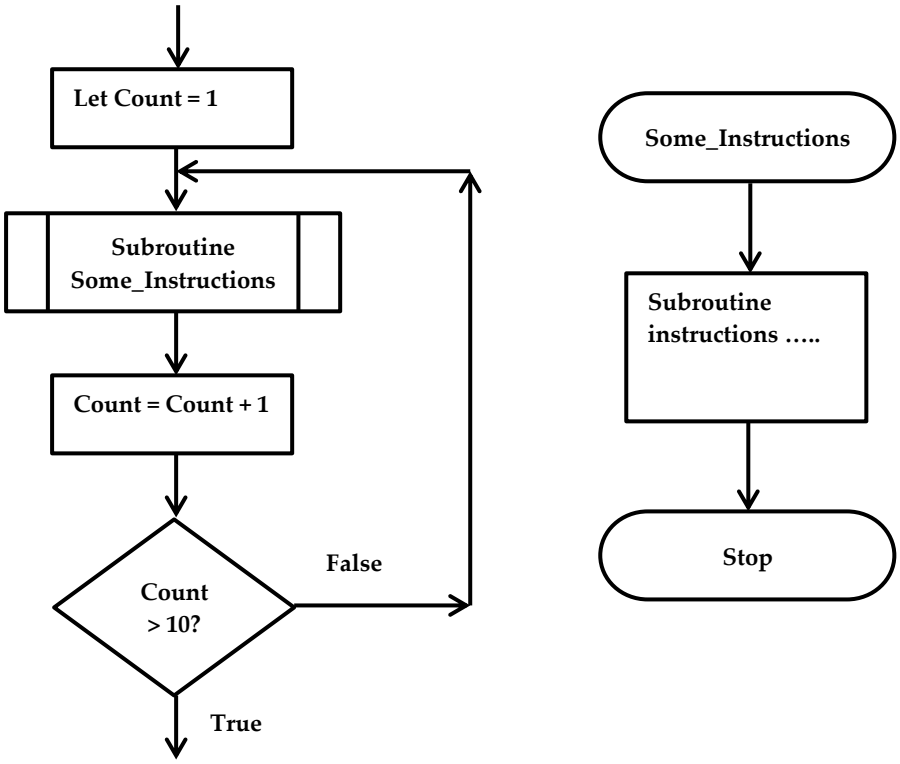


You begin and end the flowchart with a start / stop symbol. You then input your membership type.

- If it is gold, you get 25% discount and a message is displayed stating the discount given.
- If it is silver, you get 20% discount and a message is displayed stating the discount given.
- If it is bronze, you get 15% discount and a message is displayed stating the discount given.
- If you are not a member, you get 10% discount and a message is displayed stating the discount given.

7.7.2 An example of a FOR loop and using subroutines

Suppose you had to carry out a set of instructions a fixed number of times. In programming code, you would use a FOR loop for this. But stepping back for a moment, how might you represent this as a flowchart?



There are some useful tools for making flowcharts around. There is a drawing toolbar in both Word and OpenOffice that can be used for drawing neat flowcharts. Meesoft's 'Diagram Designer' is also excellent (and free). If your school has Logicator, that's also an excellent way of getting experience producing good flowcharts.

Q1. What is an 'algorithm'?
Q2. Explain what is meant by pseudo-code being 'language-independent'?
Q3. What is meant by top-down programming?
Q4. What are the benefits of top-down programming?
Q5. State two other possible names for a 'module of code'.
Q6. What is meant by unit testing?
Q7. Define 'integration testing'.
Q8. What is the flowchart symbol for a decision?
Q9. What is the flowchart symbol for a subroutine?
Q10. What is the flowchart symbol for the input or output of data?

Chapter 8 - Programming constructs

8.1 introduction

When you look at any high-level language, you can identify the features in the language that make it so powerful. Instructions in a program flow one after another in a sequence. However, there are instructions available that will allow blocks of code to be repeated many times. There are also instructions that allow decisions to be made about which code from a selection of code will be executed. The specific instructions provided by a particular language that allow code to be repeated, or code to be selected from a choice of code, are known as **control structures**. If we also include the way that instructions flow one after the other, we can say that there are three types of programming construct we need to know about. These are known as:

- Sequence.
- Selection. (There are two types of selection we need to know about.)
- Iteration, also called repetition. (There are three types of iteration we need to know about.)

8.2 Sequence

There isn't a lot to say about the programming construction 'sequence'. It simply describes one instruction following on from the next instruction. An example of this, in pseudo-code is:

```
INITIALISE Variables
WRITE 'Please enter the name of student'
INPUT Name
WRITE 'Please enter the exam mark of student'
INPUT ExamMark
PRINT Name, ExamMark
```

8.3 Selection

There are two methods of selecting which code from a choice of code will be executed, IF and CASE. We will look at the IF construction first followed by the CASE construction.

8.3.1 IF-THEN-ELSE-ENDIF

Look at the following example:

```
INPUT letter using keyboard
IF (Letter entered = P) THEN
        Do the code in here - call it block A
ELSE
        Do the code in here - call it block B
ENDIF
```

When this code is run, the program waits for the user to enter a letter via the keyboard. When the letter has been entered, it is checked. If the letter P were entered, for example, then the instructions in block A would be executed. If the letter entered was not a P, then the instructions in block B would be done. The construction then ends and the next instruction in the sequence is done.

8.3.2 Nested IF-THEN-ELSE-ENDIF

Not only can you make selections using the IF construct, you can also put them inside each other! When we do this, we say that the IF instructions are **nested**. Nesting is a very useful technique so long as the code is laid out correctly and you don't use too many nested IF statements.

Consider this example that uses nested IF statements.

```
INPUT ExamMark
IF (ExamMark < 40) THEN
        PRINT "You have failed."
ELSE
        IF (ExamMark < 60) THEN
                PRINT "You have passed."
        ELSE
                IF (ExamMark  < 70) THEN
                        PRINT "You have passed with a merit."
                ELSE
                        IF (ExamMark <80) THEN
                                PRINT "You have passed with a distinction."
                        ELSE
                                PRINT "Outstanding! You have passed with honours!"
                        ENDIF
                ENDIF
        ENDIF
ENDIF
```

8.3.3 CASE

You have probably worked out that nested IF statements are fine in moderation, but too many and the code can become hard to follow. Fortunately, there is another type of selection construct that can be used. It usually involves checking **instances** of variables. Consider this example.

```
CASE (variable) OF

        a:  Do these instructions if a is selected;
        b:  Do these instructions if b is selected;
        c:  Do these instructions if c is selected;
        d:  Do these instructions if d is selected;
        e:  Do these instructions if e is selected;
        f:  Do these instructions if f is selected;

ELSE

        Do these instructions;

ENDCASE
```

If the variable used holds an 'a', then you do the instructions after 'a' and then jump to ENDCASE. If it is a 'b', then you do the instructions after 'b' and then jump to ENDCASE, and so on. If it is not any of 'a' to 'f', then you do the instructions in the ELSE part of the construct and then jump to ENDCASE.

This is very convenient. Some languages don't use CASE, though. Python has a construct IF – ELIF – ELIF – ELIF – ELSE, which can be used just like a CASE statement so it is something to look out for. The ELIF (ELSE IF) can be used as many times as needed to test a particular variable.

8.4 Iteration (sometimes called 'repetition')

Iteration is the name given to the construct that repeats blocks of code. There are three types of iteration construct, each with a subtle difference from the other two!

8.4.1 FOR COUNTER=1 TO MAX DO

If you need to call a block of code a **fixed** number of times then you should use a FOR loop. Study the following example.

```
INPUT MAX
FOR COUNTER = 1 TO MAX DO
BEGIN
        PRINT "This is loop number", MAX
END
MORE instructions
```

When the above code is run, a value is entered from the keyboard and assigned to the variable MAX. The FOR loop is entered. The Counter is assigned to 1 and the code between BEGIN and END is done. The COUNTER is incremented and the code between BEGIN and END is done again. This continues until MAX is reached. After the MAX loop is done, the program drops out of the loop and MORE instructions in the program sequence are done.

8.4.2 WHILE (CONDITION) DO ENDWHILE

The FOR construction is used if you want to do a block of code a **fixed** number of times. The WHILE (CONDITION) DO ENDWHILE construction is used if you want to do a block of code a number of times, but you don't know how many! The number of times will be determined by a test **before** the block of code is executed. If the result of the test is TRUE then the code will be run. If the result is FALSE, then you will drop out of the WHILE loop. Study this example of some code used to allow a user to read some instructions and then press a particular key to continue.

```
WRITE "Press C to continue".
READ KeyPress
WHILE (KeyPress NOT EQUAL TO C) DO
BEGIN
        WRITE "Press C to continue".
        READ KeyPress
END
ENDWHILE
```

In a previous chapter, we saw this example. It uses both selection (the IF construct) and iteration (the WHILE construct).

```
RecordFound=FALSE
EndOfFile=FALSE
READ Keyfield of record you want to find
READ First record in file
WHILE ((EndOFFile=FALSE) AND (RecordFound=FALSE)) DO
BEGIN
        COMPARE record you are looking for with record from file
        IF same THEN
                RecordFound=TRUE
                PRINT "Record found"
        ELSE
                IF EndOfFile Then
                        EndOfFile=TRUE
                        PRINT "End of file reached. Record not found."
                ELSE
                        READ next record
                ENDIF
        ENDIF
END
ENDWHILE
```

Look at the code between WHILE and ENDWHILE. The code between these two keywords is the code that is executed each time the WHILE loop is done. This is how the pseudo-code works:

- When this code is run, the WHILE ((EndOFFile=FALSE) AND (RecordFound=FALSE)) DO line is reached.
- The WHILE condition is tested - ((EndOFFile=FALSE) AND (RecordFound=FALSE)) must be TRUE.

- If the (EndOFFile=FALSE) statement is TRUE, and (RecordFound=FALSE) is TRUE, then ((EndOFFile=FALSE) AND (RecordFound=FALSE)) will be TRUE and the code in the loop is run.
- When the code has run once, the loop returns to the WHILE ((EndOFFile=FALSE) AND (RecordFound=FALSE)) DO line
- The WHILE condition is tested again.
- The code in the WHILE loop will continue to run until either the (EndOFFile=FALSE) statement becomes FALSE or (RecordFound=FALSE) becomes FALSE. (You might have to think about this!)
- When this happens, ((EndOFFile=FALSE) AND (RecordFound=FALSE)) becomes FALSE.
- Because the WHILE condition is false, the code will not be executed. The program continues from the next one in sequence after the ENDWHILE.

You should notice that with the WHILE loop, it is possible that the block of code associated with it may never actually get executed! In the ' Press C to continue' example on the previous page, if C is actually pressed then (KeyPress NOT EQUAL TO C) becomes a FALSE statement. This results in the WHILE loop being skipped completely and the program continues on from after the ENDWHILE instruction. The WHILE block of code doesn't get executed, not even once in this case!

8.4.3 REPEAT UNTIL (CONDITION)

This is very similar to a WHILE loop, except the condition is tested at the **end** of the code in the loop! This is a tiny difference, but means that the code will **always be executed at least once**. Compare this to the WHILE loop, where the code might not get executed at all. A program using a REPEAT construct will look like this:

```
TOTAL=0
REPEAT
BEGIN
        READ VALUE
        TOTAL=TOTAL+VALUE
END
UNTIL (TOTAL > 1000)
```

It is important to ensure that WHILE and REPEAT constructs have a way of changing the variable that is being tested to see if the loop should run again. If there is no way of changing this variable, the loop might continue forever. This is known as an 'infinite loop'. In the above REPEAT example, TOTAL = TOTAL + VALUE is used inside the loop to change TOTAL, the variable that gets tested to see if the loop should run again.

Not all languages make use of REPEAT. For example, if you are using Python, then you will only use the FOR and WHILE iterative constructs. This is not at all a problem as you can always construct code to do exactly what you need from those functions and keywords available.

Q1. State the three programming constructs.
Q2. Give an example of the sequence construct.
Q3. Give an example of the selection construct.
Q4. Give an example of the iterative construct.
Q5. What is meant by a nested construct?
Q6. Give an example of a nested IF construct.
Q7. Give an example of a nested FOR construct.
Q8. What is the difference between how a FOR, WHILE and REPEAT construct works?
Q9. What is meant by an infinite loop?
Q10. How do you avoid an infinite loop in a WHILE construct?

Chapter 9 - Procedures and functions

9.1 Procedures and functions

We have already discussed the idea behind top-down programming in a previous chapter. This technique breaks down problems into 'modules' of code. A module is also known as a 'function' or a 'procedure'. Both procedures and functions are modules of code, sub-programs that perform a specific task. Each one is given a name. This name can then be used by the main program to 'call' it when needed. The procedure or function may or may not receive data from the main program when it is called and may or may not pass data back to the main program when it has finished.

A good program design will generally consist of many procedures and functions. These will then be called in turn by a main program. We saw the shape of a well-designed program in an earlier section. We can see the outline again in the next diagram, with an alternative way of viewing the program. Notice that the main program is in fact very simple, consisting only of calls to procedures.

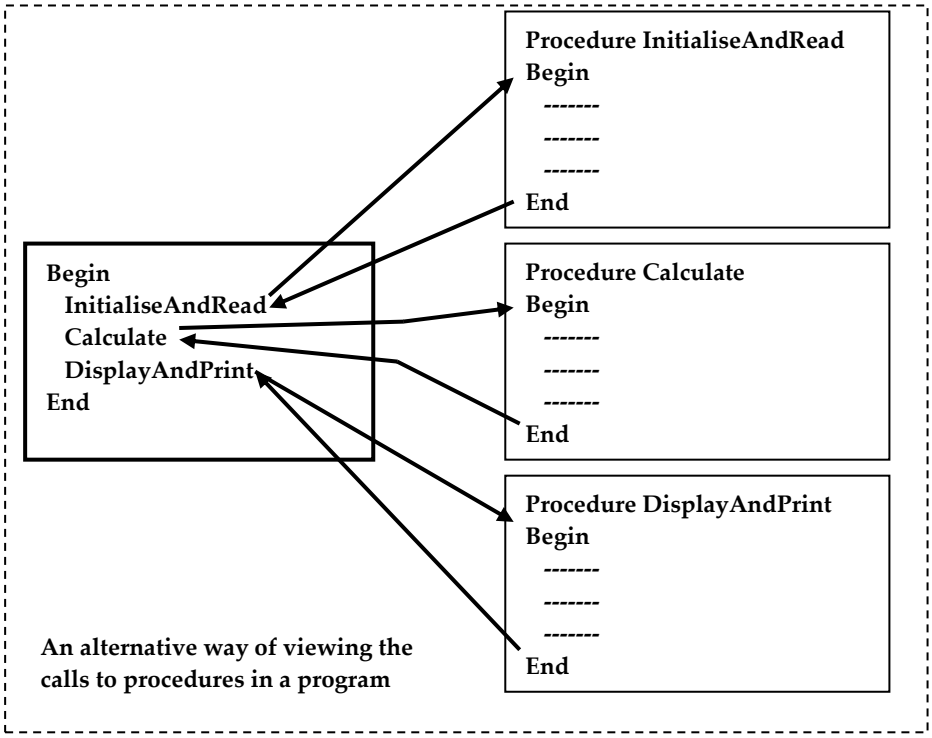
```
Program commission

Procedure InitialiseAndRead
Begin
        -------
        -------
End

Procedure Calculate
Begin
        -------
        -------
End

Procedure DisplayAndPrint
Begin
        -------
        -------
End

Begin
        InitialiseAndRead
        Calculate
        DisplayAndPrint
End
```



The 'shape' of a program written using procedures.

9.1.1 The difference between procedures and functions

Both of these programming units are broadly similar. They are both modules of code. They can both accept parameters (pieces of data that are passed to it when they are called), both can do calculations on data and both can return values that can be used by the main program. Functions, however, traditionally return only a **single value** to the main program. In addition, this value is passed back to the main program using a variable that has the **same name** as the function! Just to confuse things, however, some languages like Python can return multiple values using functions. We will stay with the traditional view for the moment.

If that is difficult to get your head around, look at the recursion example in the next chapter. Each time a function call ended, a single value was passed back to where the function was called from. The value was passed back using a variable called **Fact**. This variable had the same name as the actual function itself. Procedures can also pass back values but they could also pass no values back at all, or just one value, or many values. Functions only pass back one value using a variable that is the same name as the function itself.

9.2 Variables

A variable is a name given to a particular memory location, a location in RAM. By using variables, a programmer can refer to it to store and retrieve data, even though they may not even know what data is in that location! For example, a programmer can use the instruction:

PRINT Result

The variable name is 'Result'. 'Result' actually refers to a RAM location and in that location is a piece of data. Even though the programmer doesn't know what data is being held there, they can still print it out, by using the instruction just given. High-level languages make a lot of use of variables. It means that the programmer doesn't have to concern itself with actual memory addresses, only the names of RAM addresses! By using meaningful names, the programmer can write programs that can easily be followed and understood.

Every variable in a program must be 'declared'. That means that the programmer must state in the program what kind of data type that particular variable will hold. Then, when the translator comes to translate the program, they will know how much RAM to reserve for that variable - different data types require different amounts of RAM to hold the actual data item.

Variables can be declared in one of two places. They can be declared at the very beginning of a program or they can be declared within a particular function or procedure. If a variable is declared at the beginning of a program then that variable is available to all the functions and procedures that need to use it throughout the entire program. If it is declared in a particular function or procedure then it is available only in that module. This may seem confusing so before we discuss the **scope** of a variable, study the following example!

9.2.1 An example of an outline program

```
Program Wages
Var {declare global variables here}        // declare the global variables here, at the beginning of the program.

Procedure CalculatePay
Var {Declare local variables here}         // declare the local variables for the procedure CalculatePay here.
Begin

End;

Procedure CalculateBonus
Var {Declare local variables here}         // declare the local variables for the procedure CalculateBonus here.
Begin

End;

Procedure CalculateDeductions
Var {Declare local variables here}         // declare the local variables for the procedure CalculateDeductions
here.
Begin

End;

Begin {This is the main program}
   Procedure CalculatePay;
   Procedure CalculateBonus;
   Procedure CalculateDeductions;
End.
```

We have said that variables can be declared in one of two places.

1) At the beginning of the program.
2) Within in a function or procedure.

9.2.2 The scope of a variable

Some variables need to be available in more than one procedure. For example, in the above wages program example, you may need a variable called HoursWorked in the two procedures CalculatePay and CalculateBonus. Both procedures may need to retrieve this information. Because HoursWorked is needed in more than one procedure, it should be declared in the main program section. When you declare a variable in this position so that it is available to any procedure or function in the program then it is known as a **global variable**.

Some variables are only required in a particular procedure or function. The most obvious example of a variable needed within a particular module of code but not anywhere else is a counter. For example, in the CalculateDeductions procedure, you may have the following iteration:

```
FOR COUNT = 1 to 10 DO
BEGIN
        ----------
        ----------
        ----------
END.
```

COUNT is an example of a variable that should be declared only within the procedure CalculateDeductions. Variables declared within a module are known as **local variables**. Unlike global variables, the data they hold are not available to other modules. Whether a variable is available locally or globally is sometimes referred to as the '**scope**' of the variable.

9.2.3 Why use local variables?

Because one of the aims of modular programming is to produce self-contained modules that can be tested on their own, (and contribute towards a library of modules) it is a good idea to use local variables wherever possible. They help to ensure that modules are indeed standalone because the variables cannot have an effect on other modules in the program. This is one of the main weaknesses of procedural programming languages. One module can be unintentionally affected by what happens in other modules because of the way global variables are used.

9.3 Passing parameters

If you write a program as a set of procedures and functions, how can you pass data to them so they can work on it? This is done using 'parameter passing'. The passing of parameters encourages good program design. Since there is now a method that allows independent modules to communicate with each other, we are in a position to write as many independent modules as we like! That means that we could have lots of different people writing modules of code and then simply connect the modules together. It also allows us to start building libraries of modules.

Consider the procedure CalculatePay given in the example earlier. This procedure will calculate the pay for an individual based upon the number of hours they work. If you want to work out the pay for someone, you must call the procedure, telling it what number of hours you want it to work with. To be able to do this you need to

1) Set up the procedure to accept the number of hours.
2) Call the procedure with the hours someone has worked.

You might write the procedure something like this:

```
Procedure CalculatePay (Hours:integer)
Rate := 5
Begin
        Pay := Rate * Hours
End
```

'Hours:integer' is known as the **formal parameter**. Formal parameters define what data the procedure needs to receive so that the procedure can actually work. Formal parameters don't tell the procedure what actual data to use. They only define what data the procedure must get so that it can work! The first line in this procedure,

```
Procedure CalculatePay (Hours:integer)
```

is saying that when the procedure CalculatePay is called, it needs to be given an integer value. That integer value will then be assigned to the variable name Hours.

For example, if you wanted to calculate the pay for someone who has worked 10 hours, then you would need to call the procedure with the value 10, like this:

CalculatePay (10)

This calls the procedure CalculatePay and passes to it the **actual parameter** to use - in this case 10. It is also commonly known as the **argument**. The variable Hours would then be assigned the value 10.

You should see now that a programmer could write a standalone procedure called, for example, CalculateTax. They could set up the procedure with two formal parameters, perhaps called AmountPaid and TaxRate. If this procedure was put into a library then anyone who needs to write a program that does a tax calculation can simply use this procedure - they don't have to write a completely new one from scratch. They simply put the procedure into their program and call it with the necessary arguments.

Of course, we have only passed parameters **into** a procedure in these examples. We may also need to pass parameters **out of** the procedure as well. There would be little point in having a procedure that can accept values and calculate somebody's pay, if we then couldn't pass that value out to the main program so a different module of code can use it!

In the previous pay example, we saw that the first line of the procedure was:

Procedure CalculatePay (Hours:integer)

If we wanted to pass a value **out** of this procedure, perhaps called Pay, then we would need to modify this line:

Procedure CalculatePay (Hours:integer; var Pay:real)

When the CalculatePay procedure has finished, the main program can use the value that it finds in the variable called Pay. Another procedure could refer to this variable if it needs to use the value held in it. Because other procedures can refer to it, this type of parameter is known as a **variable parameter**. When the CalculatePay procedure calculated somebody's pay, it passed the pay back using a variable. We can say that it passed the number back using a **reference to a variable**.

9.4 An introduction to the role of the stack when a procedure is called

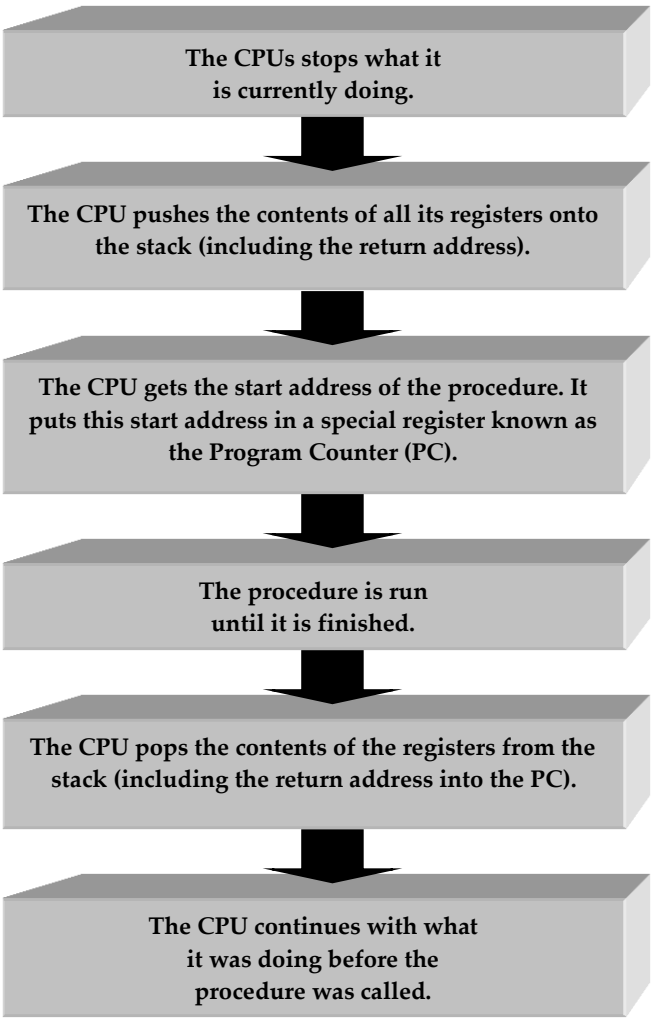
We have already said that the CPU can only work on one program at a time. It may look like you are using a number of programs concurrently but that is because the CPU is switching between them so fast that you hardly notice!

When you call a procedure, the CPU has to stop what it is doing. It then needs to save its position exactly at the moment in time when the procedure was called. In fact, it needs to save the contents of its own **registers**. Then it can jump to the procedure and run that, using its registers as it wants (because it has already saved the contents of them). When the CPU has finished running the procedure, it can return to what it was doing, simply be retrieving the contents of the registers that it had previously saved.

When the procedure was called, the registers were saved in a special area of RAM known as the **stack**. When we put values onto the stack, we talk about **pushing** values onto the stack. When we get values off of the stack, we talk about **popping** values. Every time a function or procedure is called, the contents of all of the registers in the CPU are pushed into the special area in RAM called the stack. If a function calls another function, then exactly the same thing happens before the jump to the function occurs: the contents of all the registers are pushed onto the stack.

The stack will hold a copy of what is in the registers every time a function is called. The stack is a fixed size, however. If too many calls are made to functions, the stack can get used up. This will usually result in an error, your program crashing and an error message about a 'stack overflow' occurring.

We can now summarise what happens when a procedure is called.



The roll of the stack when a procedure or function is called.

Q1. What is a function?
Q2. What is a parameter?
Q3. What is meant by 'calling a function'?
Q4. How do you call a function?
Q5. What is the traditional difference between a procedure and function?
Q6. What is meant by the 'scope' of a variable?
Q7. What is the difference between a global variable and a local variable?
Q8. What is the difference between an argument and a formal parameter?
Q9. What is the stack used for when talking about function calls?
Q10. What is meant by 'pushing values' and 'popping values' when referring to the stack?

Chapter 10 - An introduction to recursion

10.1 Introduction

Recursion is the term used to describe when a sub-program calls itself! After every call to itself a test is done. This checks to see if the sub-program should call itself again. This continues until the test result is such that the recursion (and therefore the calls to itself) ends. Recursion can require a little thought to fully understand the mechanism! Let's look at some examples. The first example is the classic factorial example using functions.

The 'factorial' of a number is arrived at by multiplying every integer from the number down to 1. So, for example,

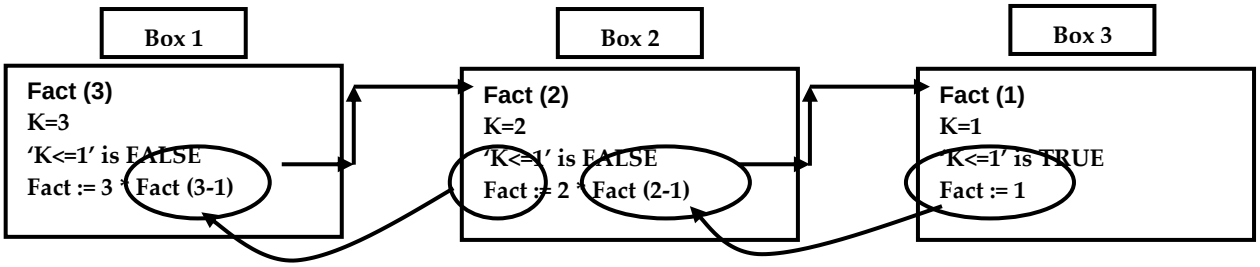
Factorial(6) = 6 x 5 x 4 x 3 x 2 x 1 = 720
Factorial(4) = 4 x 3 x 2 x 1 = 24

We can define a function that will work out the factorial of a number, K.

```
Function Fact(K)
BEGIN
IF K <=1 then
        Fact := 1
ELSE
        Fact := K x Fact(K-1)
END
```

10.2 A worked example

To see how this works, look at the following diagram and then read the description underneath. We want to find out the factorial of 3 so we call the function using the instruction Fact(3). Note that we have drawn a box around each call to the function. This helps us to remember the values of the variables. If you are working in a box then you use any variables' values in that box!



An example to illustrate how recursion works.

1) The function is called using Fact(3).
2) K is set to 3 in this box.
3) The test 'K <= 1' is FALSE. Therefore the 'ELSE' part of the selection construct is done.
4) A variable called Fact in box 1 is set to K x Fact(K-1), or 3 x Fact(3-1).
5) The Fact(3-1) part is a call to the function Fact using the parameter 3-1, or 2!
6) We now jump to a new call to the function using Fact(2).
7) K is set to 2 in this box.
8) The test 'K <= 1' is FALSE. Therefore the 'ELSE' part of the selection construct is done.
9) A variable called Fact in box 2 is set to K x Fact(K-1), or 2 x Fact(2-1).
10) The Fact(2-1) part is a call to the function Fact using the parameter 2-1, or 1!
11) We now jump to a new call to the function using Fact(1).
12) K is set to 1 in this box.
13) The test 'K <= 1' is TRUE. The IF part is executed and the variable Fact is set to 1.
14) The call to box 3 ends and control is passed back to where the call was made from.

15) Don't forget with functions, a value held in a variable whose name is the same as the function, is also passed back! So Fact:=1 is passed back and can replace the call that was made from box 2. In box 2, Fact is now set to 2 x 1 = 2.
16) But box 2 has now finished. The value of Fact is passed back to replace where the call was made. So Fact:=2 is passed back to box 1.
17) In box 1, Fact is set to 3 x 2 = 6. The answer to Fact(3) is therefore 6.

10.3 Another example

Study the next, totally meaningless procedure! If you can follow how this works, you have a good understanding of recursion that can only get better with experience!

The following procedure called Prog has been written. It accepts 3 values, an Integer and two strings. This recursion example doesn't use a function. It uses a procedure. In this case, values aren't passed back when a call has ended, although there are lots of outputs sent to the output screen.

```
Procedure Prog (n:Integer, S:String, T:String)
IF n=2 THEN
        OUTPUT S "I really love programming", T
ELSE
        Prog (n-2, T, S)
        OUTPUT "Hello", T
        Prog (n-2, S, T)
ENDIF
```

As before, get into the habit of drawing a box around each call and identify the value of each of the variables within each box! It is also a good idea to show an output screen, so that you can display everything in the right order

Q1. Define 'the factorial of a number'.
Q2. What is factorial 5?
Q3. What is factorial 7?
Q4. How many values does a function traditionally return?
Q5. Can functions in Python return more than one value?
Q6. Define 'recursion'.
Q7. What is the stack used for with respect to calling a function?
Q8. Is the stack in RAM or ROM?
Q9. What might happen to the stack if you recursively call a function too many times?
Q10. Use the function in 10.3 above to draw a block diagram of the calls and returns for:

Prog(6, Hello, Coops)

Clearly show whatever outputs occur.