CAMBRIDGE
UNIVERSITY PRESS

Computer Science

for Cambridge International AS & A Level

COURSEBOOK

Sylvia Langfield & Dave Duddell

Contents

Part 1: Theory fundamentals

Chapter 1 Information representation

Chapter 2 Communication and networking technologies

Chapter 3 Hardware

Chapter 4 Logic gates and logic circuits

Chapter 5 Processor fundamentals

Chapter 6 Assembly language programming

Chapter 7 Monitoring and control systems

Chapter 8 System software

Chapter 9 Security, privacy and data integrity

Chapter 10 Ethics and ownership

Chapter 11 Databases

Part 2: Fundamental problem-solving and programming skills

Chapter 12 Algorithm design and problem-solving

Chapter 13 Data types and structures

Chapter 14 Programming and data representation

Chapter 15 Software development

Part 3: Advanced theory

Chapter 16 Data representation

Chapter 17 Communication and Internet technologies

Chapter 18 Hardware and virtual machines

Chapter 19 Logic circuits and Boolean algebra

Chapter 20 System software

Chapter 21 Security

Chapter 22 Artificial Intelligence (AI)

Part 4: Further problem-solving and programming skills

Chapter 23 Algorithms

Chapter 24 Recursion

Chapter 25 Programming paradigms

Chapter 26 File processing and exception handling

Chapter 27 Object-oriented programming (OOP)

Chapter 28 Low-level programming

Chapter 29 Declarative programming

Acknowledgements

Introduction

This full-colour, illustrated textbook has been written by experienced authors specifically for the Cambridge International AS & A Level Computer Science syllabus (9618) for examination from 2021. It is based on the first edition by the same authors for the previous Cambridge International AS & A Level Computer Science syllabus (9608). There are substantial changes, the most important being the inclusion of the topic of Artificial Intelligence (See Chapter 22) and the replacement of the Pascal programming language by the Java programming language.

The presentation of the chapters in this book reflects the content of the syllabus:

- The book is divided into four parts, each of which is closely matched to the corresponding part of the syllabus.
- Each chapter defines a set of learning objectives which closely match the learning objectives set out in the syllabus.
- The chapters in Parts 1 and 3 have been written with emphasis on the promotion of knowledge and understanding. The chapters in Parts 2 and 4 have been written with an emphasis on problem solving and programming.

The key concepts for Cambridge International AS & A Level Computer Science are:

Computational thinking

Computational thinking is a set of skills such as abstraction, decomposition and algorithmic thinking. Chapter 12 (Algorithm design and problem-solving), Chapter 15 (Software development) and Chapter 23 (Algorithms) concentrate on this key concept.

Programming paradigms

A programming paradigm is a way of thinking about or approaching problems. Most of the programming in this book follows the imperative (procedural) paradigm. Chapter 25 (Programming paradigms) gives an overview of other paradigms, while Chapter 6 (Assembly language programming), Chapter 28 (Low-level programming), Chapter 27 (Object Oriented Programming) and Chapter 29 (Declarative programming) give an insight into these paradigms.

Communication

Communication in this context ranges from the internal transfer of data within a computer system to the transfer of data across the internet. See Chapter 2 (Communication and networking technologies) and Chapter 17 (Communication and internet technologies).

Computer architecture and hardware

Computer architecture is the design of the internal operation of a computer system. Computer systems consist of hardware (internal components and peripherals) and software that makes the hardware functional. See Chapter 3 (Hardware), Chapter 4 (Logic gates and logic circuits), Chapter 8 (System software), Chapter 18 (Hardware and virtual machines), Chapter 19 (Logic circuits and Boolean algebra) and Chapter 20 (System software).

Data representation and structures

An understanding of binary numbers and how they can be interpreted in different ways is covered in Chapter 1 (Information representation) and Chapter 16 (Data representation). Chapter 11 covers databases. Chapter 13 (Data types and structures) and Chapter 14 (Programming and data representation) show how data can be organised for efficient use and storage.

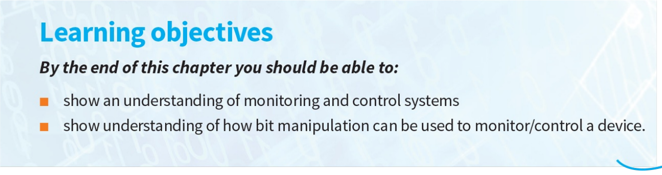
The chapters in Parts 1 and 3 have a narrative which involve a number of interdependent topics. We would encourage learners to read the whole chapter first before going back to revisit the individual sections.

The chapters in Parts 2 and 4 contain many more tasks. We would encourage learners to approach these chapters step-by-step. Whenever a task is presented, this should be carried out before progressing further.

In particular, Chapter 12 (Algorithm design and problem-solving) may be worked through in parallel with Chapter 14 (Programming and data representation). For example, Task 14.03 is based on Worked Example 12.03. After studying this worked example, learners may wish to cover the first part of Chapter 14 and write the program for Task 14.03. This will give the learner the opportunity to test their understanding of an algorithm by implementing it in their chosen programming language. Then further study of Chapter 12 is recommended before attempting further tasks in Chapter 14.

How to use this book

This book contains a number of features to help you in your study.

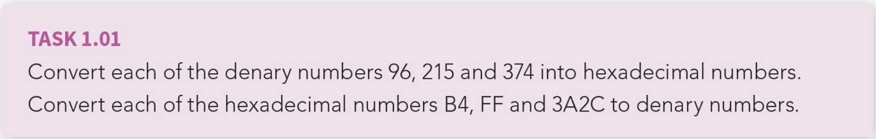


Learning objectives

By the end of this chapter you should be able to:

- show an understanding of monitoring and control systems
- show understanding of how bit manipulation can be used to monitor/control a device.

Learning objectives – each chapter begins with a short list of the learning objectives and concepts that are explained in it.



TASK 1.01

Convert each of the denary numbers 96, 215 and 374 into hexadecimal numbers.

Convert each of the hexadecimal numbers B4, FF and 3A2C to denary numbers.

Task – exercises for you to test your skills.

Question – questions for you to test your knowledge and understanding.

Question 1.01

Does a computer ever use hexadecimal numbers?

Discussion Point:

What is the two's complement of the binary value 1000? Are you surprised by this?

Discussion Point – discussion points intended for class discussion.

Reflection Point – opportunities for you to check your understanding of the topic that has just been covered.

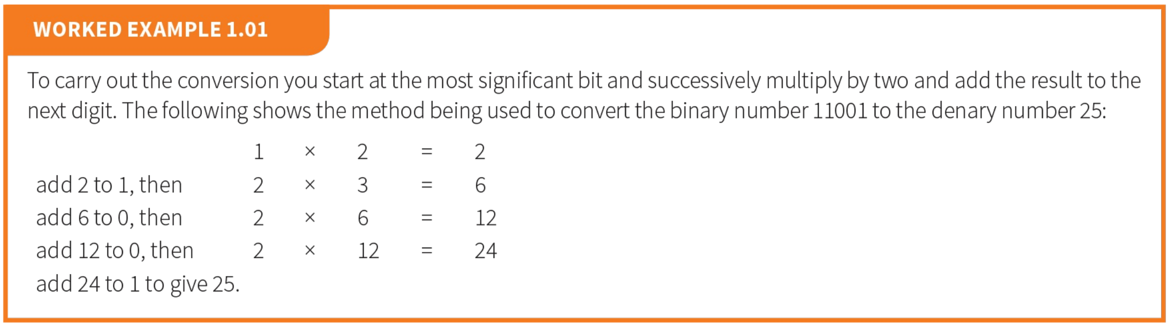
Reflection Point:

Can you recall the different possibilities for what one byte might be coded to represent?

Extension Question – extended questions for consideration of more advanced aspects or topics beyond the immediate scope of the Cambridge International AS & A Level syllabus.

Extension Question 1.01

Graphic files can be stored in a number of formats. For example, JPEG, GIF, PNG and TIFF are just a few of the possibilities. What compression techniques, if any, do these use?



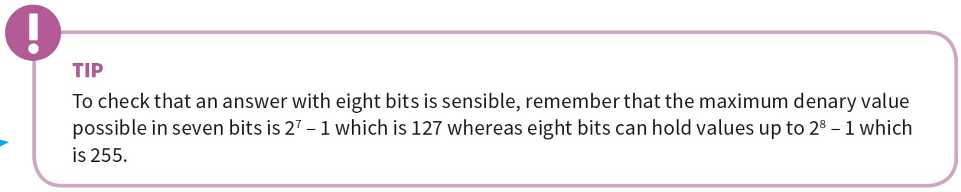
WORKED EXAMPLE 1.01

To carry out the conversion you start at the most significant bit and successively multiply by two and add the result to the next digit. The following shows the method being used to convert the binary number 11001 to the denary number 25:

	1	×	2	=	2
add 2 to 1, then	2	×	3	=	6
add 6 to 0, then	2	×	6	=	12
add 12 to 0, then	2	×	12	=	24
add 24 to 1 to give 25.					

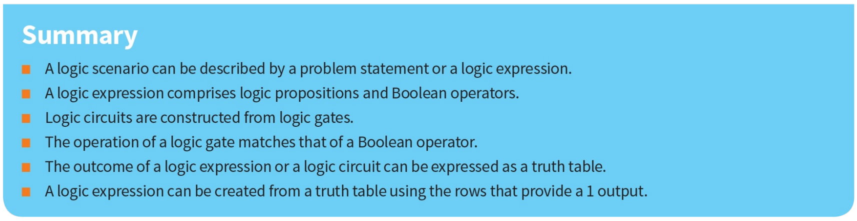
Worked Example – step-by-step examples of solving problems or implementing specific techniques.

Tip – quick notes to highlight key facts and important points.



TIP

To check that an answer with eight bits is sensible, remember that the maximum denary value possible in seven bits is $2^7 – 1$ which is 127 whereas eight bits can hold values up to $2^8 – 1$ which is 255.

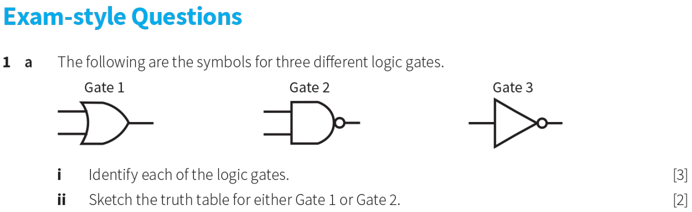


Summary

- A logic scenario can be described by a problem statement or a logic expression.
- A logic expression comprises logic propositions and Boolean operators.
- Logic circuits are constructed from logic gates.
- The operation of a logic gate matches that of a Boolean operator.
- The outcome of a logic expression or a logic circuit can be expressed as a truth table.
- A logic expression can be created from a truth table using the rows that provide a 1 output.

Summary – these appear at the end of each chapter to help you review what you have learned

Exam-style Questions – these aim to test your skills, knowledge and understanding using exam-style questioning.



Exam-style Questions

1 a The following are the symbols for three different logic gates.

Gate 1
Gate 2
Gate 3

i Identify each of the logic gates. [3]

ii Sketch the truth table for either Gate 1 or Gate 2. [2]

Part 1

Theory fundamentals

Chapter 1:
Information representation

Learning objectives

By the end of this chapter you should be able to:

- show understanding of binary magnitudes and the difference between binary prefixes and decimal prefixes
- show understanding of the basis of different number systems
- perform binary addition and subtraction
- describe practical applications where Binary Coded Decimal (BCD) and Hexadecimal are used
- show understanding of and be able to represent character data in its internal binary form, depending on the character set used
- show understanding of how data for a bitmapped image are encoded
- perform calculations to estimate the file size for a bitmap image
- show understanding of the effects of changing elements of a bitmap image on the image quality and file size
- show understanding of how data for a vector graphic are encoded
- justify the use of a bitmap image or a vector graphic for a given task
- show understanding of how sound is represented and encoded
- show understanding of the impact of changing the sampling rate and resolution
- show understanding of the need for and examples of the use of compression
- show understanding of lossy and lossless compression and justify the use of a method in a given situation
- show understanding of how a text file, bitmap image, vector graphic and sound file can be compressed.

1.01 Number systems

Denary numbers

As a child we first encounter the numbers that we use in everyday life when we are learning to count. Specifically, we learn to count using 1, 2, 3, 4, 5, 6, 7, 8, 9, 10. This gives us ten different symbols to represent each individual digit. This is therefore a base-10 number system. Numbers in this system are called **denary numbers** or, more traditionally, decimal numbers.

When a number is written down the value that it represents is defined by the place values of the digits in the number. This can be illustrated by considering the denary number 346 which is interpreted as shown in Table 1.01.

Place value	$10^2 = 100$	$10^1 = 10$	$10^0 = 1$
Digit	3	4	6
Product of digit and place value	300	40	6

Table 1.01 Use of place values in the representation of a denary number

You can see that starting from the right-hand end of the number (which holds the least significant digit), the place value increases by the power of the base number.

Binary numbers

The binary number system is base-2. Each binary digit is written with either of the symbols 0 and 1. A binary digit is referred to as a **bit**.

As with a denary number, the value of a binary number is defined by place values. For example, see Table 1.02 for the binary number 101110.

Place value	$2^5 = 32$	$2^4 = 16$	$2^3 = 8$	$2^2 = 4$	$2^1 = 2$	$2^0 = 1$
Digit	1	0	1	1	1	0
Product of digit and place value	32		8	4	2	0

Table 1.02 Use of place values in the representation of a binary number

By adding up the values in the bottom row you can see that the binary number 101110 has a value which is equivalent to the denary number 46.

You must be able to use the binary number system in order to understand computer systems. This is because inside computer systems there is no attempt made to represent ten different digits individually. Instead, all computer technology is engineered with components that represent or recognise only two states: ‘on’ and ‘off’. To match this, all software used by the hardware uses binary codes which consist of bits. The binary code may represent a binary number but this does not have to be the case.

Binary codes are most often based on the use of one or more groups of eight bits. A group of eight bits is called a **byte**.

Hexadecimal numbers

These are base-16 numbers where each hexadecimal digit is represented by one of the following symbols: 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F. The symbols A through to F represent the denary values 10 through to 15. The value of a number is defined by place values. For example, see Table 1.03 for the hexadecimal number 2A6.

Place value	$16^2 = 256$	$16^1 = 16$	$16^0 = 1$
Digit	2	A	6

Product of digit and place value	512	160	6

Table 1.03 Use of place values in the representation of a hexadecimal number

Adding up the values in the bottom row shows that the equivalent denary number is 678.

In order to explain why hexadecimal numbers are used we need first to define the **nibble** as a group of four bits.

A nibble can be represented by one hexadecimal digit. This means that each byte of binary code can be written as two hexadecimal digits. Two examples are shown in Table 1.04 together with their denary equivalent.

Binary	Hexadecimal	Denary
00001010	0A	10
11111111	FF	255

Table 1.04 Examples of a byte represented by two hexadecimal digits

Note here that if you were converting the binary number 1010 to a hexadecimal number as an exercise on a piece of paper you would not bother with including leading zeros. However, a binary code must not include blanks; all positions in the byte must have either a 0 or a 1. This is followed through in the hexadecimal representation.

One example when you will see hexadecimal representations of bytes is when an error has occurred during the execution of a program. A memory dump could be provided which has a hexadecimal representation of the content of some chosen part of the memory. Another use is when the bytes contain binary numbers in the charts that define character codes. This is discussed later in this chapter.

In the character code charts and in other online sources you may see references to octal numbers which are base-8. You can ignore these.

Converting between binary and denary numbers

One method for converting a binary number to a denary number is to add up the place values for every digit that has a value 1. This was illustrated in Table 1.02.

An alternative method is shown in Worked Example 1.01.

WORKED EXAMPLE 1.01

To carry out the conversion you start at the most significant bit and successively multiply by two and add the result to the next digit. The following shows the method being used to convert the binary number 11001 to the denary number 25:

	1	×	2	=	2
add 2 to 1, then	2	×	3	=	6
add 6 to 0, then	2	×	6	=	12
add 12 to 0, then	2	×	12	=	24

add 24 to 1 to give 25.

To convert a denary number to binary begin by identifying the largest power of 2 that has a value less than the denary number. You can then write down the binary representation of this power of 2 value. This will be a 1 followed by the appropriate number of zeros.

Now subtract the power of two value from the denary number. Then identify the largest power of 2 value that is less than the remainder from the subtraction. You can now replace a zero in the binary representation with a 1 for this new power of 2 position.

Repeat this process until you have accounted for the full denary number.

For example, for the denary number 78 the largest power of two value less than this is 64 so you can start by writing down 1000000. The remainder after subtracting 64 from 78 is 14. The largest power of two value less than this is 8 so the replacement of a zero by 1 gives 1001000. Repeating the process finds values of 4 then 2 so the final answer is 1001110.

An alternative approach is shown in Worked Example 1.02.

WORKED EXAMPLE 1.02

A useful way to convert a denary value to its binary equivalent is the procedure of successive division by two with the remainder written down at each stage. The converted number is then given as the set of remainders in reverse order.

This can be illustrated by the conversion of denary 246 to binary:

246	÷	2	→	123	with remainder 0
123	÷	2	→	61	with remainder 1
61	÷	2	→	30	with remainder 1
30	÷	2	→	15	with remainder 0
15	÷	2	→	7	with remainder 1
7	÷	2	→	3	with remainder 1
3	÷	2	→	1	with remainder 1
1	÷	2	→	0	with remainder 1

Thus, the binary equivalent of denary 246 is 11110110.

TIP

To check that an answer with eight bits is sensible, remember that the maximum denary value possible in seven bits is $2^7 - 1$ which is 127 whereas eight bits can hold values up to $2^8 - 1$ which is 255.

Conversions for hexadecimal numbers

It is possible to convert a hexadecimal number to denary by using the method shown in Table 1.03. However, if there are more than a few digits, the numbers involved in the conversion become very large. Instead, the sensible approach is to first convert the hexadecimal number to a binary number which can then be converted to denary.

To convert a hexadecimal number to binary, each digit is treated separately and converted into a 4-bit binary equivalent, remembering that F converts to 1111, E converts to 1110 and so on.

To convert a binary number to hexadecimal you start with the four least significant bits and convert them to one hexadecimal digit. You then proceed upwards towards the most significant bit, successively taking groupings of four bits and converting each grouping to the corresponding hexadecimal digit.

TASK 1.01

Convert each of the denary numbers 96, 215 and 374 into hexadecimal numbers.

Convert each of the hexadecimal numbers B4, FF and 3A2C to denary numbers.

Question 1.01

Does a computer ever use hexadecimal numbers?

1.02 Numbers and quantities

There are several different types of numbers within the denary system. Examples of these are provided in Table 1.05.

Type of number	Examples	Comments
Integer	3 or 47	A whole number used for counting
Signed integer	−3 or 47	The positive number has an implied + sign
Fraction	2/3 or 52/17	Rarely used in computer science
A number with a whole number part and a fractional number part	−37.85 or 2.83	The positive number has an implied + sign
A number expressed in exponential notation	-3.6×10^8 or 4.2×10^{-9}	The value can be positive or negative and the exponent can be positive or negative

Table 1.05 Different ways to express a value using the denary number system

We will focus on how large values are represented. If we have a quantity that includes units of measurement, it can be written in three different ways. For example, a distance could be written in any one of these three ways:

- 23 567 m
- 23.567×10^3 m
- 23.567 km

The second example has used an exponential notation to define the magnitude of the value. The third example has added a prefix to the unit to define this magnitude. We read this as 23.567 kilometres.

The ‘kilo’ is an example of a **decimal prefix**. There are four decimal prefixes commonly used for large numbers. These are shown in Table 1.06.

Decimal prefix name	Symbol used	Factor applied to the value
kilo	k	10^3
mega	M	10^6
giga	G	10^9
tera	T	10^{12}

Table 1.06 The decimal prefixes

Unfortunately, for a long time the computing world used these prefix names but with a slightly different definition. The value for 2^{10} is 1024. Because this is close to 1000, computer scientists decided that they could use the kilo prefix to represent 1024. So, for example, if a computer system had the following values quoted for the processor speed and the size of the memory and of the hard disk:

Processor speed	1.6 GHz
Size of RAM	8 GB
Size of hard disk	400 GB

The prefix G would represent 10^9 for the processor speed but would almost certainly represent 1024 × 1024 × 1024 for the other two values.

This unsatisfactory situation has now been resolved by the definition of a new set of names which can be used to define a **binary prefix**. A selection of these is shown in Table 1.07.

Binary prefix name	Symbol used	Factor applied to the value
kibi	Ki	2^{10}
mebi	Mi	2^{20}
gibi	Gi	2^{30}
tebi	Ti	2^{40}

Table 1.07 Some examples of binary prefixes

When a number or a quantity is presented for a person to read it is best presented with either one denary digit or two denary digits before the decimal point. If a calculation has been carried out, the initial result found may not match this requirement. A conversion of the presented value will be needed by choosing a sensible magnitude factor. For example, consider the following two answers calculated for the size of a file:

a 34 560 bytes

Here, a conversion to kibibytes would be sensible using the calculation:

$344560\text{B}=345601024\text{KiB}=33.75\ \text{KiB}$

b 3 456 000 bytes

Here, a conversion to mebibytes would be sensible using the calculation:

$3456000\text{B}=(34560001024)1024\text{MiB}=3.296\ \text{MiB}$

If a calculation is to be performed with values quoted with different magnitude factors there must first be conversions to ensure all values have the same magnitude factor. For example, if you needed to know how many files of size 2.4 MiB could be stored on a 4 GiB memory stick there should be a conversion of the GiB value to the corresponding MiB value.

The calculation would be:

$(4\times1024)\text{MiB}2.4\text{MiB}=1076$

1.03 Internal coding of numbers

The discussion in this chapter relates only to the coding of integer values. The coding of non-integer numeric values (real numbers) is considered in Chapter 16 (Section 16.03).

Coding for integers

Computers need to store integer values for a number of purposes. Sometimes only a simple integer is stored, with the understanding that it is a positive number. This is stored simply as a binary number. The only decision to be made is how many bytes should be used. If the choice is to use two bytes (16 bits) then the range of values that can be represented is 0 to (2^{16} – 1) which is 0 to 65 535.

However, in many cases we need to identify whether the number is positive or negative, so we use a signed integer. A signed integer can just have the binary code for the value with an extra bit to define the sign. This is referred to as 'sign and magnitude representation'. For this the convention is to use a 0 to represent + and a 1 to represent –. A few examples of this are shown in Table 1.08.

However, there are a number of disadvantages to using this format, so signed integers are usually in **two's complement** form. Here we need two definitions.

The **one's complement** of a binary number is defined as the binary number obtained if each binary digit is individually subtracted from 1. This means that each 0 is switched to 1 and each 1 switched to 0. The two's complement is defined as the binary number obtained if 1 is added to the one's complement number.

If you need to convert a binary number to its two's complement form, you can use the method indicated by the definition but there is a quicker method. For this you start at the least significant bit and move left ignoring any zeros up to the first 1, which you also ignore. Finally you change any remaining bits from 0 to 1 or from 1 to 0.

For example, expressing the number 10100100 in two's complement form leaves the right-hand 100 unchanged, then the remaining 10100 changes to 01011, so the result is 01011100.

To represent a positive denary integer value as the equivalent two's complement binary form, the process is as follows.

- Use one of methods from Section 1.01 to convert the denary value to a binary value.
- Add a 0 in front of this binary value.

To represent a negative denary integer value as the equivalent two's complement binary form the process is as follows.

- Disregard the sign and use one of methods from Section 1.01 to convert the denary value to a binary value.
- Add a 0 in front of this binary value.
- Convert this binary value to its two's complement form.

A few simple examples of two's complement representations are shown in Table 1.08.

To convert a two's complement binary number representing a positive value into a denary value, the leading zero is ignored and one of the methods in Section 1.01 is applied to convert the remaining binary.

There are two alternative methods for converting a two's complement binary number representing a negative number into a denary value. These are illustrated in Worked Example 1.03.

WORKED EXAMPLE 1.03

Methods for converting a negative number expressed in two's complement form to the corresponding denary number

Consider the two's complement binary number 10110001.

Method 1. Convert to the corresponding positive binary number then convert to denary before adding the minus sign

- Converting 10110001 to two's complement leaves unchanged the 1 in the least significant bit position then changes all of the remaining bits to produce 01001111.
- You ignore the leading zero and apply one of the methods from Section 1.01 to convert the remaining binary to denary which gives 79.
- You add the minus sign to give −79.

Method 2. Sum the individual place values but treat the most significant bit as a negative value

You follow the approach illustrated in Table 1.02 to convert the original binary number 10110001 as follows:

Place value	$-2^7 = -128$	$2^6 = 64$	$2^5 = 32$	$2^4 = 16$	$2^3 = 8$	$2^2 = 4$	$2^1 = 2$	$2^0 = 1$
Digit	1	0	1	1	0	0	0	1
Product	−128	0	32	16	0	0	0	1

You now add the values in the bottom row to get −79.

Some points to note about two's complement representation are as follows.

- There is only one representation of zero.
- Starting from the lowest negative value, each successive higher value is obtained by adding 1 to the binary code. In particular, when all digits are 1 the next step is to roll over to an all-zero code. This is the same as any digital display would do when each digit has reached its maximum value.
- Just adding a leading zero to an unsigned binary value converts it to the two's complement representation of the corresponding positive number
- You use a two's complement conversion to change the sign of a number from positive to negative or from negative to positive. We say that the two's complement values are self-complementary.
- You can add any number of leading zeros to a representation of a positive value without changing the value.
- You can add any number of leading ones to a representation of a negative value without changing the value.

Signed denary number to be represented	**Sign and magnitude representation**	**Two's complement representation**
7	0111	0111
1	0001	0001
0	0000	0000
–0	1000	Not represented
–1	1001	1111
–7	1111	1001
–8	Not represented	1000

Table 1.08 Representations of signed integers

TIP

If you are converting a negative denary number into two’s complement you begin by converting the denary value to a binary value. Then you must not forget to add a leading zero before taking the two's complement to convert the positive value to a negative value.

TASK 1.02

Take the two’s complement of the binary code for –7 and show that you get the code for +7.

TASK 1.03

Convert the two’s complement number 1011 to the denary equivalent. Then do the same for 111011 and convince yourself that you get the same value.

Discussion Point:

What is the two’s complement of the binary value 1000? Are you surprised by this?

Binary arithmetic

Before considering the addition of binary numbers it is useful to recall how we add two denary numbers. Two rules apply. The first rule is that the process is carried out starting with addition of the two least significant digits and then working right to left. The second rule is that if an addition produces a value greater than 9 there is a carry of 1. For example in the addition of 48 to 54, the first step is adding 8 to 4 to get 2 with a carry of 1. Then 5 is added to 4 plus the carried 1 to give 0 with carry 1. The rules produce 102 for the sum which is the correct answer.

For binary addition, starting at the least significant position still applies. The rules for the addition of binary digits are:

- 0 + 0 = 0
- 0 + 1 = 1
- 1 + 1 = 0 with a carry of 1
- 1 + 1 + 0 = 0 with a carry of 1
- 1 + 1 + 1 = 1 with a carry of 1

The last two rules are used when a carried 1 is included in the addition of two digits.

As an example, the addition of the binary equivalent of denary 14 to the binary equivalent of denary 11 can be examined.

$$\begin{array}{cccccc} & & 1 & 0 & 1 & 1 \\ + & & 1 & 1 & 1 & 0 \\ \hline & 1 & 1 & 0 & 0 & 1 \end{array}$$

The steps followed from right to left are:

- 1 + 0 = 1 with no carry
- 1 + 1 = 0 with carry 1
- 0 + 1 + carried 1 = 0 with carry 1
- 1 + 1 + carried 1 = 1 with carry 1

The rules have correctly produced the 5-bit answer which is the binary equivalent of 25. In a paper exercise like this these rules for addition will always produce the correct answer.

Again for subtraction we can first consider how this is done for denary numbers. As for addition the process starts with the least significant digits and proceeds right to left. The special feature of subtraction is the “borrowing” of a 1 from the next position when a subtracting digit is larger than the digit it is being subtracted from.

For example in subtracting 48 from 64 the first step is to note that 8 is larger than 4. Therefore 1 has to be borrowed as 10. The 10 added to 4 gives 14 and 8 subtracted from this gives 6. When we proceed to the next digit subtraction we first have to reduce the 6 to 5 because of the borrow. So we have subtraction of 4 from 5 leaving 1. The answer for the subtraction is 16.

For binary subtraction, starting at the least significant position still applies. The rules for the subtraction of binary digits are:

- 0 – 0 = 0
- 0 – 1 = 1 after a borrow
- 1 – 0 = 1
- 1 – 1 = 0

As an example, the subtraction of the binary equivalent of denary 11 from the binary equivalent of denary 14 can be examined.

$$\begin{array}{ccccc} & 1 & 1 & 1 & 0 \\ - & 1 & 0 & 1 & 1 \\ \hline & 0 & 0 & 1 & 1 \end{array}$$

The steps followed from right to left are:

- 1 is larger than 0 so 1 is borrowed giving subtraction of 1 from 10 leaving 1
- Because of the borrow the 1 is reduced to 0 so that 1 is to be subtracted from 0. This requires a further borrow giving subtraction of 1 from 10 leaving 1
- Because of the borrow the 1 is reduced to 0 leaving subtraction of 0 from 0
- 1 – 1 gives 0

The answer is the binary value for denary 3.

When binary addition is carried out by a computer using internally stored numbers there is a major difference. This arises from the fact that the storage unit will always have a defined number of bits. For example, in the above addition, if binary values were limited to being stored in a nibble the result of the addition would be incorrectly stored as 1001. This is an example of an **overflow**. The value produced is too large to be stored.

When the values in a computer system are stored in two’s complement form this problem has a characteristic behaviour.

In the following addition where +63 is added to +63 there is no problem; the answer is correctly obtained as +126:

$$\begin{array}{ccccccccc} & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 \\ + & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 \\ \hline & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \end{array}$$

However, if the binary for +96 is added to +96 the result is as follows:

$$\begin{array}{cccccccccc} & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ + & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ \hline & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}$$

The overflow means that the answer has a leading 1, which causes a computer system to interpret the answer as a negative number.

A similar problem can occur when two negative values are added. For example the addition of −96 to the same value results in the following:

$$\begin{array}{cccccccccc} & & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ + & & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ \hline & (1) & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}$$

This time there has been a carry when the most significant bits were added and the result obtained is a positive number.

Clearly we need the processor to detect overflow and output an error message. There is a discussion of how a processor can detect overflow in Chapter 6 (Section 6.07).

One of the advantages of using two's complement representations is that it simplifies the process of subtracting one number from another. The number being subtracted is converted to its two's complement form, which is then added to the other number.

TASK 1.04

Using a byte to represent each value, carry out the subtraction of denary 35 from denary 67 using binary arithmetic with two's complement representations.

Binary coded decimal (BCD)

One exception to grouping bits in bytes to represent integers is the **binary coded decimal (BCD)** scheme. This is useful in applications that require single denary digits to be stored or transmitted. The BCD code uses a nibble to represent a denary digit. We consider the simple scheme where the digits are coded as the binary values from 0000 to 1001. The remaining codes 1010 to 1111 do not have any meaning.

If a denary number with more than one digit is to be converted to BCD there has to be a group of four bits for each denary digit. There are, however, two options for BCD; the first is to store one BCD code in one byte, leaving four bits unused. The other option is **packed BCD** where two 4-bit codes are stored in one byte. Thus, for example, the denary digits 8503 could be represented by either of the codes shown in Figure 1.01.

One BCD digit per byte	00001000	00000101	00000000	00000011
Two BCD digits per byte	10000101	00000011		

Figure 1.01 Alternative BCD representations of the denary digits 8503

There are a number of applications where BCD can be used. The obvious type of application is where denary digits are to be displayed, for instance on the screen of a calculator or in a digital time display. A somewhat unexpected application is for the representation of currency values. When a currency value is written in a format such as $300.25 it is as a fixed-point decimal number (ignoring the dollar sign). It might be expected that such values would be stored as real numbers but this cannot be done accurately (this type of problem is discussed in more detail in Chapter 16 (Section 16.03)). One solution to the problem is to store each denary digit as a BCD code.

Let's consider how BCD arithmetic might be performed by a computer if fixed-point decimal values for currency were stored as BCD values. Here is an example of addition.

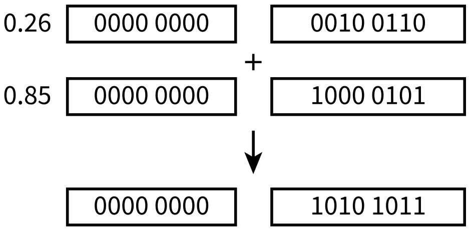


Figure 1.02 Incorrect addition using BCD coding

We will assume a two-byte packed BCD representation. The first byte represents two denary digits for the whole part of the number and the second byte represents two denary digits for the fractional part. If the two values are $0.26 and $0.85 then the result of the addition should be $1.11. This would involve a carry from the first decimal place to the whole number 1. However, applying simple binary addition of the BCD codes would produce the result shown in Figure 1.02.

The additions for the fractional parts have produced values corresponding to the denary values 10 and 11 but a BCD value is supposed to be a single digit. The error has resulted in no carry to the whole number column.

We need the processor to recognise that an impossible value has been produced and apply a method to correct this. The solution is to add 0110 whenever the problem is detected. This is illustrated in Figure 1.03.

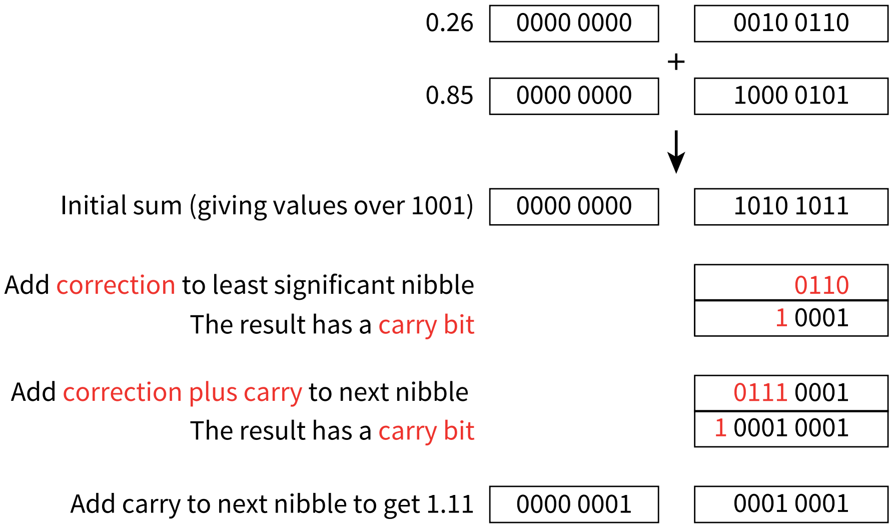

0.26
0000 0000
0010 0110
+
0.85
0000 0000
1000 0101
Initial sum (giving values over 1001)
0000 0000
1010 1011
Add correction to least significant nibble
0110
The result has a carry bit
1 0001
Add correction plus carry to next nibble
0111 0001
The result has a carry bit
1 0001 0001
Add carry to next nibble to get 1.11
0000 0001
0001 0001

Figure 1.03 Use of the correction value to perform BCD addition

The steps shown in Figure 1.03 are as follows.

- Starting with the least significant nibble, adding 0110 to 0101 gives 1011 which is recognised as being incorrect.
- The 0110 correction value is added to produce 10001.
- The 0001 is stored and the leading 1 is carried to the next nibble.
- In the first decimal position adding 0100 to 1000 then adding the carry bit 1 gives 1011 which is recognised as being incorrect.
- The 0110 correction is added to produce 10001.
- The 0001 is stored and the leading 1 is carried to the next nibble.

In this example the two whole number nibbles have zero values so adding these has no effect.

1.04 Internal coding of text

To store text in a computer, we need a coding scheme that provides a unique binary code for each distinct individual component item of the text. Such a code is referred to as a character code. There have been many different examples of character coding schemes throughout the history of computing.

ASCII code

The scheme which has been used for the longest time is the ASCII (American Standard Code for Information Interchange) coding scheme. The 7-bit version of the code (often referred to as US ASCII) was standardised many years ago by ANSI (American National Standards Institute). The codes are always presented in a table. Table 1.09 shows an edited version of a typical table. The first column contains the binary code which would be stored in one byte, with the most significant bit set to zero and the remaining bits representing the character code. The second column shows the hexadecimal equivalent.

Binary code	Hexadecimal equivalent	Character	Description
00000000	00	NUL	Null character
00000001	01	SOH	Start of heading
00100000	20		Space
00100001	23	#	Number
00110000	30	0	Zero
00110001	31	1	One
01000001	41	A	Uppercase A
01000010	42	B	Uppercase B
01100001	61	a	Lowercase a
01100010	62	b	Lowercase b

Table 1.09 Some examples of ASCII codes stored in one byte with the remaining, most significant bit set to zero

A full table would show the 2^7 (128) different codes available for a 7-bit code.

TIP

Do not try to remember any of the individual codes

You need to remember these key facts about the ASCII coding scheme.

- A limited number of the codes represent non-printing or control characters; these were introduced to assist in data transmission or for data handling at a computer terminal.
- The majority of the codes are for characters that would be found in an English text and which are available on a standard keyboard.
- These include upper- and lower-case letters, punctuation marks, denary digits and arithmetic symbols.
- The codes for numbers and for letters are in sequence so that, for example, if 1 is added to the code for seven, the code for eight is produced.
- The codes for the upper-case letters differ from the codes for the corresponding lower-case letters only in the value of bit 5, which allows a simple conversion from upper to lower case or the reverse. (Don’t forget that the least significant bit is bit 0.)

Note that this coding for numbers is exclusively for use in the context of stored, displayed or printed text. All of the other coding schemes for numbers are for internal use in a computer system and would not be used in a text.

Although a standard version of ASCII has been created, different versions of 7-bit ASCII are tailored to different software or different countries. Mostly, the coding for the printable characters has remained unchanged. A notable exception was the use in some countries of the code 00100001 to represent a currency symbol rather than #. However, because most of the control characters became of limited use, there were versions of ASCII that used these codes to produce small graphic icons. For example, the code 00000001 would show ☺.

Extended ASCII is a code that uses all eight bits in a byte. The most used standardised version is often referred to as ISO Latin-1. The name Latin-1 reflects the fact that many of the new character definitions are for accented or otherwise modified alphabetic characters found in European languages, for example Ç or ü. As with the 7-bit code, there are many variations of the standard code.

Question 1.02

Many years ago, a byte was defined as six bits. If a character was to be represented by one byte, which characters would you expect to be representable and which ones would you expect to be unavailable?

Unicode

Although ASCII codes are widely used, they do not cover all the characters needed for some uses. For this reason, new coding schemes have been developed and continue to be developed further. The discussion here describes one of the Unicode schemes. It should be noted that Unicode codes have been developed in tandem with the Universal Character Set (UCS) scheme, standardised as ISO/IEC 10646.

The aim of Unicode is to be able to represent any possible text in code form. In particular, this includes all languages in the world. The most popular version of Unicode which is discussed here is named UTF-8. The inclusion of 8 in the name indicates that this version of the standard includes codes defined by one byte in addition to codes using two, three and four bytes.

Figure 1.04 shows the structure of the codes. The 1 byte code reproduces 7-bit ASCII. Because the byte has the most significant bit set to 0 there can be no confusion with any byte which is part of a multiple byte code. Note that for the two-byte, three-byte and four-byte representations all continuing bytes have the two most significant bits set to 10. Whenever a byte has the most significant bits set to 11 there will be at least one continuation byte following.

0???????			
110?????	10??????		
1110????	10??????	10??????	
11110???	10??????	10??????	10??????

Figure 1.04 Byte formats for Unicode UTF-8

The number of codes available is determined by the number of bits that are not pre-defined by the format. For example, there are eleven bits free to identify codes in the 2-byte format. This allows 2^{11} = 2048 different codes.

Unicode has its own special terminology and symbolism. A character code is referred to as a ‘code point’. In any documentation a code point is identified by U+ followed by a 4-digit hexadecimal number. The code points U+0000 to U+00FF define characters which are a duplicate of those in the standard Latin-1 scheme. The binary codes corresponding to U+0000 to U+007F use one byte only and range from 00000000 through to 01111111. Then the binary codes for U+0080 to U+00FF require two bytes and range from 11000000 for the first byte followed by 10000000 for the second byte through to 11000001 followed by 10111111.

1.05 Images

Images can be stored in a computer system for the eventual purpose of displaying the image on a screen or for presenting it on paper, usually as a component of a document. Such an image can be created by using an appropriate graphics package. Alternatively, when an image already exists independently of the computer system, the image can be captured by using photography or by scanning.

Vector graphics

In an image that is created by a drawing package or a computer-aided design (CAD) package each component is an individual **drawing object**. The image is then stored, usually as a **vector graphic** file.

We do not need to consider how an image of this type would be created. We do need to consider how the data is stored after the image has been created. A vector graphic file contains a **drawing list**. The list contains a command for each object included in the image. Each command has a list of attributes, each attribute defines a **property** of the object. The properties include the basic geometric data such as, for a circle, the position of the centre and its radius. In addition, properties are defined such as the thickness and style of a line, the colour of a line and the colour that fills the shape. An example of what could be created as a vector graphic file is shown in Figure 1.05.

TASK 1.05

Construct a partial drawing list for the graphic shown in Figure 1.05. You can take measurements from the image and use the bottom left corner of the box as the origin of a coordinate system. You can invent your own format for the drawing list.

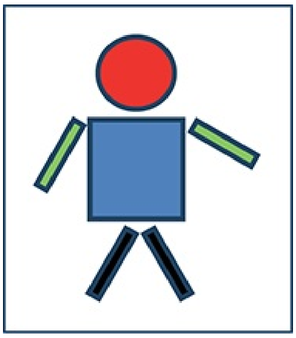


Figure 1.05 A simple example of a vector graphic image

The most important property of a vector graphic image is that the dimensions of the objects are not defined explicitly but instead are defined relative to an imaginary drawing canvas. In other words, the image is scalable. Whenever the image is to be displayed the file is read, the appropriate calculations are made and the objects are drawn to a suitable scale. If the user then requests that the image is redrawn at a larger scale the file is read again and another set of calculations are made before the image is displayed. This avoids image distortion, such as the image appearing squashed or stretched.

Note that a vector graphic file can only be displayed directly on a graph plotter, which is an expensive specialised piece of hardware. For the image to appear correctly on other types of display, the vector graphic file often has to be converted to a bitmap.

Bitmaps

Most images do not consist of geometrically defined shapes, so a vector graphic representation is inappropriate. Instead, generally an image is stored as a bitmap. Typical uses are when capturing an existing image by scanning or perhaps by taking a screen-shot. Alternatively, an image can be created by using a simple graphics package.

The fundamental concept underlying the creation of a bitmap file is that the **picture element (pixel)** is the smallest identifiable component of a bitmap image. The image is stored as a two-dimensional matrix of pixels. The pixel itself is a very simple construct; it has a position in the matrix and it has a colour. It does not matter whether each pixel is a small rectangle, a small circle or a dot.

The scheme used to represent the colour has to be defined. The simplest option is to use one bit to represent the colour, so that the pixel is either black or white. Storage of the colour in four bits allows simple greyscale colouring. At least eight bits per pixel are necessary to provide a sufficient range of colours to provide a reasonably realistic representation of any image. The number of bits per pixel is sometimes referred to as the **colour depth**.

An alternative definition is the **bit depth**. Although these terms are sometimes used interchangeably, bit depth is best defined as the number of bits used to store each of the red, green and blue primary colours in the RGB colour scheme.

A colour depth of 8 bits per pixel provides 256 different colours. A bit depth of 8 bits per primary colour provides 256 × 256 × 256 = 16 777 216 different colours. The eye cannot distinguish this number of different colours. However, this many are needed if an image contains areas of gradually changing colour such as in a picture of the sky. If a lower bit depth is used the image will show bands of colour.

We also need to decide which resolution to use for the image, which can be represented as the product of the number of pixels per row times the number of rows. When considering resolution it is important to distinguish between an **image resolution**, as defined in a bitmap file, and a **screen resolution** for a particular monitor screen that might be used to display the image. Both of these have to be considered if a screen display is being designed.

A bitmap file does not define the physical size of a pixel or of the whole image. When the image is scaled the number of pixels in it does not change. If a well-designed image is presented on a suitable screen the human eye cannot distinguish the individual pixels. However, if the image is magnified too far the individual pixels will be seen. This is illustrated in Figure 1.06 which shows an original small image, a magnified version of this small image and a larger image created with a more sensible, higher resolution.

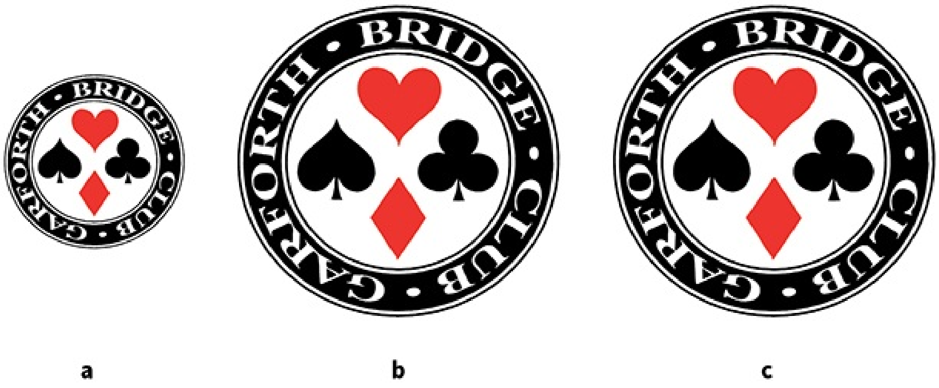


Figure 1.06 (a) a bitmap logo; (b) an over-magnified version of the image; (c) a sensible larger version

File size is always an issue with an image file. A large file occupies more memory space and takes longer to display or to be transmitted across a network. Usually, a vector graphic file uses considerably less memory space than a corresponding bitmap file.

You can calculate the size of a bitmap graphic knowing the resolution and the colour depth. As an example, consider that a bitmap graphic is needed to fill a laptop screen where the resolution is 1366 by 768. If we want colour depth of 24 then the number of bits we need is:

1366 × 768 × 24 = 25 178 112 bits

The result of this calculation shows the number of bits, but a size is always quoted as a number of bytes or multiples of bytes. For our bitmap graphic:

25 178 112 bits = 25 178 112 ÷ 8 = 3 147 264 bytes

= 3 147 264 ÷ 1024 = 3073.5 kibibytes (3073.5 KiB)

= 3073.5 ÷ 1024 = approximately 3 MiB

Note that this calculation has assumed that the colour depth specifies the total number of bits used to define each pixel. If the information given was that the bit depth was eight, then the calculation would use 8 + 8 + 8 for the number of bits per pixel.

WORKED EXAMPLE 1.04

You have been asked to calculate a value for the minimum size of a bitmap file. The bitmap is to use a bit depth of 8 and the bitmap is to be printed with 72 dpi (dots per inch) and to have dimensions 5 inches by 3 inches.

We use the information provided about the colour depth or the bit depth to give the number of bits per pixel. In this case the bit depth is 8, which means 8 bits for each of the RGB components, so 24 bits are needed for one pixel.

Let's state that 72 dpi means 72 pixels per inch.

So, the number of pixels per row is 5 × 72 = 360

And the number of pixels per column is 3 × 72 = 216

Therefore, the total number of pixels is 360 × 216 = 77 760

The total number of bits is this value multiplied by 24. However, we want the size in bytes not bits, so we multiply by 3 because there are 8 bits in a byte. So, we get:

77 760 × 3 = 233 280 bytes.

We can quote this in kibibytes by dividing by 1024:

233 280 / 1024 = 227.8 KiB

A bitmap file has to store the pixel data that defines the graphic, but the file must also have a **file header** that contains information on how the graphic has been constructed. Because of this, the bitmap file size is larger than the size of the graphic alone. At the very least the header will define the colour depth or bit depth and the resolution.

The following are considerations when justifying the use of either a bit map or a vector graphic for a specific task.

- A vector graphic is chosen if a diagram is needed to be constructed for part of an architectural, engineering or manufacturing design.
- If a vector graphic file has been created but there is a need to print a copy using a laser or inkjet printer the file has first to be converted to a bitmap.
- A digital camera automatically produces a bitmap.
- A bitmap file is the choice for insertion of an image into a document, publication or web page.

1.06 Sound

Natural sound consists of variations in pressure which are detected by the human ear. A typical sound contains a large number of individual waves, each with a defined frequency. The result is a wave form in which the amplitude of the sound varies in a continuous but irregular pattern.

If we want to store sound or transmit it electronically the original analogue sound signal has to be converted to a binary code. The measured sound values are input to a sound encoder which has two components. The first is a band-limiting filter. This is needed to remove high-frequency components. A human ear cannot detect these very high frequencies and they could cause problems for the coding if not removed. The other component in the encoder is an analogue-to-digital converter (ADC) which converts the **analogue data** to **digital data**.

Figure 1.07 shows the **sampling** operation of the ADC. The amplitude of the wave (the red line) has to be sampled at regular intervals. The blue vertical lines indicate the sampling times. The amplitude cannot be measured exactly; instead the amplitude is approximated by the closest of the defined amplitudes represented by the horizontal lines. In Figure 1.07, sample values 1 and 4 will be an accurate estimate of the actual amplitude because the wave is touching an amplitude line. In contrast, samples 5 and 6 will not be accurate because the actual amplitude is approximately half way between the two closest defined values.

To code sound, we need to make two decisions. The first is the number of bits we will use to store the amplitude values, which defines the **sampling resolution**. If we use only three bits then eight levels can be defined as shown in Figure 1.07. If too few are used there will be a significant error when the closest amplitude in the scale of values dictated by the sampling resolution is used as the approximation for the real value. In practice, 16 bits provides reasonable accuracy for most digitised sound.

We also need to choose the **sampling rate**, which is the number of samples taken per second. This should be in accordance with Nyquist’s theorem which states that sampling must be done at a frequency at least twice the highest frequency of the sound in the sample.

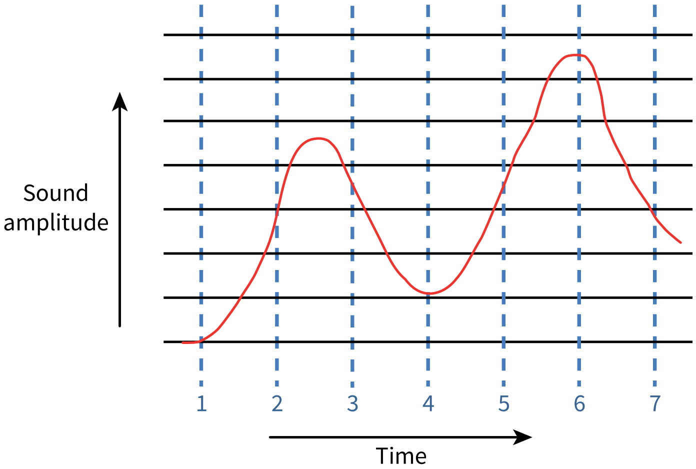


Figure 1.07 ADC sampling

Once again file size can be an issue. An increased sampling rate and an increased sampling resolution will both cause an increase in file size.

1.07 Compression techniques

Larger files require larger storage capacity but more importantly, larger files have lower transmission or download rates. For this reason, compression techniques are often used to reduce file size.

There are two categories of compression. The first is **lossless compression** where the file size is reduced but no information is lost. The process can be reversed to re-create the original file. The second is **lossy compression** where the file size is reduced with some loss of information and the exact original file can never be recovered. In many applications a combination of lossless and lossy methods are used.

We could use the same type of lossless file compression for everything, because all files contain binary codes. A good compression application will recognise patterns in files that it can compress, without any knowledge of what file type the code represents. However, most compression techniques have been developed to work with a particular type of file.

A common lossless compression technique is run-length encoding. This works particularly well with a bitmap file. The idea is that compression converts sequences of the same byte value into a code that defines the byte value and the number of times it is repeated (the count).

For example, the sequence of the same four bytes:

01100110 01100110 01100110 01100110

could be replaced by:

00000100 01100110

which says that there is a run of four of the bytes.

However, this is not the full story because in this simple form it is not obvious which byte represents the number (count) in the sequence. There are a number of methods used to distinguish the count byte from a data byte, but we do not need to go into the details.

If a file contains text, then compression must be lossless because any loss of information would lead to errors in the text. One possible compression method is called Huffman coding. The procedure used to carry out the compression is quite detailed, but the principle is straightforward. Instead of having each character coded in one byte, the text is analysed to find the most often used characters. These are then given shorter codes. The original stream of bytes becomes a bit stream.

A possible set of codes if a text contained only eight different letters is shown in Table 1.10. The important point to note here is the prefix property. None of the codes begins with the sequence of bits representing a shorter code. This means that there can be no ambiguity when the transmitted compressed file has to be converted back to the original text.

Code	Character
10	e
01	t
111	o
110	h
0001	l
0000	p
0011	w
0010	z

Table 1.10 An example of Huffman coding

Huffman coding can also be used for compressing a sound file. This is effective because some values for the amplitude occur far more often than others do.

If a vector graphic file needs to be compressed it is best converted to a Scalable Vector Graphics format. This uses a markup language description of the image which is suitable for lossless compression.

Lossy compression can be used in circumstances where a sound file or an image file can have some of the detailed coding removed or modified. This can happen when it is likely that the human ear or eye will hardly notice any difference. One method for lossy compression of a sound file takes advantage of the fact that the successive sampled values are unlikely to change very much. The file of individual sample amplitudes can be converted to a file of amplitude differences. Compression is achieved by using a lower sample resolution to store the differences. An alternative is to convert the sampled amplitudes that represent time domain data and transform them to a frequency domain representation. The values for frequencies that would be barely audible are then re-coded with fewer bits before the data is transformed back to the original time domain form.

For a bitmap a simple lossy compression technique is to establish a coding scheme with reduced colour depth. Then for each pixel in the original bitmap the code is changed to the one in the new scheme which represents the closest colour.

Extension Question 1.01

Graphic files can be stored in a number of formats. For example, JPEG, GIF, PNG and TIFF are just a few of the possibilities. What compression techniques, if any, do these use?

Reflection Point:

Can you recall the different possibilities for what one byte might be coded to represent?

Summary

- A binary code or a binary number can be documented as a hexadecimal number.
- Internal coding of signed integers is usually based on a two’s complement representation.
- Binary addition can cause overflow.
- BCD is a convenient coding scheme for single denary digits.
- ASCII and Unicode are standardised coding schemes for text characters.
- An image can be stored either in a vector graphic file or in a bitmap file.
- An ADC works by sampling a continuous waveform.
- Lossless compression allows an original file to be recovered by a decoder; lossy compression irretrievably loses some information.

Exam-style Questions

1 A file contains binary coding. The following are two successive bytes in the file: 10010101 and 00110011

a One possibility for the information stored is that the two bytes together represent one unsigned integer binary number.

i Calculate the denary number corresponding to this. [2]

ii Calculate the hexadecimal number corresponding to this. [2]

b Give **one** example of when a hexadecimal representation is used. [1]

c Another possibility for the information stored is that the two bytes individually represent two signed integer binary numbers in two's complement form.

i State which byte represents a negative number and explain the reason for your choice.

ii Calculate the denary number corresponding to each byte. [3]

d Give **two** advantages of representing signed integers in two's complement form rather than using a sign and magnitude representation. [2]

e Give **three** different examples of other options for the types of information that could be represented by two bytes. For each example, state whether a representation requires two bytes each time, just one byte or only part of a byte each time. [3]

2 A designer wishes to include some multimedia components on a web page.

a If the designer has some images stored in files there are two possible formats for the files.

i Describe the approach used if a graphic is stored in a vector graphic file. [2]

ii Describe the approach used if a graphic is stored in a bitmap file. [2]

iii State which format gives better image quality if the image has to be magnified and explain why. [2]

b The designer is concerned about the size of some bitmap files.

i If the resolution is to be 640 × 480 and the colour depth is to be 16, calculate an approximate size for the bitmap file. State the answer using sensible units. [2]

ii Explain why this calculation only gives an approximate file size. [1]

c The designer decides that the bitmap files need compressing.

i Explain how a simple form of lossless compression could be used. [2]

ii Explain **one** possible approach to lossy compression that could be used. [2]

3 An audio encoder is to be used to create a recording of a song. The encoder has two components.

a One of the components is an analogue-to-digital converter (ADC).

i Explain why this is needed. [2]

ii Two important factors associated with the use of an ADC are the sampling rate and the sampling resolution. Explain the two terms. Sketch a diagram if this will help your explanation. [5]

b The other component of an audio encoder has to be used before the ADC is used.

i Identify this component. [1]

ii Explain why it is used. [2]

4 a i Using two's complement, show how the following denary numbers could be stored in an 8-bit register:

124								

–77								

[2]

ii Convert the two numbers in **part (a) (i)** into hexadecimal. [2]

b Binary Coded Decimal (BCD) is another way of representing numbers.

i Write the number 359 in BCD form. [1]

ii Describe a use of BCD number representation. [2]

Cambridge International AS & A level Computer Science 9608 paper 13 Q1 June 2015

5 **a** Sound can be represented digitally in a computer.

Explain the terms sampling resolution and sampling rate. [4]

b The following information refers to a music track being recorded on a CD:

- music is sampled 44 100 times per second
- each sample is 16 bits
- each track requires sampling for left and right speakers.

i Calculate the number of bytes required to store one second of sampled music. Show your working. [2]

ii A particular track is four minutes long.

Describe how you would calculate the number of megabytes required to store this track. [2]

c When storing music tracks in a computer, the MP3 format is often used. This reduces file size by about 90%.

Explain how the music quality is apparently retained. [3]

Cambridge International AS & A level Computer Science 9608 paper 12 Q4 November 2015

Chapter 2:
Communication and networking technologies

Learning objectives

By the end of this chapter you should be able to:

- show understanding of the purpose and benefits of networking devices
- show understanding of the characteristics of a LAN (local area network) and a WAN (wide area network)
- explain the client-server and peer-to-peer models of networked computers
- show understanding of thin-client and thick-client and the differences between them
- show understanding of the bus, star, mesh and hybrid topologies
- show understanding of cloud computing
- show understanding of the differences between and implications of the use of wireless and wired networks
- describe the hardware that is used to support a LAN
- describe the role and function of a router in a network
- show understanding of Ethernet and how collisions are detected and avoided
- show understanding of bit streaming
- show understanding of the differences between the World Wide Web (WWW) and the Internet
- describe the hardware that is used to support the Internet
- explain the use of IP addresses in the transmission of data over the Internet
- explain how a Uniform Resource Locator (URL) is used to locate a resource on the World Wide Web (WWW) and the role of the Domain Name Service (DNS).

2.01 The evolution of the purpose and benefits of networking

Wide area network (WAN)

During the 1970s it would be normal for a large organisation to have a computer. This computer would be a mainframe or minicomputer. The computer could have been running a time-sharing operating system with individual users accessing the computer using a terminal connected to the computer with a cable. Technology was developed that allowed computers in different organisations to be networked using what would now be described as a **wide area network (WAN)**. In a WAN, the networked computers could be thousands of kilometres apart.

The benefits of having the computers connected by a WAN were:

- a ‘job’ could be run on a remote computer that had the required application software
- a data archive that was stored on a remote computer could be accessed
- a message could be transmitted electronically to a user on a remote computer.

Today, a typical WAN is characterised by the following.

- It will be used by an organisation or a company to connect sites or branches.
- It will not be owned by the organisation or company.
- It will be leased from a public switched telephone network company (PSTN).
- A dedicated communication link will be provided by the PSTN.
- The transmission medium will be fibre-optic cable.
- Transmission within the WAN will be from switch to switch.
- A switch will connect the WAN to each site.
- There will not be any end-systems connected directly to the WAN.

Local area network (LAN)

In the 1980s the arrival of the microcomputer or personal computer (PC) changed computing. In an organisation, a user could have their own computer on their desk. Initially this was used as a stand-alone system. However, very soon the decision would be made to connect the PCs a **local area network (LAN)**. It was called a local area network because it typically connected PCs that were in one room or in one building or on one site.

The benefits of connecting PCs in a LAN included the following.

- The expense of installing application software on each individual PC could be saved by installing the software on an application server attached to the LAN instead.
- A file server could be attached to the LAN that allowed users to store larger files and also allowed files to be shared between users.
- Instead of supplying individual printers to be connected to a user’s PC, one or more printers could be attached to a print server that was connected to the LAN; these could be higher quality printers.
- Managers in organisations could use electronic mail to communicate with staff rather than sending round memos on paper.
- The ‘paper-less office’ became a possibility, where files were to be stored in digital form on a file server rather than as paper copies in a filing cabinet.

Today, a typical LAN is characterised by the following.

- It will be used by an organisation or a company within a site or branch.

- It will be owned by the organisation or company.
- It will be one of many individual LANS at one site.
- The transmission medium will be twisted pair cable or WiFi.
- The LAN will contain a device that allows connection to other networks.
- There will be end-systems connected which will be user systems or servers.

Discussion Point:
If a print server was attached to a network, what functionality could it provide?

Internet working

The 1990s can be said to be when the modern era of computing and network use started, with the beginning of widespread use of the Internet. The word Internet is a shortened form of the term ‘internetwork’, which describes a number of networks all connected together. LANs are connected to WANs which are in turn connected to the Internet to allow access to resources world-wide. The other technologies defining the modern era, namely mobile devices and wireless networking, started to become commonly used in the 2000s.

The purpose and benefits of networking have not changed but their scale and scope has increased enormously. In particular, people now have full access to networks from their personal devices.

The client-server model

The **client-server** model (or architecture) was first used in large organisations when they had installed internal networks. Typically, the organisation would have individual LANs connected via an organisation-wide WAN. An individual LAN might have had an application server attached. The organisation was likely to need a powerful central computer. The central computer could be connected to the WAN as a server. It would probably not have individual users connected to it directly. A PC, attached to a LAN, could access the server as a client.

The client-server mode of operation nowadays is different. The client is a web browser connected to the Internet. The server is a web server hosted on the Internet.

The server provides an application and the client uses the application. There are two options for how the client functions.

A **thin-client** is one which:

- chooses an application to run on the server
- sends input data to the server when requested by the application
- receives output from the application.

A **thick-client** is one which:

- chooses an application provided by the server
- possibly carries out some processing before running the application on the server and also after receiving output from the application
- alternatively, possibly downloads the application from the server and runs the application itself.

TIP
In thick-client mode the processing on the client can be controlled by the use of a scripting language. You do not need to know any details of this.

The client-server approach is the choice in the following circumstances.

- The server stores a database which is accessed from the client system.

- The server stores a web application which allows the client system to find or, sometimes, supply information.
- The server stores a web application which allows the client system to carry out an e-commerce or financial transaction.

File sharing

If a user uploads files to a file server then the client-server operation can be used by another user to download these from the server.

An alternative mode of operation for sharing files is peer-to-peer networking. Instead of having one server that many clients access, a peer-to-peer network operates with each peer (networked computer) storing some of the files. Each peer can therefore act as a client and request a file from another peer or it can act as a server when another peer requests the download of a file.

The peer-to-peer model has several advantages compared to client-server file downloading:

- it avoids the possibility of congestion on the network when many clients are simultaneously attempting to download files
- parts of a file can be downloaded separately
- the parts are available from more than one host.

The client-server model has the following advantages.

- It allows an organisation to control the downloading and use of files.
- The files can be better protected from malware attacks because the files are stored on one server which will be regularly scanned using appropriate anti-virus software.

2.02 Network topologies

There are five requirements for a data communications system: a sender, a receiver, a transmission medium, a message and a protocol (see Chapter 17 for details about protocols). A transmission medium can be air (e.g. for WiFi) or cables (e.g. for Ethernet). Data can be sent through the medium in different modes:

- simplex mode where data flow is one-way only
- half duplex where data can flow either way but not simultaneously
- full duplex where simultaneous both-ways data flow is possible.

A 'message' is any type of data, which can be sent as either:

- a broadcast, which is a one-to-all communication (as used traditionally for radio and television)
- a multicast, which is from one source to many destinations
- a unicast, which is a one-to-one communication.

A data communications system may consist of a single isolated network. There are several possibilities for the **topology** of an isolated network. The simplest of these is where two systems are connected by a network link as shown in Figure 2.01. This is an example of a point-to-point connection, which is a dedicated link. Transmission might be simplex or duplex and a message can only be unicast.

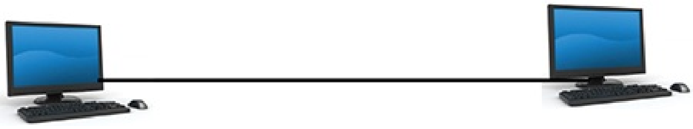


Figure 2.01 A point-to-point network

Early LAN topologies used either a ring or a **bus topology**. We don't need to cover the ring topology as it is not used very often now. A bus topology has only one link but it is shared by a number of end-systems and is therefore described as a multi-point connection. The configuration is shown in Figure 2.02. There is no direct connection between any pair of end-systems. A message must therefore be broadcast even though it might only be intended for one **end-system**. The topology is resilient because a fault in an end-system or in the link to it does not affect the use of the network by the other end-systems.

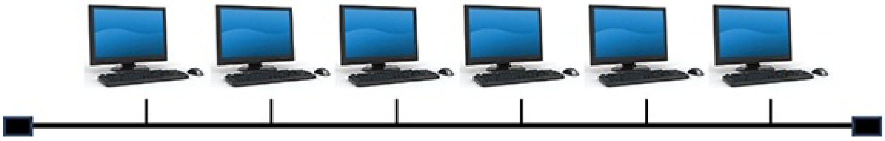


Figure 2.02 A bus network

An example of a fully-connected **mesh topology** is shown in Figure 2.03. In this configuration, each end-system has a point-to-point connection to each of the other end-systems. Transmission is duplex; messages might be unicast, multicast or broadcast.

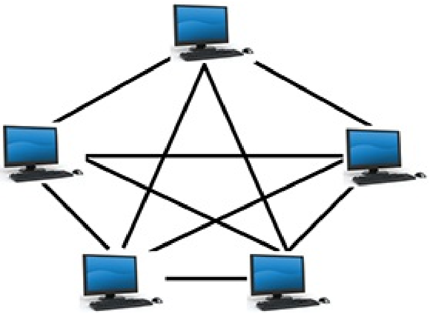


Figure 2.03 A mesh network.

Figure 2.03 shows end-systems connected in a mesh topology but this is unrealistic because of the amount of cabling required. A mesh topology can be used when individual LAN switches are connected in a network. The topology is essential for the connection of routers within the infrastructure of the Internet.

The final possibility is a **star topology** which is shown in Figure 2.04.

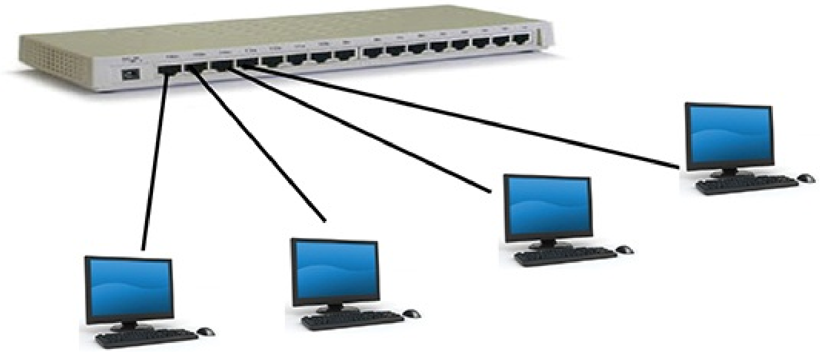


Figure 2.04 A star topology

Figure 2.04 could have been drawn so that it looked like a star but has been drawn to show the physical configuration that is used in a real life installation. In a star topology, each end-system has a point-to-point connection to the central device. Transmission is duplex and messages from the central device might be unicast, multicast or broadcast. As with the bus topology, the failure of an end-system, or its link, leaves the other end-systems unaffected. However, the central device must not fail.

In the bus topology most of the end-systems might be user workstations and the others are servers. However, in the star topology, the end-systems might be user workstations or servers but the central device is different. It is a specialised device with the purpose of connecting other devices in the network. Currently, the star topology is the usual way to configure a network. There are several reasons for this. The most important is that the central device can be used to connect the network to other networks and, in particular, to the Internet.

Discussion Point:

Which network topologies have you used? You might wish to defer this discussion until you have read about network devices later in this chapter.

In a situation where several LANs are connected, they can have different topologies or supporting technologies. This collection of LANs then becomes a **hybrid network**. A special connecting device is needed to ensure that the hybrid network is fully functional. It is often an advantage to be able to connect a new topology LAN to existing LANs where it is not sensible or not possible to use the existing topology for the new LAN. An example is when a wired LAN is already installed but a new wireless LAN is to be connected to it.

2.03 Transmission media

Cable

A network **cable** can be twisted pair, coaxial or fibre-optic. The twisted pair and coaxial cables both use copper for the transmission medium. In discussing suitability for a given application there are a number of factors to consider. One of these is the cost of the cable and connecting devices. Another is the best **bandwidth** that can be achieved. The bandwidth governs the possible data transmission rate. There are then two factors that can cause poor performance: the likelihood of interference affecting transmitted signals and the extent of attenuation (deterioration of the signal) when high frequencies are transmitted. These factors will dictate whether repeaters or amplifiers are needed in transmission lines and how many will be needed. Table 2.01 shows some comparisons of the different cable types.

	Twisted pair	Coaxial	Fibre-optic
Cost	Lowest	Higher	Highest
Bandwidth or data rate	Lowest	Higher	Much higher
Attenuation at high frequency	Affected	Most affected	Least affected
Interference	Worst affected	Less affected	Least affected
Need for repeaters	More often	More often	Less often

Table 2.01 Comparisons between cable types

You need to understand that for each of the three types of cabling there are defined standards for different grades of cable which must be considered when you decide which type of cable to use. Fibre-optic cable performs best but costs more than the other kinds. For a new installation the improved performance of fibre-optic cable is likely to be the factor that governs your choice. However, where copper cable is already installed the cost of replacement by fibre-optic cable may not be justified.

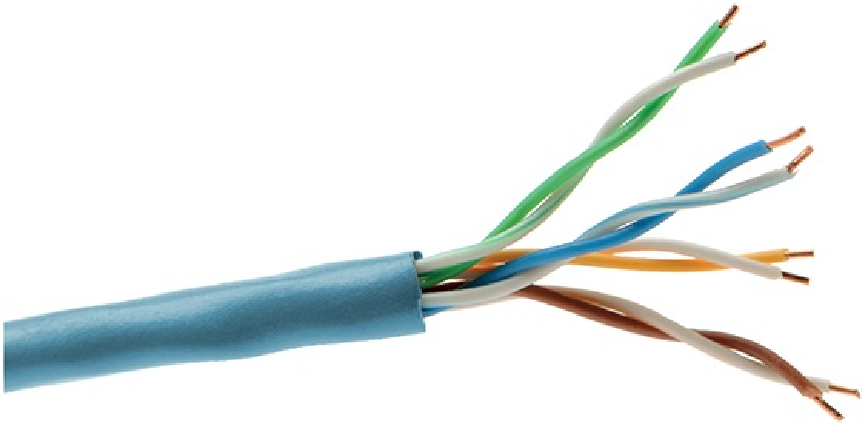


Figure 2.05 One cable with four twisted pairs with differing twist rates to reduce interference

Currently, twisted pair cable is normally used to connect telephone handsets to telephone lines. This type of cable is illustrated in Figure 2.05. It is also the technology of choice for high-speed local area networks.

Question 2.01

Twisted pair cable can be shielded or unshielded. What are the options for this? How does shielding affect the use of the cable?

Coaxial cable is used extensively by cable television companies and in metropolitan area networks. It is not usually used for long-distance telephone cabling. Fibre-optic cable is the technology of choice for long-distance cabling. As shown in Figure 2.06, coaxial cable is not bundled but a fibre-optic cable contains many individual fibres.

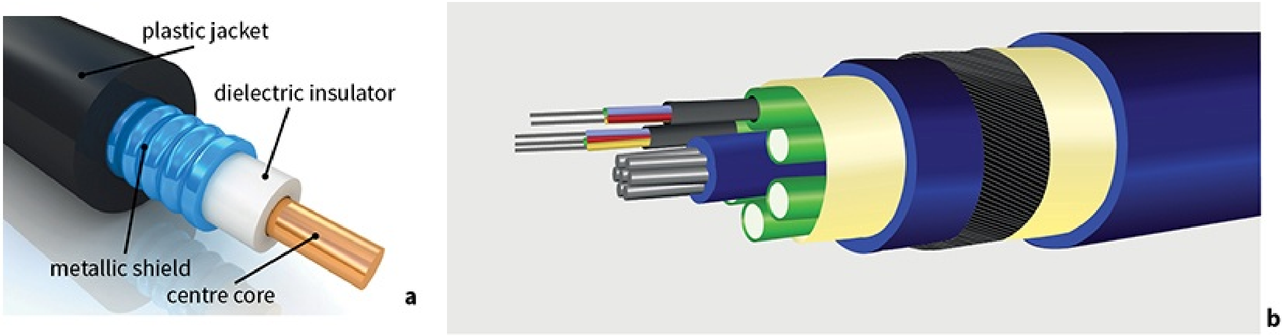


Figure 2.06 (a) Coaxial cable and (b) a bundled fibre-optic cable

Wireless

The alternative to cable is **wireless** transmission. The three options here are radio, microwave or infrared. These are all examples of electromagnetic radiation; the only intrinsic difference between the three types is the frequency of the waves.

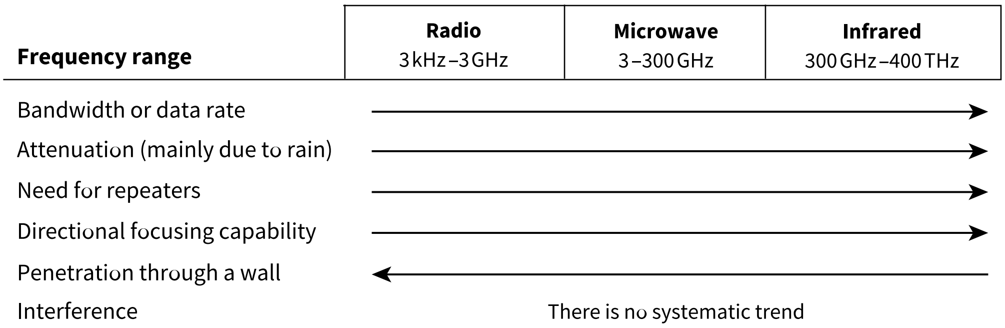


Figure 2.07 Frequency ranges and frequency dependency of factors affecting wireless transmission

When making a choice of which wireless option to use you need to consider all of the same factors that were discussed when comparing different kinds of cable. In addition, the ability of the radiation to transmit through a solid barrier is an important factor. Also, the extent to which the transmission can be focused in a specific direction needs to be considered. Figure 2.07 shows the approximate frequency ranges for the three types of radiation. The factors listed on the left increase in the direction of the arrows. The bandwidth increases through radio and microwave to infrared but the ability of the waves to penetrate solid objects is greatest for radio waves. Interference is not consistently affected by the frequency.

The increased attenuation for infrared transmission, which has the highest frequency, means that it is only suitable for indoor applications. The fact that it will not penetrate through a wall is then of benefit because the transmission cannot escape and cause unwanted interference elsewhere. For most applications, microwave transmission is the best option because it has a better bandwidth compared to that available using radio waves.

Comparing cable and wireless transmission

It is worth noting that cables are often referred to as ‘guided media’ and wireless as ‘unguided media’. This is slightly misleading because only radio wave transmission fits the description of unguided. It is possible with microwaves or infrared to direct a transmission towards a particular receiver (as suggested in Figure 2.07).

There are other points to consider when we compare the relative advantages of transmission through a cable or wireless transmission.

- The use of certain wireless transmission frequencies is regulated by government agencies and so permission has to be obtained before wireless transmission is used.
- Outside these frequencies, no permission is needed to use the air for transmission but cables can only be laid in the ground with the permission of landowners.

- For global communications, the two competing technologies are: transmission through fibre-optic cables laid underground (or on the sea bed) and satellite transmission (discussed later in this section).
- Interference is much more significant for wireless transmission and its extent is dependent on which frequencies are being used for different applications.
- Repeaters are needed less often for wireless transmission.
- Mobile (cell) phones now dominate Internet use and for these, only wireless transmission is possible.
- For home or small office use, wired or wireless transmission is equally efficient; often, not having to install cables favours wireless connections for a small network.

Satellites are components of modern communication systems. Figure 2.08 shows the altitudes (distances above Earth) of three different types of satellite. The Van Allen belts are areas containing high levels of electrically charged particles, which interfere with satellites.

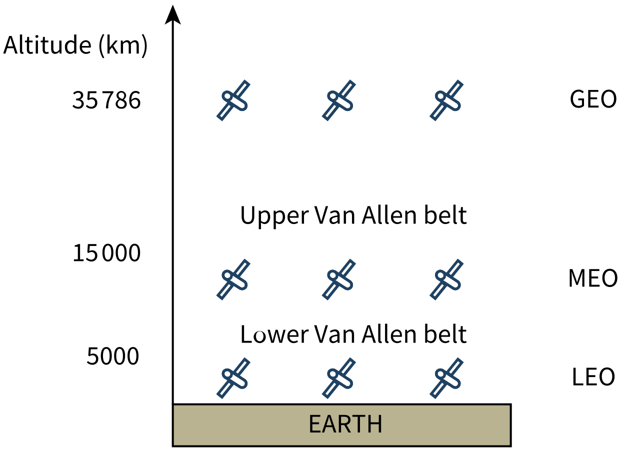


Figure 2.08 Satellite altitudes

The highest altitude satellites are in geostationary Earth orbit (GEO) over the equator and these are used to provide long-distance telephone and computer network communication. ‘Geostationary’ means that the satellite orbits at the same speed as the Earth spins, so from a point on the Earth the satellite always appears to be at the same point in the sky. Only three GEO satellites are needed for full global coverage. Closer to Earth are a group of medium-Earth-orbit (MEO) satellites some of which provide the global positioning system (GPS). Ten MEO satellites are needed for global coverage. Finally, low-Earth-orbit (LEO) satellites work in ‘constellations’ to supplement the mobile phone networks. Fifty LEO satellites are needed for full global coverage but currently there are several hundred LEO satellites in orbit.

A satellite can act as a component in a network and can directly connect with ground-based components. These ground-based components can be much further apart than in a network with no satellites. The disadvantage of satellites is that the greater transmission distance causes transmission delays, which can cause technical problems for the network.

TASK 2.01

Calculate the approximate time taken for a transmission from the surface of the Earth to a medium-Earth-orbit satellite. (Take the speed of light to be 300 000 km per second.)

The use of satellites in networks tends to be for specialised applications such as the Global Positioning System (GPS) or for Internet use in remote locations. At one stage, a lot of Internet communication was expected to make use of satellites, but the development of high-speed fibre-optic cabling at relatively low cost has reduced the need for satellites.

2.04 LAN hardware

Wired LANs

In the early years, coaxial cable was used for LANs. Nowadays, twisted pair cables are probably the most widely used networking connections, and fibre-optic cables are becoming more common. In a bus configuration the bus will consist of a series of sockets linked by cables. The ends of the bus have terminators attached that prevent signals from reflecting back down the bus. Each end-system (which is either a user workstation or a **server**), has a short length of cable with an RJ-45 connector at each end. One end is plugged into a bus socket and the other end is plugged into the LAN port of the end-system.

In a star configuration each end-system has the same type of cable with the same connectors but the cable tends to be much longer because it has to plug into a socket on the central device.

A bus can be extended by linking two bus cables using a **repeater**. A repeater is needed because over long distances, signals become attenuated (reduced in strength), making communication unreliable. A repeater receives an input signal and generates a new full-strength signal. Sometimes a bus network is constructed in what are called segments. Two segments are connected using a **bridge**. The bridge stores the network addresses for the end-systems in the two segments it connects.

The LAN port on an end-system is connected to a **Network Interface Card (NIC)**. The NIC is manufactured with a unique network address that is used to identify the end-system in which it has been installed. The addressing system is discussed in Chapter 17 (Section 17.05). For a star network, the central device might be a hub, a **switch** or a router. The switch is by far the most likely. A switch is a connecting device that can direct a communication to a specific end-system. There is discussion of how it functions in Section 2.05. The router is discussed later in this chapter and also in Chapter 17.

Wireless LANs

WiFi (WLAN in some countries) is a term used to describe wireless Ethernet. Its formal description is IEEE 802.11. This is a wireless LAN standard that uses radio frequency transmission. The central device in a WiFi LAN is a **Wireless Access Point (WAP)**. This can be an end-system in a wired network. The WAP can communicate with an end-system in the WiFi LAN provided that the end-system has a **Wireless Network Interface Card (WNIC)** installed.

2.05 Ethernet

Ethernet is one of the two dominant technologies in the modern networked world. It is primarily focused on LANs. Although Ethernet was first devised in the 1970s independently of any organisation, it was later adopted for standardisation by the Institute of Electrical and Electronics Engineers (IEEE). In particular it was their 802 committee that took responsibility for the development of the protocol. The standard for a wired network is denoted as IEEE 802.3 which is sometimes used as an alternative name for Ethernet. The standard has so far evolved through five generations: standard or traditional, fast, gigabit, 10 gigabit and 100 gigabit. The gigabit part of the name indicates its data transfer speed capability.

Original (or 'legacy') Ethernet was implemented on a LAN configured either as a bus or as a star with a hub as the central device. In either topology, a transmission was broadcast type. Any message would be made available to all of the end-systems without any controlled communication exchange between any pair of end-systems. For each message received an end-system had to check the destination address defined in the message to see if it was the intended recipient.

The use of a shared medium for message transmission has the potential for messages to be corrupted during transmission. If two end-systems were to transmit messages at the same time there would be what is described as a 'collision'. This is when the voltages associated with the transmission interfere with each other causing corruption of the individual messages. The method adopted for dealing with this was CSMA/CD (carrier sense multiple access with collision detection). This relied on the fact that if a message was being transmitted there was a voltage level on the Ethernet cable which could be detected by an end-system.

The transmitter uses the following procedure.

1. Check the voltage on the transmission medium.
2. If this indicates activity, wait a random time before checking again.
3. If no activity is detected, start transmission.
4. Continuously check for a collision.
5. If no collision is detected, continue transmission.
6. If a collision is detected, stop transmission of the message and transmit a jamming signal to warn all end-stations; after a random time, try again.

Although there might be some legacy Ethernet LANs still operating, modern Ethernet is switched. The star configuration has a switch as the central device. The switch controls transmission to specific end-systems. Each end-system is connected to the switch by a full-duplex link, so no collision is possible along that link. Because there might be high levels of activity the switch needs to be able to store an incoming message in a buffer until the cable is free for the transmission to take place. Since collisions are now impossible, CSMA/CD is no longer needed. Some further details concerning Ethernet are provided in Chapter 17 (Section 17.04).

Discussion Point:

Carry out some research about the different versions of Ethernet. Which version is implemented for the systems you use? For how long will it give sufficient performance?

2.06 The Internet infrastructure

To describe the Internet as a WAN pays little attention to its size and complexity. The Internet is the biggest internetwork in existence. Furthermore, it has never been designed as a single ‘whole’; it has just evolved to reach its current form and is still evolving towards whatever future form it will take.

Internet Service Provider (ISP)

One of the consequences of the Internet not having been designed is that there is no agreed definition of its structure. However, there is a hierarchical aspect to the structure (meaning that there are several distinct ‘levels’ within the structure). For example, the initial function of an Internet Service Provider (ISP) was to give Internet access to an individual or company. This function is now performed by what we can call an ‘access ISP’. These access ISPs then connect to what we can call ‘middle tier’ or regional ISPs, which in turn are connected to tier 1 (or ‘backbone’) ISPs. An ISP is a network and connections between ISPs are handled by Internet Exchange Points (IXPs). The tier 1 ISPs are at the top of the hierarchy, along with major Internet content providers.

Discussion Point:

How many ISPs or major Internet content providers are you familiar with?

Router

We can also think of the Internet in terms of the connections that carry the most traffic, which consist of a set of fibre-optic cables laid under the sea and across land, which can be described as a ‘mesh’ structure. This mesh of cables contains many points where the cables connect together, which we call nodes. At every node is a device called the **router**. Routers are found not only in the general ‘mesh’ of the Internet but also within the ISP networks. Each router is connected to several other routers and its function is to choose the best route for a transmission. The details of how a router works are discussed in Chapter 17 (Section 17.05).

Question 2.02

How near are you to an under-the-sea Internet fibre-optic cable?

Public switched telephone network (PSTN)

Communication systems that were not originally designed for computer networking provide significant infrastructure support for the Internet. The longest standing example is what is often referred to as POTS (plain old telephone service) but is more formally described as a PSTN (public switched telephone network). There is some discussion about how PSTNs provide that support in Chapter 17. During the early years of networking the telephone network carried analogue voice data. However, digital data could be transmitted provided that a modem was used to convert the digital data to analogue signals. Another modem was used to reverse the process at the receiving end. Such so-called ‘dial-up’ connections provided modest-speed, shared access when required. However, an organisation could instead pay for a leased line service that provided a dedicated, permanently connected link with guaranteed transmission speed. Typically, organisations made use of leased lines to establish WANs (or possibly MANs (metropolitan area networks)).

More recently, the PSTNs have upgraded their main communication lines to fibre-optic cable employing digital technology. This has allowed them to offer improved leased line services to ISPs but has also given them the opportunity to provide their own ISP services. In this role they provide two types of service. The first is a broadband network connection for traditional network access. The second is WiFi hotspot technology, where an access point as described in Section 2.04 has a connection to a wired network providing Internet access.

Cell phone network

For users of devices with mobile (cell) phone capability there is an alternative method for gaining Internet access. This is provided by mobile phone companies acting as ISPs. The mobile phone, equipped with the appropriate software, communicates with a standard cell tower to access the wireless telephone network, which in turn provides a connection to the Internet.

2.07 Applications that make use of the Internet

The World Wide Web (WWW)

It is common practice to talk about ‘using the web’ or ‘using the Internet’ as though these were just two different ways of saying the same thing. This is not true. The Internet is, as has been described above, an Internetwork. By contrast, the World Wide Web (WWW) is a distributed application which is available on the Internet.

Specifically, the web consists of an enormous collection of websites each having one or more web pages. The special feature of a web page is that it can contain hyperlinks which, when clicked, give direct and essentially immediate access to other web pages.

Cloud computing

Cloud computing is the provision of computing services usually via the Internet. An organisation may choose to establish its own **private cloud**. In this case there are three possible approaches:

- The organisation takes full responsibility for creating and managing the cloud installed on-site and connected to a private network
- The organisation outsources to a third-party the creation and management of an on-site installation connected to a private network
- The organisation outsources the creation and management of an Internet accessible system by a third-party.

The alternative is a **public cloud**. This is created, managed and owned by a third-party cloud service provider.

The services provided by a cloud are familiar ones provided by file servers and application servers. They are accessible via a browser and therefore accessible from any suitable device in any location. A public cloud can be accessed by an individual user or by an organisation. One major difference is the scale of the systems. The provision is established using large mainframe computers or server farms. The services provided can be characterised as being one of:

- infrastructure provision
- platform provision
- software provision

Many of the advantages to a cloud user arise from the fact that the cloud does not have the limitations that the systems already available have. For the infrastructure provision, the advantages include the better performance when running software and the increased storage capacity. For the platform provision, the cloud can offer facilities for software development and testing. For the software provision, the cloud will be able to run applications that require high performance systems. Alternatively, it could be that the costs to a company of buying and installing a software package themselves would be far too high. The other advantage is the familiar one with regard to outsourcing. The cloud user no longer needs technical expertise.

The disadvantages to a cloud user relate to the use of a public cloud. The cloud service provider has complete access to all of the data stored on the cloud. The cloud user cannot be sure that their data is not being shared with third-parties. This is a concern with regard to data privacy. The security of the data stored is an issue; the cloud service provider is being relied on to ensure data cannot be lost.

Bit streaming

Streaming media make use of the Internet for leisure activities like listening to music or watching a video. But what is a ‘bit stream’? In general, before data is transmitted it is stored in bytes which can be transmitted one after the other as a ‘byte stream’. Because of the file sizes involved, streamed media is always compressed to a sequence of bits - a ‘bit stream’. Generic compression techniques mentioned in Chapter 1 (Section 1.07) can convert the byte stream to a bit stream with fewer bits overall. For the decoding process at the receiver end to work properly, the data must be transferred as a bit stream.

For one category of streaming media, the source is a website that has the media already stored. One option in this case is for the user to download a file then listen to it or watch it at some future convenient time. However, when the user does not wish to wait that long there is the streaming option. This option is described as viewing or listening **on demand**. In this case the delivery of the media and the playing of the media are two separate processes. The incoming media data are received into a buffer created on the user’s computer. The user’s machine has media player software that takes the media data from the buffer and plays it.

The other category of streaming media is **real-time** or live transmission. In this case the content is being generated as it is being delivered such as when viewing a sporting event. At the receiver end the technology is the same as before. The major problem is at the delivery end because a very large number of users may be watching simultaneously. The way this is managed now is to transmit the media initially to a large number of content provider servers which then transmit onwards to individual users.

A crucial point with media streaming is whether the technology has sufficient power to provide a satisfactory user experience. When the media is created it is the intention that the media is to be delivered to the user at precisely the same speed as used for its creation; a song that lasted four minutes when sung for the recording would sound very peculiar if, when it is received by a user, it lasts six minutes. The process of delivering the content is determined by the **bit rate**. For example, a relatively poor-quality video can be delivered at a bit rate of 300 kbps but a reasonably good-quality audio file only requires delivery at 128 kbps. Figure 2.09 shows a simple schematic diagram of the components involved in the streaming.

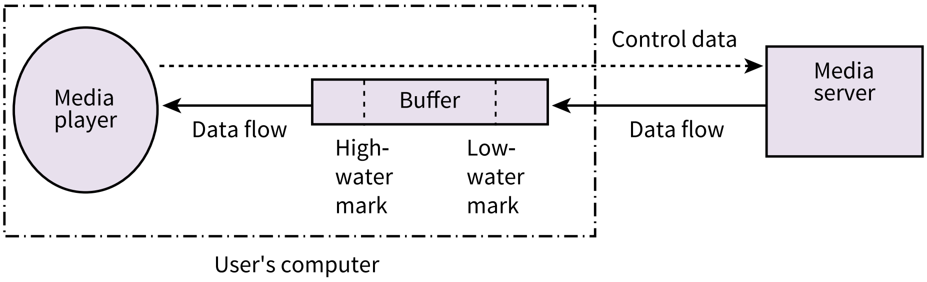


Figure 2.09 Schematic diagram of bit streaming

The buffer must deliver the data to the user, at the correct bit rate for the media being used. Data which is sent into the buffer should be sent at a higher rate to allow for unexpected delays. The media player continuously monitors how full the buffer is and controls the bit rate in relation to the defined high- and low-water marks. It is essential to have a buffer size that is sufficiently large for it never to get filled.

The rate of transmission to the buffer is limited by the bandwidth of the network connection. For a connection via a PSTN, a broadband link is essential. For good-quality movie presentation the broadband requirement is about 2.5 Mbps. Because this will not be available for all users it is often the practice that an individual video is made available at different levels of compression. The most highly compressed version will be the poorest quality but the bit rate may be sufficiently low for a reasonable presentation with a relatively low bandwidth Internet connection.

TASK 2.02

Consider a bit-streaming scenario for a video where the following values apply:

- the buffer size is 1 MiB
- the low-water mark is set at 100 KiB
- the high-water mark is set at 900 KiB

- the incoming data rate is 1 Mbps
- the video display rate is 300 Kbps.

Assume that the video is playing and that the buffer content has dropped to the low-water mark. The media player sets the controls for data input to begin again.

Calculate the amount of data that will be input to the buffer in two seconds and the amount of data that will be removed from the buffer in the same time period.

Repeat the calculation for 4, 6, 8, 10 and 12 seconds.

From this data, estimate when the buffer will have filled up to the high-water mark.

Assuming that the incoming transmission is halted at this time, calculate how long it will be before the buffer content has again fallen to the low-water mark level.

2.08 IP addressing

The Internet requires technical protocols to function. A protocol suite called TCP/IP is used as a standard (see Chapter 17). One aspect of this is IP addressing, which is used to define from where and to where data is being transmitted.

IPv4 addressing

Currently the Internet uses Internet Protocol version 4 (IPv4) addressing. IPv4 was devised in the late 1970s, before the invention of the PC and the mobile phone. IPv4 provides for a large but limited number of addresses for devices, which is no longer enough to cover all the devices expected to use the Internet in future.

The IPv4 addressing scheme is based on 32 bits (four bytes) being used to define an **IPv4 address**. It is worth putting this into context. The 32 bits allow 2^{32} different addresses. For big numbers like this it is worth remembering that 2^{10} is approximately 1000 in denary so the 32 bits provide for approximately four billion addresses. The population of the world is about seven billion and it is estimated that approaching half of the world's population has Internet access. From this we can see that if there was a need to supply one IP address per Internet user the scheme would just about be adequate. However, things are not that simple.

The original addressing scheme was designed on the basis of a hierarchical address with a group of bits defining a network (a netID) and another group of bits defining a host on that network (a hostID). The aim was to assign a unique, universally recognised address for each device on the Internet. The separation into two parts allows the initial transmission to be routed according to the netID. The hostID only needs to be examined on arrival at the identified network. Before proceeding, it is important to note that the term 'host' is a little misleading because some devices, particularly routers, have more than one network interface and each interface requires a different IP address.

The other feature of the original scheme was that allocated addresses were based on the concept of different classes of networks. There were five classes; we are going to look at the first three classes. The structures used for the addresses are shown in Table 2.02.

Class	Class identifier	Number of bits for netID	Number of bits for hostID
Class A	0	7	24
Class B	10	14	16
Class C	110	21	8

Table 2.02 Address structure for three classes of IPv4 address

It can be seen from Table 2.02 that the most significant bit or bits identify the class. A group of the next most significant bits define the netID and the remaining, least significant, bits define the hostID. The reasoning behind this was straightforward. The largest organisations would be allocated to Class A. There could only be 2^7 i.e. 128 of these but there could be 2^{24} distinct hosts for each of them. This compared with 2^{21} (approximately two million) organisations that could be allocated to Class C but each of these could only support 2^8 i.e. 256 hosts.

The problems with this scheme arose once LANs supporting PCs became commonplace. The number of Class B netIDs available was insufficient but if organisations were allocated to Class C the number of hostIDs available was too small. There have been a number of different modifications made available to solve this problem.

Before considering some of these, the representation used for an IP address needs to be introduced. During transmission, the technology is based on the 32-bit binary code for the address; to make it simpler for users, we write the address using decimal numbers separated by dots. Each byte is written as the denary equivalent of the binary number represented by the binary code. For example, the 32 bit code:

```
    10000000 00001100 00000010 00011110
```

is written in dotted decimal notation as:

```
128.12.2.30
```

Discussion Point:
There were options available when the dotted decimal notation was chosen. Can you identify these?

Classless inter-domain routing (CIDR)

The first approach developed for improving the addressing scheme is called 'classless inter-domain routing' (CIDR). This retains the concept of a netID and a hostID but removes the rigid structure and allows the split between the netID and the hostID to be varied to suit individual need. The simple method used to achieve this is to add an 8-bit suffix to the address that specifies the number of bits for the netID. If, for instance, we define the suffix as 21, that means that 21 bits are used for the netID and there are 11 bits remaining (of a 32-bit address) to specify hostIDs allowing 2^{11} (i.e. 2048) hosts. One example of an IP address using this scheme is shown in Figure 2.10. The 21 bits representing the netID have been highlighted. The remaining 11 bits represent the hostID which would therefore have the binary value 11000001110.

Binary code: $\underbrace{110000110000110000000}_{\text{netID}}11000001110/\underbrace{00010101}_{\text{suffix}}$

Dotted decimal notation: 195.12.6.14/21

Figure 2.10 A CIDR IPv4 address

Note that with this scheme there is no longer any need to use the most significant bit or bits to define the class. However, it does allow already existing Class A, B or C addresses to be used with suffixes 8, 16 or 24, respectively.

TASK 2.03

Create an example of the binary code for a Class C address expressed in CIDR format. Give the corresponding dotted decimal representation.

Sub-netting

Sub-netting is a different approach. It allows a more efficient use of a hostID by applying a structure to it.

To illustrate an example of this we can consider a medium-sized organisation with about 150 employees each with their own computer workstation. Let's assume that there are six individual department LANs and one head-office LAN. Figure 2.11 shows a schematic diagram of how the LANs would be connected to the Internet if the original scheme were used. Note that the diagram has been simplified by showing the LANs connected to a gateway. This is a device that connects networks with different protocols. For the connection to the Internet the gateway would either first connect to a router or have the capability to act as a router itself.

The organisation would need seven individual Class C netIDs; one for each LAN. Each of these would point to one of the LAN gateways. The netID for each LAN would be identified by the first 24 bits of the IPv4 address, leaving 8 bits for the hostID. This would mean 256 individual codes for identifying different workstations on just one LAN. For the seven LANs the total number of workstations that could be identified would be:

$$256 \times 7 = 1792$$

Since the organisation only has 150 workstations in total, there are 1642 unused addresses. Not only would these be unused they would be unavailable for use by any other organisation.

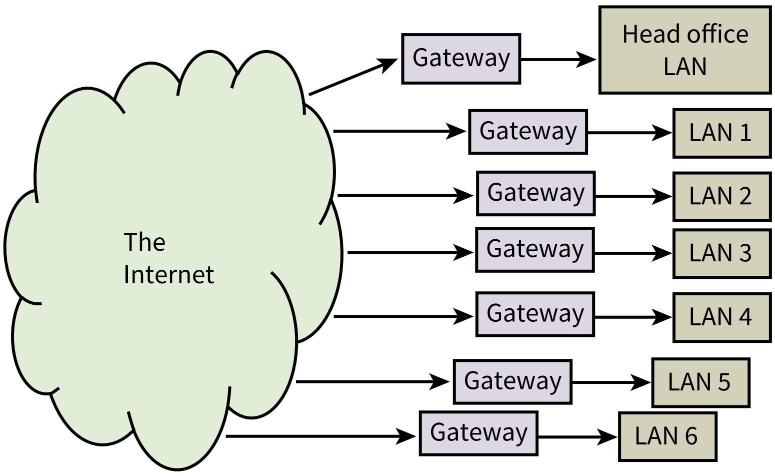


Figure 2.11 Connecting LANs using the original classful IPv4 scheme

The sub-netting solution for this organisation would require allocating just one Class C netID. For example, the IP addresses allocated might be 194.10.9.0 to 194.10.9.255 where the netID comprises the first three bytes, represented by the decimal values 194, 10 and 9.

The sub-netting now works by having a defined structure for the 256 codes constituting the hostID. A sensible solution for this organisation is to use the top three bits as a code for the individual LANs and the remaining five bits as codes for the individual workstations. Figure 2.12 shows a schematic diagram of this arrangement.

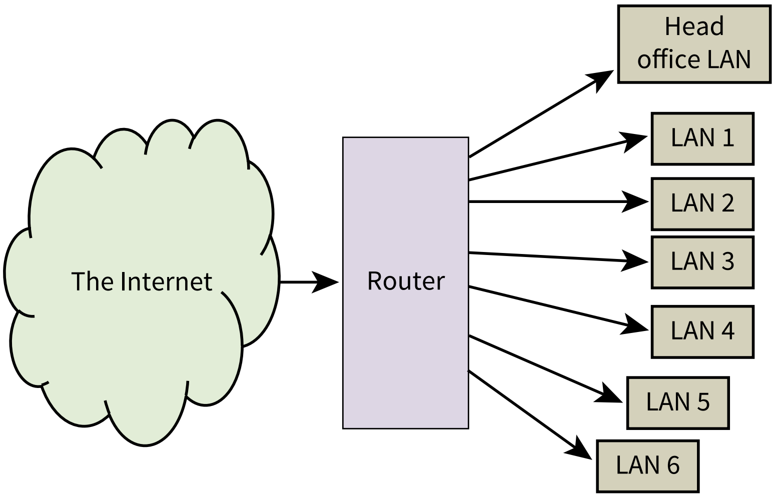


Figure 2.12 Connecting LANs using sub-netting

On the Internet, all of the allocated IP addresses have a netID pointing to the router. The router then has to interpret the hostID to direct the transmission to the appropriate workstations on one of the LANS via a gateway. Examples of workstation identification:

- hostID code `00001110` would be the address for workstation 14 on the head office LAN 0 (LAN 000)
- hostID code `01110000` would be the address for workstation 16 on LAN 3 (LAN 011).

With 150 workstations the organisation hasn't used all of the 256 allocated IP addresses. However, there are only 106 unused which is a reasonable number to have available in case of future expansion. Only one netID has been used leaving the other six that might have been used still available for other organisations to use.

Network address translation (NAT)

The final scheme to be considered is different in that it deviates from the principle that every IP address should be unique. In this scheme, provision has been made for large organisations to have private networks (intranets) which use the same protocols as those used for the Internet. One justification for using a private network has always been that this provides extra security because of the isolation from the Internet. However, this is no longer normal practice. Organisations want private networks but they also want Internet connectivity.

The solution for dealing with the addressing is to use network address translation (NAT). Figure 2.13 shows a schematic diagram of how this can be used.

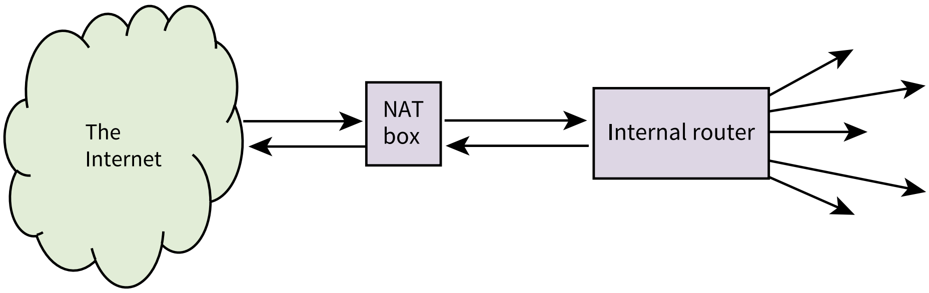


Figure 2.13 An intranet connected to the Internet using a NAT box

The NAT box has one IP address which is visible over the Internet and so can be used as a sending address or as a receiving address. Internally the IP addresses have to be chosen from one of the three ranges of IP addresses shown in Table 2.03 that have been allocated for such networks. (You do not need to remember these numbers!)

Lower bound	Upper bound
10.0.0.0	10.255.255.255
172.16.0.0	172.31.255.255
192.168.0.0	192.168.255.255

Table 2.03 IPv4 addresses to be used in private networks

The important point is that each address can be simultaneously used by any number of different private networks. There is no knowledge of such use on the Internet itself or in any other private network. The interface in the NAT box has software installed to examine each incoming or outgoing transmission. There can be a security check before an incoming transmission is directed to the correct internal address. The diagram shows undefined arrows from the router connected to the NAT box. These indicate that the network structure within the organisation could take many different forms.

Static and dynamic IP addresses

As discussed in Section 2.06, when a user wishes to have a connection to the Internet the connection is handled by an Internet Service Provider. The ISP will have available a large number of hostIDs. However, the number of users that the ISP is supporting could very likely be larger than the total number of addresses available. Fortunately for the ISP and for an individual user many of these potential users will not be engaged in Internet interaction. The normal practice is for the ISP to create a 'dynamic address' for a user. This is one that the ISP is free to change if it suits but more importantly the address is available for re-allocation once a user disconnects from the Internet. The alternative is a 'static address' which never changes and can be provided if a user is prepared to pay an extra charge.

Discussion Point:

Can you find out which IP addressing scheme is being used when you are connected to the Internet?

IPv6 addressing

Today there are combinations of IPv4 approaches in use and these allow the Internet to continue to function. Respected sources argue that this cannot continue beyond the current decade. There must soon be a migration to IP version 6 (IPv6), which uses a 128-bit addressing scheme allowing 2^{128} different addresses, a huge number! In practice, this will allow more complex structuring of addresses. Documenting these addresses is going to be difficult. The addresses are written in a colon hexadecimal notation. The code is broken into 16-bit parts, with each part represented by four hexadecimal characters. Fortunately, some abbreviations are allowed. A few examples are given in Table 2.04.

IPv6 address	Comment
68E6:7C48:FFFE:FFFF:3D20:1180:695A:FF01	A full address
72E6::CFFE:3D20:1180:295A:FF01	:0000:0000: has been replaced by ::
6C48:23:FFFE:FFFF:3D20:1180:95A:FF01	Leading zeros omitted
::192.31.20.46	An IPv4 address used in IPv6

Table 2.04 Some examples of IPv6 addresses

Extension Question 2.01

If IPv6 addressing is used, how many addresses would be available per square metre of the Earth's surface? Do you think there will be enough to go round?

2.09 Domain names

In everyday use of the Internet, a user needs to identify a particular web page or email box. As users, we would much prefer not to identify each IP address using its dotted decimal value! To get round this problem the **domain name service (DNS, also known as domain name system)** was invented in 1983. The DNS service allocates readable domain names for Internet hosts and provides a system for finding the IP address for an individual domain name.

The system is set up as a hierarchical distributed database which is installed on a large number of domain name servers covering the whole of the Internet. The domain name servers are connected in a hierarchy, with powerful root servers at the top of the hierarchy supporting the whole Internet. The root servers are replicated, meaning that multiple copies of all their data are kept at all times. DNS name space is then divided into non-overlapping zones. Each zone has a primary name server with the database stored on it. Secondary servers get information from this primary server.

As a result, the naming system is hierarchical. There are more than 250 top-level domains which are either generic (e.g. .com, .edu, and .gov) or represent countries (e.g. .uk and .nl).

The domain name is included in a universal resource locator (URL), which identifies a web page, or an email address. A domain is named by the path upward from it. For example, .eng .cisco.com. refers to the .eng subdomain in the .cisco domain of the .com top-level domain.

Looking up a domain name to find an IP address is called ‘name resolution’. For such a query there are three possible outcomes.

- If the domain is under the control of the server to which the query is sent then an authoritative and correct IP address is returned.
- If the domain is not under the control of the server, an IP address can still be returned if it is stored in a cache of recently requested addresses but it might be out of date.
- If the domain in the query is remote then the query is sent to a root server which can provide an address for the name server of the appropriate top-level domain. This in turn can provide the address for the name server in the next lower domain. This continues until the query reaches a name server that can provide an authoritative IP address.

Reflection Point:

In several places you have been asked to carry out some research. Are you using the most efficient search methods? Specifically, how could they be improved?

Summary

- Client-server and peer-to-peer networking are options for file sharing.
- The star topology is the one most commonly used for a LAN.
- The main transmission media are copper (twisted pair, coaxial) cables, fibre-optic cables and wireless (radio, microwave, infrared).
- Factors to consider when choosing a medium are bandwidth, attenuation, interference and the need for repeaters.
- CSMA/CD (carrier sense multiple access with collision detection) has been used to detect and avoid message collisions in shared media.
- The Internet is the largest internetwork in existence.
- ISPs provide access to the Internet.
- Internet infrastructure is supported by PSTNs and cell phone companies.
- The World Wide Web is a distributed application accessible on the Internet.
- The current Internet addressing scheme is IPv4, with IPv6 a future contender.
- The DNS resolves a domain name to an IP address.

Exam-style Questions

1 A new company has been established. It has bought some new premises which consist of a number of buildings on a single site. It has decided that all of the computer workstations in the different buildings need to be networked. They are considering ways in which the network might be set up.

a One option they are considering is to use cabling for the network and to install it themselves.

i Name the **three** types of cabling that they might consider. [2]

ii Explain **two** factors, other than cost, that they need to consider when choosing suitable cabling. [4]

b Another option they are considering is to use wireless technology for at least part of the network.

i Explain **one** option that might be suitable for wireless networking. [2]

ii Identify **one** advantage, other than cost, of using wireless rather than cable networking. [1]

iii Identify **one** disadvantage (other than cost) of using wireless rather than cable networking. [1]

c The final option they are considering is to use the services of a PSTN.

i Define what a PSTN is or does. [1]

ii Explain how a PSTN could provide a network for the company. [3]

2 **a** The Domain Name System is vitally important for Internet users.

i Name the type of software used by the system and the type of hardware on which the software is installed. [2]

ii Name **two** types of application that use the Domain Name System and for each give a brief description of how it is used. [4]

b In the classful IPv4 addressing scheme, the 32-bit binary code for the address has the top (most significant) bit set to 0 if it is of class A, the top two bits set to 10 if class B or the top three bits set to 110 if class C. In a document an IPv4 address has been written as 205.124.16.152.

i Give the name for this notation for an IP address and explain how it relates to the 32-bit binary code. [2]

ii Identify the class of the address and explain your reason. [2]

iii Explain why an IPv4 address defines a netID and a hostID. [3]

c If the CIDR scheme for an IPv4 address is used the IP address 205.124.16.152 would be written as:

205.124.16.152/24

State the binary code for the hostID in this address, with a reason. [2]

3 A user watches a video provided by a website that uses on-demand bit streaming.

Describe the measures needed to ensure that the video does not periodically pause when it is being watched. [6]

4 **a** Describe where private IP addresses can be used. [2]

b Explain how it can be ensured that private and public IP addresses are not used in the wrong context. [4]

5 **a** An IP address has the following value:

11.64.255.90

i Write the above IP address in hexadecimal. [4]

ii Explain the format of an IP address. [2]

b Study the following sentence:

“When a user enters a URL into their web browser, the DNS service locates the required resource.”

Explain how a URL and DNS are used to locate a resource. [4]

Cambridge International AS & A level Computer Science 9608 paper 12 Q9 June 2015

6 Access to World Wide Web content uses IP addressing.

a State what IP stands for. [1]

b The following table shows four possible IP addresses.

Indicate for each IP address whether it is valid or invalid and give a reason.

Address	**Denary/Hexadecimal**	**Valid or Invalid**	**Reason**
`3.2A.6AA.BBBB`	Hexadecimal		
`2.0.255.1`	Denary		
`6.0.257.6`	Denary		
`A.78.F4.J8`	Hexadecimal		

[4]

c Describe **two** differences between public and private IP addresses. [2]

Cambridge International AS & A level Computer Science 9608 paper 11 Q7 June 2016

Chapter 3: Hardware

Learning objectives

By the end of this chapter you should be able to:

- show understanding of the need for input, output, primary memory and secondary (including removable) storage
- show understanding of embedded systems
- describe the principal operations of hardware devices
- show understanding of the use of buffers
- explain the differences between Random Access Memory (RAM) and Read Only Memory (ROM)
- explain the differences between Static RAM (SRAM) and Dynamic RAM (DRAM)
- explain the difference between Programmable ROM (PROM), Erasable Programmable ROM (EPROM) and Electrically Erasable Programmable ROM (EEPROM).

3.01 Overview of computer system hardware functionality

A computer system has to support three major areas of operational capability:

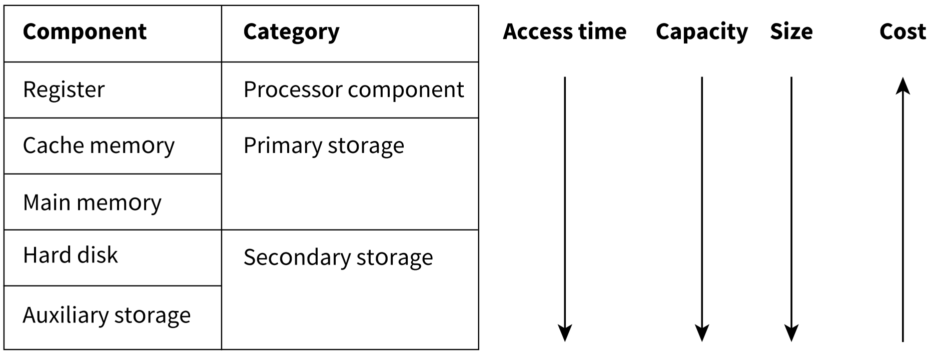
- the processing of data
- the storage of data
- the input and output of data.

At the heart of the system the processing of data is carried out by the CPU (Central Processing Unit). The workings of the CPU are the subject of Chapter 5 and will not be discussed further here.

Data storage

The terminology used in the computer literature to describe components for storing data is not always consistent. One variation is to distinguish between memory as the component which the processor can access directly and the (file-) store used for long-term storage. An alternative is to describe the memory as primary storage and the remainder as secondary storage.

Whatever names are used, the memory hierarchy is a useful concept when we choose the components to be used in a computer system for data storage. Figure 3.01 shows a version of this hierarchy that includes the trends in the important factors that affect our choice. The factors increase in the direction of the arrow. The register is a component within the CPU that has the fastest access speed. The cache memory has faster access speed than that of main memory, particularly when the cache is a built-in part of the CPU chip.



Component	Category
Register	Processor component
Cache memory	Primary storage
Main memory	
Hard disk	Secondary storage
Auxiliary storage	

Figure 3.01 Trends in the factors affecting the choice of memory components

Computer users would really like to have a large amount of primary storage that costs little and allows quick access. This is not possible; the fastest components cost more and have limited capacity. In practice, the choice made is a compromise. It could be argued that there is a need for secondary storage because the use of primary storage alone would be far too expensive. However, it is more sensible simply to recognise that long-term storage of data requires separate dedicated components.

An alternative approach when choosing a storage medium or device is to consider which of the following applies:

- it is an integral part of the system, to which the user cannot normally get access – the options here are a hard disk or solid-state drive
- it is an individual item that can be inserted into a drive which is part of the computer system or which can be connected to it – could be a floppy disk, optical disc or magnetic tape cartridge
- it is a peripheral device that can be connected to the system when needed – there are many possibilities here including a hard drive, a memory stick or a memory card
- it is a portable item that the user can carry around with them for attaching to different systems; possibly used for personal backup – usually a flash memory stick nowadays but a floppy disk or optical disc is an alternative

- it is remote from the system, possibly accessible via a network connection; often used for backup – cloud storage is one option, but others are magnetic tape, RAID (Redundant Arrays of Independent Disks) or SAN (Storage-Area Network).

Data output

For data output from a computer system the following options are available:

- screen display
- hardcopy using a printer or plotter
- virtual headset display
- a speaker
- writing to any of the data storage devices described earlier
- transmission on a network link.

Data input

For the input of data to a computer system the following are among the options available:

- keyboard or keypad entry by a user
- user interaction with a screen using screen icons or menus; possibly using a pointing device and possibly involving the use of a touch screen
- a user using a game controller
- a user using a scanner
- a user using a microphone in tandem with voice recognition software
- reading from any of the storage devices described earlier
- transmission on a network link.

Note that input and output in a computer system are controlled by an I/O sub-system. This handles data input to or output from the computer system as well as data read from or written to the internal hard disk or solid-state drive.

3.02 Embedded systems

Much of the hardware in Section 3.01 relates to what we can call a general-purpose computer system. We also need to consider embedded computer systems because there are many more of these systems in use than there are general-purpose systems. Any manufactured item that has mechanical or electrical parts will almost certainly contain one or more embedded systems.

An embedded system must still contain a processor, memory and an I/O capability. If these are constructed on one chip this is called a microcontroller. For some applications the system will have input and output solely associated with the internal workings of the host system. In other cases, perhaps when serving a monitoring or control function, there might be input from within the system but some output is provided to the user. Alternatively, the embedded system can provide a full user interface as, for example, in a mobile phone.

The major advantage of embedded systems is that they are special-purpose; possibly performing only a single function. This function is likely to be required in a wide variety of different manufactured products. Mass production of an embedded system brings economies of scale: the more we make, the cheaper they become. During the early years of their use, embedded systems had the disadvantage that programming was difficult because the memory space available to store a program was limited. For this reason, programs had to be short. In addition, there was the disadvantage that if errors were found following installation then new chips had to be manufactured and used to replace the faulty ones. In modern systems these problems are less likely, but a new problem has developed. Embedded systems are now part of what is called the IoT (Internet of Things). More and more embedded systems are being installed with a network connection. This can greatly improve the usefulness of a product, for example by providing information and updates to the owner. However, this accessibility via a network is a security concern. Embedded systems are less likely to be protected against unlawful actions than general-purpose systems.

Discussion Point:

How might useful information from an embedded system installed in a domestic appliance be communicated over a network to the owner of the appliance?

3.03 Memory components

The components that make up the main memory of a general-purpose computer system are called **random-access memory (RAM)**. The name has been chosen because such memory can be accessed at any location independently of which previous location was used. Because of this it might have been better called ‘direct-access memory’. Another possible name would be ‘read–write memory’ because RAM can be repeatedly read from or written to. A key feature of RAM is that it is volatile, which means that when the computer system is switched off the contents of the memory are lost.

There are two general types of RAM technology. Dynamic RAM (DRAM) is constructed from capacitors that leak electricity and therefore need regularly recharging (every few milliseconds) to maintain the identity of the data stored. Static RAM (SRAM) is constructed from flip-flops that continue to store data indefinitely while the computer system is switched on. The circuits and logic for flip-flops are discussed in Chapter 19 (Section 19.02)).

The major difference between the two types of RAM is that DRAM requires fewer electronic components per bit stored. This means DRAM is cheaper to make and has a higher density for data storage. The major advantage of SRAM is that it provides shorter access time. In a general-purpose computer system, it is normal practice for main memory to be constructed from DRAM but for cache memory to be provided by SRAM because of the faster access speed. By contrast, embedded systems that need RAM with only limited capacity often use SRAM for this.

The second category of memory component is called **read-only memory (ROM)**. Again, this name does not give a full picture of the characteristics of this type of component. ROM shares the random-access or direct-access properties of RAM. However, as the name implies it cannot be written to when in use within the computer system. The other key feature is that the data in ROM is not lost when the computer system is switched off; the memory is non-volatile.

TIP

The word volatile has several meanings. Try to remember that volatile memory no longer stores data when the system is switched off.

ROM has specialised uses for the storage of data or programs that are going to be used unchanged over and over again. In a general-purpose system the most important use is in storing the bootstrap program. This is a program that runs immediately when a system is switched on. There are a number of other uses for ROM in such a system, some of which we will see later in this book. In addition, ROM is used in many embedded systems.

There are four different types of ROM.

1. In the simplest type of ROM the programs or data are installed as part of the manufacturing process. If different contents are needed the chip must be replaced.
2. An alternative is Programmable ROM (PROM). The manufacturer of the chip supplies chips to a system builder. The system builder installs the program or data into the chips. This allows the system builder to test some samples of programmed chip before committing the whole batch to be programmed. As with the simplest type of ROM, the program or data once installed cannot be changed.
3. A more flexible type of ROM is Erasable PROM (EPROM). The installed data or program can be erased (using ultraviolet light) and new data or a new program can be installed. However, this reprogramming usually requires the chip to be removed from the circuit.
4. The most flexible type of ROM is Electrically Erasable PROM (EEPROM). As the name suggests, this works in a similar way to EPROM, except an electrical signal can be used to remove existing data. This has the major advantage that the chip can remain in the circuit while the contents are changed. However, the chip is still used as read-only.

Discussion Point:

Can you find out what memory components are in the computer system you are using and any details about them such as the type and storage capacity?

Buffers

Whenever data has to be transferred from one part of a computer system to another, a problem occurs if the data can be sent more quickly than it can be received. The solution to the problem is to use a **buffer**. Data enters a buffer before being transmitted to its destination. The buffer functions as a queue so the data emerges in the order that it has entered the buffer. Typically, the buffer is created in the computer memory.

Question 3.01

Can you think of examples of data transfer that would need a buffer?

3.04 Secondary storage devices

Before discussing storage devices, we should introduce some terminology. For any hardware device, whether an integral part of the computer system or a connected peripheral, its operation requires appropriate software to be installed. This software is referred to as the 'device driver'. This should not be confused with the term 'drive' associated specifically with a storage device. The term 'drive' initially referred to the hardware that housed a storage medium and physically transferred data to it or read data from it. However, as so often happens, such distinctions are often ignored. As a result, for example, references to a 'hard disk', a 'hard disk drive' or to a 'hard drive' have the same meaning.

Magnetic media

Magnetic media have been the mainstay of filestore technology for a very long time. The invention of magnetic tape for sound recording pre-dates the invention of the computer by many years. As a result, magnetic tape was the first storage device. In contrast, the hard disk was specifically invented for computer storage. The hard disk also used magnetisation to write data, and arrived a few years after magnetic tape was first used for storage.

For either type of magnetic medium the interaction with it is controlled by a read head and a write head. A read head uses the basic law of physics that a state of magnetisation will affect an electrical property; a write head uses the reverse law. Although they are separate devices the two heads are combined in a read–write head. The two alternative states of magnetisation are interpreted as a 1 or 0.

A schematic diagram of a hard disk is shown in Figure 3.02. Points to note about the physical construction are:

- there is more than one platter (disk)
- each platter has a read–write head for each side
- the platters spin in unison (all together and at the same speed)
- the read–write heads are attached to actuator arms which allow the heads to move over the surfaces of the platters
- the motion of each read–write head is synchronised with the motion of the other heads
- a cushion of air ensures that a head does not touch a platter surface.

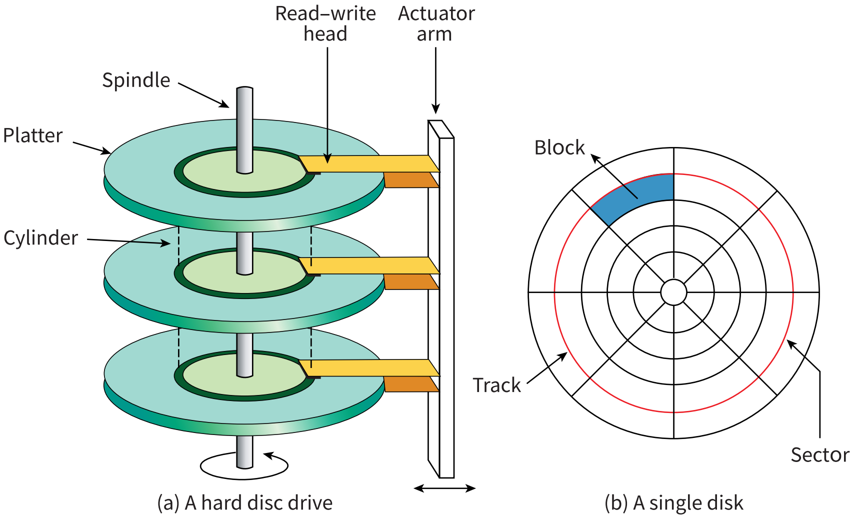


Figure 3.02 A schematic drawing of the components of a hard disk drive

Data are stored in concentric tracks (tracks sharing the same centre). Each track consists of a sequence of bits. These are formatted into sectors where each sector contains a defined number of bytes. The sector becomes the smallest unit of storage. Because the movement of the heads is synchronised, the same tracks on different disks can have related data stored on them. These are accessible by just one movement of the head. The collection of tracks is referred to as a ‘cylinder’.

To store a file, a sufficient number of sectors have to be allocated but these might or might not be next to each other. As files are created and subsequently deleted or edited the use of the sectors becomes increasingly fragmented, which degrades the performance of the disk. A defragmentation program can reorganise the allocation of sectors to files to restore performance. This is discussed in Chapter 8 (Section 8.03).

A hard drive is considered to be a direct-access read–write device because any sector can be chosen for reading or writing. However, the data in a sector has to be read sequentially (in order).

This is only a simplified explanation of hard drive technology. There are several issues that arise when making hard drives. For example, the length of a track on the disk gets larger as you move from centre to edge. Manufacturers have to take account of this in their designs, otherwise the data storage capacity will be less than it potentially might be.

Optical media

As with the magnetic tape medium, optical storage was developed from existing technology not associated with computing systems. The compact disc (CD) evolved into CD digital audio (CD-DA) and this became the technology used in the CD-ROM. This was extensively used for distributing software but was of no value as a replacement for the floppy disk. The read–write version (CD-RW) which came later finally meant CD was a complete alternative to floppy disks. However, the CD has now given way to the DVD (originally ‘digital video disc’ but later renamed as ‘digital versatile disc’). The latest and most powerful technology is the Blu-ray disc (BD).

A schematic diagram of a design for an optical disc drive is shown in Figure 3.03. This is equipped to read a CD with infrared laser light of wavelength 780 nm or a DVD with red laser light of wavelength 680 nm.

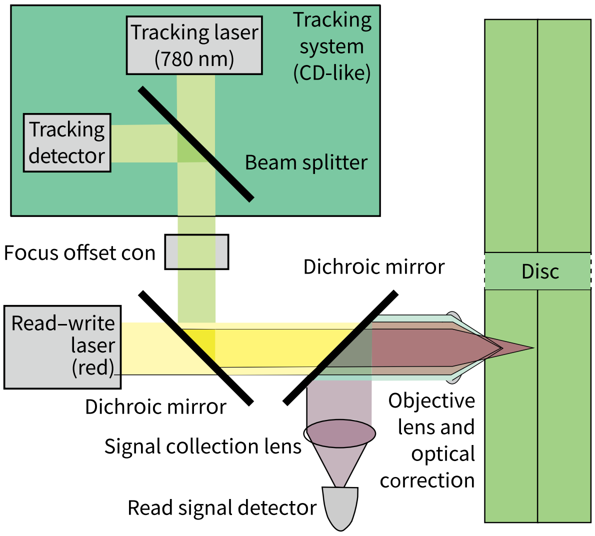


Figure 3.03 A schematic drawing of an optical disc drive

We can ignore the finer details of the construction of the drive and concentrate on the principles of how it operates. The important features for the process of reading data from the disc are as follows.

- The optical disc has one spiral track running from the inner extreme of the surface to the outer edge.
- During operation, the disc spins.
- Simultaneously the laser moves across ensuring that it is continuously focused on the spiral track.
- The track on the surface of the disc has what are referred to as ‘pits’ and ‘lands’.
- The laser beam is reflected from the surface of the disc.

- The difference between the reflection from a pit compared to that from a land can be detected.
- This difference in the intensity of the light the detector receives can be interpreted as either a 1 or a 0 to allow a binary code to be read from the disc.

For CD-RW and DVD-RW technologies, the reflective surface is a special alloy material. When data is being written to the disc (the ‘burn’ process) the heat generated by the absorption of the laser light changes the material to liquid form. Depending on the intensity of the laser light the material reverts to either a crystalline or an amorphous solid form when it cools. When the disc is read, the laser light is reflected from the crystalline solid but not from the amorphous solid, allowing the coding of a 1 or 0.

Despite there being only one track the disc functions as a direct-access device because the laser can move forwards or backwards. The data is formatted into sectors along the track in a similar way to the formatting of a magnetic hard disk.

Another similarity with magnetic disk technology is that the storage capacity is dependent on how close together individual physical representations of a binary digit can get. There are two aspects governing this for an optical disc. The speed of rotation is one but the most important is the wavelength of the light. Shorter wavelength light can be better focused. This is why a DVD can store more than a CD but much less than a Blu-ray disc.

Solid-state media

Despite the continued improvement in optical technology there is now a powerful competitor in the form of solid-state storage. The basis for this is ‘flash’ memory, which is a semiconductor technology with no moving parts. The circuits consist of arrays of transistors acting as memory cells. The most frequently used technology is called ‘NAND’ because the basic circuitry resembles that of a NAND logic gate (see Chapter 4 Section 4.04) with the memory cells connected in series. The writing to the memory and the reading from it is handled by a NAND flash controller. The special feature is that blocks of memory cells can have their contents erased all at once ‘in a flash’. Furthermore, before data can be written to a block of cells in the memory the data in the block first has to be erased. A block consists of several pages of memory. When data is read, a single page of data can be read in one operation.

The most frequent use is either in a memory card or in a USB flash drive (memory stick). In the latter case the flash memory is incorporated in a device with the memory chip connected to a standard USB connector. This is currently the technology of choice for removable data storage. How long this will remain so is uncertain with alternative technologies such as phase-change random access memory (PRAM) already under development.

The alternative use is as a substitute for a hard disk when it is often referred to as a solid-state drive (SSD). You might think that, with no moving parts, the technology would last forever. This is not true; with continuous use there is a degradation in the material used for construction. However, this is only gradual and it can be detected and its effects corrected for. Another major advantage over the traditional hard drive is the faster access speed.

Extension Question 3.01

Carry out some research into the technologies currently available for storage.

Consider first the options available for the storage device inside a laptop computer. Create a table showing cost, storage capacity and access speed for typical examples. Then consider the options available for peripheral storage devices. Create a similar table for these.

Can you identify which technologies remain viable and which ones are becoming uncompetitive? Are there any new technologies likely to come into common use?

3.05 Output devices provided for a user of a general-purpose computer system

Screen display

Chapter 1 (Section 1.05) described how an image could be stored as a bitmap built up from pixels. Screen displays are also based on the pixel concept but with one major difference. A screen pixel consists of three sub-pixels typically one each for red, green and blue. Varying the level of light emitted from the individual sub-pixels allows a full range of colours to be displayed.

There have been a number of very different technologies used to create a pixel. In the original cathode ray tube (CRT) technology, there is no individual component for a pixel. The inner surface of the screen is covered with phosphor, which is a material that emits light when electrons fall on it. An individual pixel is lit up by controlling the direction of the electron beam used. Colour CRT displays have individual red, green and blue phosphors arranged so as to create an array of pixels.

Flat-screen technologies now dominate. The **liquid-crystal display (LCD)** screen is an example. It has individual cells containing a liquid crystal to create each pixel. The pixel matrix is illuminated by back-lighting and each pixel can affect the transmission of this light to create the on-screen display. A typical arrangement is shown in Figure 3.04.

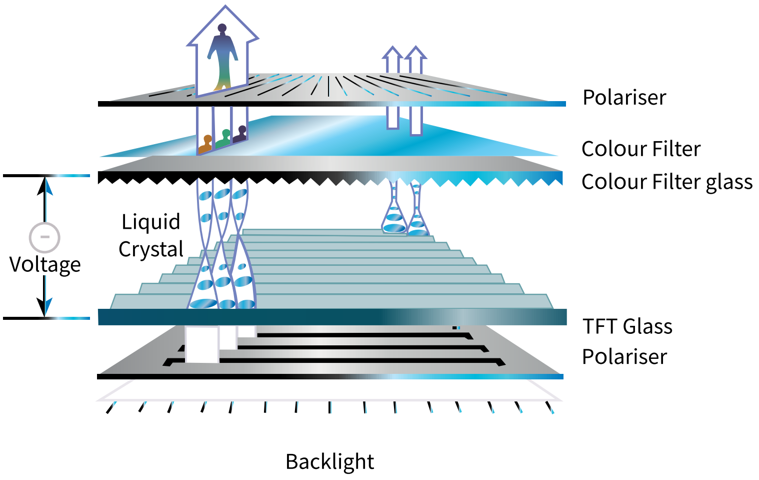


Figure 3.04 The components of a liquid-crystal display screen

The back-lighting is usually provided by light-emitting diodes (LEDs). Polarised light is directed towards the pixel matrix and a further polariser is placed between the pixel matrix and the screen. If a voltage is applied to an individual pixel cell the alignment of the liquid crystal molecules is affected. This changes the polarisation of the light and so changes what is displayed on the screen.

Virtual reality headset

The most important components of a virtual reality headset are the two eye-pieces. These are fed paired images from the controlling system which, when looked at together, give the eyes the sensation of being in a 3D environment. The images can be collected using specialised photographic techniques or can be created using a 3D graphics package. The wearer of the headset can control which part of the 3D environment is in view. They do this by moving their head or by using a controlling device.

Hard-copy output of text

Two technologies have come to dominate the printing of documents from data stored in a computer system. These are the inkjet printer and the laser printer. Both these technologies can be used to print text or images.

An inkjet printer works in the following way. A sheet of paper is fed in; the printhead moves across the sheet depositing ink on to the paper; the paper is moved forward a fraction and the printhead moves across the paper again. This continues until the sheet has been fully printed. The printhead consists of nozzles that spray droplets on to the paper. Ink is supplied to the printhead from one or more ink cartridges.

A schematic diagram of the workings of a laser printer is shown in Figure 3.05. The operation can be summarised as follows.

1. The drum is given an electric charge.
2. The drum starts to revolve step by step.
3. At each step a laser beam is directed by the mirror and lens assembly to a sequence of positions across the width of the drum.
4. At each position the laser is either switched off to leave the charge on the drum or switched on to discharge the position.
5. This process repeats until a full-page electrostatic image has been created.
6. The drum is coated with a charged toner that only sticks to positions where the drum has been discharged.
7. The drum rolls over a sheet of paper which is initially given an electric charge.
8. The sheet of paper is discharged and then is passed through heated rollers to fuse the toner particles and seal the image on the paper surface.
9. The drum is discharged before the process starts again for the next page.

The above sequence represents black and white printing.

For colour printing, separate toners are required for the colours and the process has to take place for each colour. The colours are created from cyan, magenta, yellow and black. The technology produces dots. Image quality depends on the number of dots per inch and software can control the number of dots per pixel.

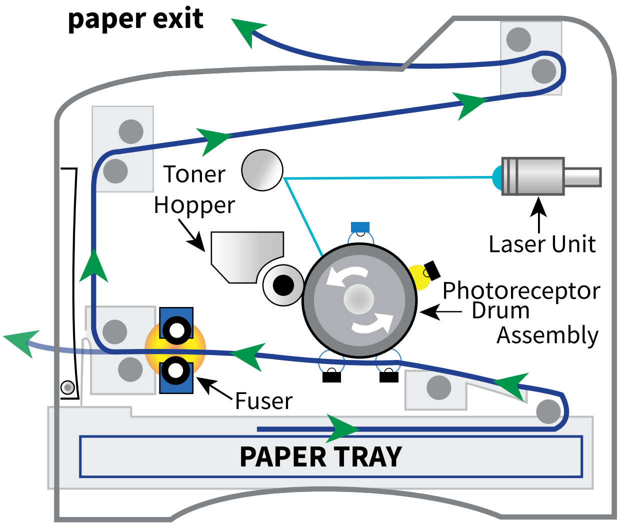


Figure 3.05 A schematic diagram of a laser printer

The same principles apply for colour printing using an inkjet printer, where separate colour inks are used.

Hard-copy graphics output

As discussed in Chapter 1 (Section 1.05) a graphic image can be stored either as a bitmap or as a vector graphic. The printing technology described above can be used to print a hard-copy of a bitmap. If a vector graphic file has been created the image can be displayed on a screen or printed by first converting the file to a bitmap version. However, specialised technical applications often require a more accurate representation to be created on paper. This requires the use of a graphics plotter. A plotter uses pens to write, usually, on a large sheet of paper constrained by sprockets along one pair of sides. The sprockets can move the paper forwards or backwards and pens can either be parked or in use at any given time. The controlling circuitry and software can create the drawing directly from the original vector graphic file.

Graph plotters are used by engineers and designers working in manufacturing. Engineers and designers may also use a 3D printer, which is a device that offers an alternative technology for computer-aided manufacture (CAM).

Figure 3.06 A bionic ear created using a 3D printer

A 3D design is created in a suitable computer-aided design (CAD) package. The design is split into layers. The data for the first layer is transmitted to the 3D printer. Rather than using ink to draw the layer, the 3D printer uses a nozzle to squirt material on to the printer bed to create a physical layer to match the design. This process is repeated for successive layers. When the whole object has been formed it has to be cured in some way to ensure that the layers are stuck together and the material has been converted to the form required for the finished product.

The technology is versatile. Figure 3.06 shows a striking example.

(For those of you interested in the details of Figure 3.06: the bionic ear was constructed with three ‘inks’. Silicone was used for the basic structure, a gel containing chondrocyte cells and silicone infused with silver nanoparticles were the other two ‘inks’. The final curing step involved incubation in a culture medium to allow the chondrocyte cells to produce cartilage. The only missing component was skin.)

3.06 Input devices provided for a user of a general-purpose computer system

The keyboard

The keyboard allows a user to input text data. During text input it appears as though a key press immediately transfers the appropriate character to the computer screen, but this is an illusion. The key press has to be converted to a character code, which is transmitted to the processor. The processor, under the control of the operating system, ensures that the text character is displayed on the screen. The same process takes place if the keyboard is used to initiate some action, perhaps by using a shortcut key combination. The difference is that the processor has to respond by taking the requested action.

To achieve this functionality the keyboard has electrical circuitry together with its own microprocessor and a ROM chip. The significant details of how a keyboard works are as follows.

- The keys are positioned above a key matrix, which consists of a set of rows of wires and another set of columns of wires.
- Pressing a key causes contact at one of the points where wires cross.
- The microprocessor continuously tests to see if any electrical circuit involving a row wire and a column wire has become closed.
- When the microprocessor recognises that a circuit has become closed, it can identify the particular intersection (wire crossing point) that is causing this.
- The processor then uses data stored in the ROM to identify the character code relating to the key associated with that intersection and sends this character to the screen.

The screen

There are a number of ways in which a user can cause data to be input through an interaction with a screen. At one time a computer system user only had access to a keyboard and a screen acting as a monitor. Even then the software could display a menu on the screen and the user could choose an option by keying in a number from the menu.

A significant step forward came with the introduction of graphical user interfaces (GUIs) as standard features for microcomputer systems in the 1980s. A GUI provides a number of different types of screen icon, each of which allows the user to control data input. The user needs a pointing mechanism to use a GUI effectively. One example of a pointing mechanism is a computer mouse that controls the position of a cursor on the screen. The screen is now not just an output device but also an input device activated by a mouse click.

Touch screens

The early versions of touch screen technology worked with a CRT screen but could equally well be used with a flat screen. The mechanism required emitters to be positioned on the sides of the screen with detectors positioned opposite to them. The emitters produced either infrared light or ultrasonic waves. When a finger touched the screen and blocked some of the light or ultrasound, some of the detectors would measure a reduced signal level.

As well as providing improved display capability, flat-screen technology has allowed new mechanisms for touch screen interaction.

The modern version of a touch-sensitive screen has layers providing the light output by the display with further, touch-detecting layers added immediately beneath the surface of the screen. There have been two approaches used. The first is the **resistive touch screen**. This type has two layers separated by a thin space beneath the screen surface. The screen is not rigid so when a finger presses on to the screen the pressure moves the topmost of these two separated layers, so that the top layer makes contact with the lower layer. The point of contact creates a voltage divider in the horizontal and vertical directions.

The second technology is the **capacitive touch screen**. This does not require a soft screen but instead makes use of the fact that a finger touching a glass screen can cause a capacitance change in a circuit component immediately below the screen. The most effective technology is projective capacitive touch (PCT) with mutual capacitance. PCT screens have a circuit beneath the screen that contains an array of capacitors. This capacitive technology can detect the touch of several fingertips at the same time, which allows for more sophisticated applications.

In any type of touch screen the processor takes readings from measuring devices and uses these readings to calculate the position of the touch. This calculation then allows the processor to set in motion whatever action the user was requesting.

Extension Question 3.02

Consider the different possibilities for interacting with a screen display. Create a table showing the advantages and disadvantages for each technique.

Discussion Point:

Investigate which flat-screen technologies are used in any computer, laptop, tablet or mobile/cell phone that you use. Discuss the benefits and drawbacks associated with their use.

Input of a graphic

There are several ways to store and use image (graphic) data in a computer. A webcam is a device used to stream video images into a computer system. A digital camera can be connected to a computer and stored images or videos can then be downloaded into the computer. Another option is to use a scanner. Effectively, a scanner reverses the printing process in that it takes an image and creates a digital representation from it. A sheet of paper containing the image (which may be text) is held in a fixed position and a light source moves from one end of the sheet to the other. It covers the width of the paper. The reflected light is directed by a system of mirrors and lenses on to a charge-coupled device (CCD).

You don’t need to know the details of how a CCD works, but three aspects to note are:

- a CCD consists of an array of photo-sensitive cells
- a CCD produces an electrical response proportional to the light intensity for each cell
- a CCD needs an analogue-to-digital converter to create digital values to be transmitted to the computer.

3.07 Input and output of sound

Voice input and output

IP telephony and video conferencing are two applications that require both voice input and voice output. In addition, voice recognition can be used as an alternative technique for data input to a computer and voice synthesis is being used for an increasing variety of applications.

For input, a microphone is needed. This is a device that has a diaphragm, a flexible material that is caused to vibrate by an incoming sound. If the diaphragm is connected to suitable circuitry the vibration causes a change in an electrical signal. A condenser microphone uses capacitance change as the mechanism; an alternative is to use a piezoelectric crystal. The analogue electrical signal is converted to a digital signal by an analogue-to-digital (ADC) converter so that it can be processed inside the computer.

For output, a speaker (loudspeaker) is needed. How this works is effectively the reverse process to that for input. Digital data from the computer system is converted to analogue by a digital-to-analogue (DAC) converter. The analogue signal is fed as a varying electrical current to the speaker. In most speakers, the current flows through a coil suspended within the magnetic field provided by a permanent magnet in the speaker. As the size and direction of the current keep changing, the coil moves backwards and forwards. This movement controls the movement of a diaphragm, which causes sound to be created.

The input and output are controlled by a sound (audio) card installed in the computer.

Other types of sound input and output

Music as well as voice sounds can be recorded or live streamed in the same way that voices are recorded. Some sound recording devices carry out the analogue to digital conversion very early on in the process so that all the sound processing is done digitally. Music can be output via speakers or stored in digital form for later play back.

Reflection Point:

The description ‘peripheral’ is often used to describe devices that can be connected to a computer. In your research did you come across the word being used? Is it a useful one or is it possibly not so because of the lack of a clear definition?

Summary

- Primary storage is main memory, consisting of RAM (DRAM or SRAM) and ROM (possibly PROM, EPROM or EEPROM).
- Secondary storage includes magnetic, optical and solid-state media.
- Output devices include screens, printers, plotters and speakers.
- Input devices include the keyboard, scanner and microphone.
- Screens can be used for both input and output.

Exam-style Questions

1 **a** Examples of primary and secondary storage devices include:

- hard disk
- DVD-RW
- flash memory

For each device, describe the type of media used.

Hard disk

DVD-RW

Flash memory [3]

b Describe the internal operation of the following devices:

DVD-RW

2 **a** Pressing a key on a computer keyboard can cause a character to be displayed on the computer screen.

i Identify **four** aspects of the basic internal operation of a keyboard that makes this happen. [4]

ii Describe an alternative method for a user to enter some text into a computer system. [2]

b **i** In the operation of a laser printer there are a number of initial stages which lead up to the creation of a full-page electrostatic image. Identify **three** of these stages and present them in the order that they would occur. [3]

ii Identify **two** of the stages that make use of this electrostatic image. [2]

iii State the difference in the procedure used for colour printing from that used for black and white printing. [1]

3 **a** Describe the operation of a touch screen technology that can be used in association with any type of computer screen. [4]

b Describe the operation of a touch screen technology that is only applicable for use with a flat screen. [4]

4 **a** Examples of primary and secondary storage devices include: [3]

- hard disk
- DVD-RW
- flash memory

For each device, describe the type of media used.

Hard disk

DVD-RW

Flash memory [3]

b Describe the internal operation of the following devices:

- DVD-RW
- DVD-RAM [2]

Cambridge International AS & A level Computer Science 9608 paper 12 Q1 November 2015

5 **a** Describe **two** differences between RAM and ROM. [2]

b State **three** differences between Dynamic RAM (DRAM) and Static RAM (SRAM). [3]

Cambridge international AS & A Level Computer Science 9608 paper 12 Q6 November 2016

Chapter 4
Logic gates and logic circuits

Learning objectives

By the end of this chapter you should be able to:

- use logic gate symbols for NOT, AND, OR, NAND, NOR and XOR
- understand and define the functions of the NOT, AND, OR, NAND, NOR and XOR (EOR) gates
- construct the truth table for each of the above logic gates
- construct a logic circuit from:
  - a problem statement
  - a logic expression
  - a truth table
- construct a truth table from:
  - a problem statement
  - a logic circuit
  - a logic expression
- construct a logic expression from:
  - a problem statement
  - a logic circuit
  - a truth table.

4.01 Boolean logic and problem statements

Consider the following question:

Is Colombo further north than Singapore?

In everyday language the answer will be either yes or no. (‘Yes’, in fact.) However, the question could be rephrased to make use of the language of Boolean logic:

Colombo is further north than Singapore: TRUE or FALSE?

More formally, the statement:

Colombo is further north than Singapore.

can be described as an example of a logic assertion or a **logic proposition** that can have only one of the two alternative Boolean logic values: TRUE or FALSE.

Now consider the following two individual statements.

- You should take an umbrella if *it is raining* or if *the weather forecast is for rain later*.
- The air-conditioning system is set to come on in an office only *during working hours* but also only *if the temperature rises to above 25°C*.

Each of these statements contains two logic propositions which are highlighted. In each statement these logic propositions are combined in some way. Finally, each statement has the addition of an outcome which is dependent on the combination of the two propositions. Each of these is, therefore, an individual example of a **problem statement**.

4.02 Boolean operators

The problem statements identified above can be more formally expressed in a form that is suitable for handling with Boolean logic. To do this it is necessary to use Boolean operators. The three basic Boolean operators are AND, OR and NOT.

The definition for AND, OR and NOT can be expressed as:

- A AND B is TRUE if A is TRUE and B is TRUE
- A OR B is TRUE if A is TRUE or B is TRUE
- NOT A is TRUE if A is FALSE.

Here, both A and B represent any logic proposition or assertion that has a value TRUE or FALSE.

The two problem statements above can now be rephrased as follows:

- Take_umbrella = TRUE IF (raining = TRUE) OR (rain_forecast = TRUE)
- System_on = TRUE IF (office hours = TRUE) AND (temperature > 25°C).

Each original problem statement has now been rephrased to include a form of **logic expression**. The format of each expression here does not follow any formally defined convention but the structure does allow the underlying logic to be understood. In general, a logic expression consists of logic propositions combined using Boolean operators. The expression may be included in an equation with a defined output.

Any logic expression can be constructed using only the Boolean operators AND, OR and NOT but it is often convenient to use other operators. Here are the definitions for the three other operators that you should be familiar with:

- A NAND B is TRUE if A is FALSE or B is FALSE
- A NOR B is TRUE if A is FALSE and B is FALSE
- A XOR B is TRUE if A is TRUE or B is true but not both of them.

WORKED EXAMPLE 4.01

Constructing a logic expression from a problem statement

Consider the following problem statement.

A shopkeeper orders a delivery of goods at the end of each month. However, if the stock of a particular item falls to the re-order level before the end of the month, a delivery is ordered immediately. Also, if a regular customer orders a large amount of goods, a delivery is ordered immediately.

We need to identify the conditions in the statement that can have true or false values. These can be underlined:

A shopkeeper orders a delivery of goods at the end of each month. However, if the stock of a particular item falls to the re-order level before the end of the month a delivery is ordered immediately. Also, if a regular customer orders a large amount of goods a delivery is ordered immediately.

The conditions can now be collected together in one logic expression:

End_of_month OR re-order_level_reached OR (regular_customer AND large_amount)

To simplify this we change each condition into a symbol.

- Let A represent End_of_month.
- Let B represent re-order_level_reached.

- Let C represent regular customer.
- Let D represent large_amount.

The logic expression can now be written in an equation using X to represent ‘a delivery is ordered’:

X = A OR B OR (C AND D)

TASK 4.01

Convert the conditions in the following problem statement into a simple logic expression:

A document can only be copied if it is not covered by copyright or if there is copyright and permission has been obtained.

4.03 Truth tables

The truth table is a simple but powerful technique for representing any logic expression or for describing the possible outputs from a logic circuit.

A truth table is presented by making use of the convention that TRUE can be represented as 1 and FALSE can be represented as 0. The simplest use of a truth table is to represent the logic associated with a Boolean operator.

As an example, let us consider the AND operator. The labelling of the truth table follows the convention that the initially defined values are represented by A and B and the value obtained from the simple expression using the AND operator is represented by X. In other words, we write the truth table for X = A AND B. Remembering that AND only returns TRUE if both A and B are TRUE we expect a truth table with only one instance of X having the value 1. The truth table is shown in Table 4.01.

The truth table has four rows corresponding to the four combinations of the truth values for A and B. Three of these lead to a 0 in the X column as expected.

A	B	X
0	0	0
0	1	0
1	0	0
1	1	1

Table 4.01 The truth table for the AND operator

TIP

When constructing a truth table make sure that the left-hand columns for the input values are written as though they were increasing binary values.

TASK 4.02

Without looking further on in the chapter, construct the truth table for the OR operator.

4.04 Logic circuits and logic gates

The digital circuits that constitute the inner workings of a computer system operate as logic circuits where each individual part of the circuit is either in an ‘on’ state, corresponding to a 1, or in an off state, corresponding to a 0. A logic circuit comprises component parts called logic gates. Each different **logic gate** has an operation that matches a Boolean operator.

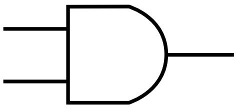
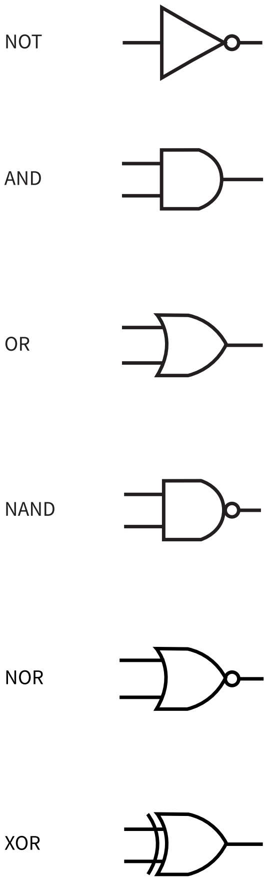


Figure 4.01 The symbol for the AND logic gate

When drawing a circuit, standard symbols are used for the logic gates. As an example, the symbol shown in Figure 4.01 represents an AND gate.

The first point to note here is that the shape of the symbol tells us the type of gate. The second point is that the inputs are shown on the left-hand side and the output is shown on the right-hand side. In general, the number of inputs is not limited to two. We will only consider circuits where the number of inputs is two or fewer.

Figure 4.02 shows the logic gate symbols and the associated truth tables for each of the six Boolean operators introduced in Section 4.02.



A	X
0	1
1	0

A	B	X
0	0	0
0	1	0
1	0	0
1	1	1

A	B	X
0	0	0
0	1	1
1	0	1
1	1	1

A	B	X
0	0	1
0	1	1
1	0	1
1	1	0

A	B	X
0	0	1
0	1	0
1	0	0
1	1	0

A	B	X
0	0	0
0	1	1
1	0	1
1	1	0

Figure 4.02 Logic gate symbols and their associated truth tables

There are two other points to note here. The NOT gate is a special case, having only one input. The second point is that a NAND gate is a combination of a AND gate followed by a NOT gate, and a NOR gate is a combination of an OR gate followed by a NOT gate. NAND and NOR gates produce a complementary output to the AND and OR gates.

TASK 4.03

Draw a circuit where A and B are input to an AND gate, from which the output is carried to a NOT gate, from which there is an output X. Show that this has the same outcome as having one NAND gate.

Extension Question 4.01

Could the same outcome be produced by positioning a NOT gate before the AND gate?

TIP

You need to remember the symbol for each of these gates. A good start here is to remember that AN**D** has the proper **D** symbol and OR has the curvy one.

You also need to remember the definitions for the gates so that you can construct the corresponding truth table for each gate.

Question 4.01

Can you recall from memory the symbols and definitions of the six logic gates introduced in this chapter?

WORKED EXAMPLE 4.02

Constructing a logic circuit from a problem statement or logic expression

You need to be able to construct a logic circuit from either a problem statement or from a logic expression. If you are given a problem statement, the best approach is to first convert it to a logic expression.

Consider the following problem statement: A bank offers a special lending rate to customers subject to certain conditions. To qualify, a customer must satisfy certain criteria.

- The customer has been with the bank for two years.
- Two of the following conditions must also apply:
  - the customer is married
  - the customer is aged 25 years or older
  - the customer's parents are customers of the bank.

To convert this statement to a logic expression using symbols we can choose:

- let A represent an account held for two years
- let B represent that the customer is married
- let C represent that the customer is aged 25 years or older
- let D represent that the customer's parents have an account.

The logic expression can then be written as:

A AND (((B AND C) OR (B AND D)) OR (C AND D))

This could alternatively be presented with an outcome:

Special_rate IF A AND (((B AND C) OR (B AND D)) OR (C AND D))

alternatively as

X = A AND (((B AND C) OR (B AND D)) OR (C AND D))

Note the use of brackets to ensure that the meaning is clear. You may think that not all of the brackets are needed. In this example, an extra pair has been included to guide the construction of the circuit where only two inputs are allowed for any of the gates.

From this, we can see that the logic circuit corresponding to this logic expression derived from the original problem statement could be constructed using four AND gates and two OR gates as shown in Figure 4.03.

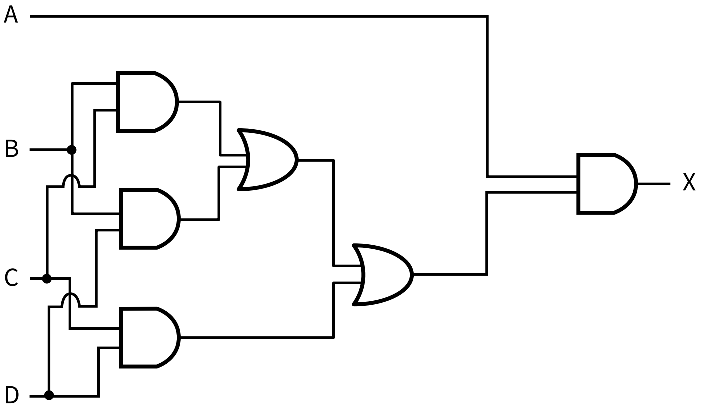


Figure 4.03 A logic circuit constructed from a problem statement

WORKED EXAMPLE 4.03

Constructing a truth table from a logic expression or logic circuit

You also need to be able to construct a truth table from either a logic expression or a logic circuit. We might have continued with the problem in Worked Example 4.02 but four inputs will lead to 16 rows in the truth table. Instead, we consider a slightly simpler problem with only three inputs and therefore only eight rows in the truth table. We will start with the circuit shown in Figure 4.04.

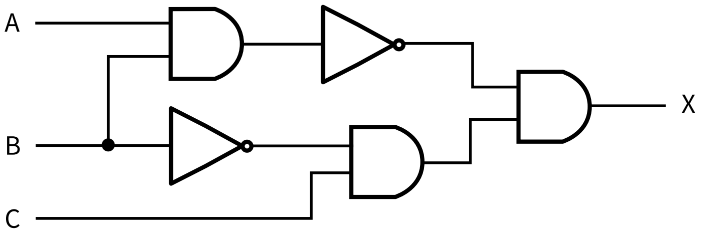


Figure 4.04 A circuit with three inputs for conversion to a truth table

Table 4.02 shows how the truth table needs to be set up initially. There are two points to note here. The first is that you must take care to include all of the eight different possible combinations of the input values. The second point is that for such a circuit it is not sensible to try to work out the outputs directly from the input values. Instead a systematic approach should be used. This involves identifying intermediate points in the circuit and recording the values at each of them in the columns headed ‘Workspace’ in Table 4.02.

Inputs			**Workspace**				**Output**
A	**B**	**C**					**X**
0	0	0					
0	0	1					
0	1	0					
0	1	1					
1	0	0					
1	0	1					
1	1	0					

1	1	1					

Table 4.02 The initial empty truth table

Figure 4.05 shows the same circuit but with four intermediate points labelled M, N, P and Q identified. Each one has been inserted on the output side of a logic gate.

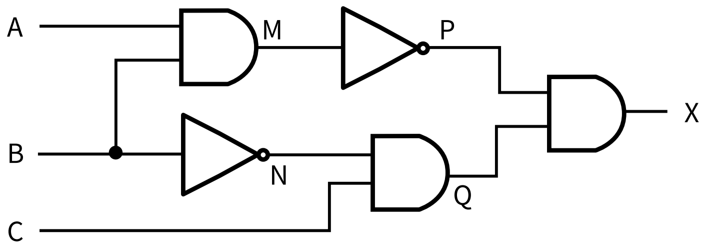


Figure 4.05 The circuit in Figure 4.04 with intermediate points identified

Now you need to work systematically through the intermediate points. You start by filling in the columns for M and N. Then you fill in the columns for P and Q which feed into the final AND gate. The final truth table is shown as Table 4.03. The circuit has two combinations of inputs that lead to a TRUE output from the circuit.

The columns containing the intermediate values (the workspace) could be deleted at this stage.

Inputs			**Workspace**				**Output**
A	**B**	**C**	**M**	**N**	**P**	**Q**	**X**
0	0	0	0	1	1	0	0
0	0	1	0	1	1	1	1
0	1	0	0	0	1	0	0
0	1	1	0	0	1	0	0
1	0	0	0	1	1	0	0
1	0	1	0	1	1	1	1
1	1	0	1	0	0	0	0
1	1	1	1	0	0	0	0

Table 4.03 The truth table for the circuit shown in Figure 4.05

One final point to make here is that you may be able to check part of your final solution by looking at just part of the circuit. For this example, if you look at the circuit you will see that the path from input C to the output passes through two AND gates. It follows, therefore, that for all combinations with C having value 0 the output must be 0. Therefore, in order to check your final solution you only need to examine the other four combinations of input values where C has value 1.

If a logic circuit is to be constructed from a truth table, the first stage is to create a logic expression. To do this only the rows producing a 1 output are used. Consider the truth table shown in Table 4.04. There are three rows producing a 1 output. Each of these produces a logic expression with AND operators. These three logic expressions are then combined with OR operators.

Inputs			**Output**
A	**B**	**C**	**X**
0	0	0	0
0	0	1	1
0	1	0	0

0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	0

Table 4.04 A truth table to be converted to a logic circuit

The three rows that produce a 1 output have the following values for the inputs:

A = 0, B = 0 and C = 1

A = 0, B = 1 and C = 1

A = 1, B = 0 and C = 0

Each one can be converted to a logical expression:

NOT A AND NOT B AND C

NOT A AND B AND C

A AND NOT B AND NOT C

The combination of the three individual expressions produces the following:

NOT A AND NOT B AND C

OR

NOT A AND B AND C

OR

A AND NOT B AND NOT C

This could be used to create a logic circuit, but the circuit would be quite complex. In Chapter 19 methods will be discussed that allow the simplest possible circuit to be constructed for a given logic problem.

If a logic expression is to be constructed from a logic circuit the first step is to construct a truth table from the circuit. Then the above method can be applied to this truth table.

TASK 4.04

An oven has a number of components that should all be working properly. For each component there is a signalling mechanism that informs a management system either if all is well, or if there is a problem. Table 4.05 summarises the signal values that record the status for each component.

Signal	Value	Component condition
A	0	Fan not working
	1	Fan working properly
B	0	Internal light not working
	1	Internal light working properly
C	0	Thermometer reading too high
	1	Thermometer reading in range

Table 4.05 Signals from the oven components

If the thermometer reading is in range but either or both the fan and light are not working, the management system has to output a signal to activate a warning light on the control panel. Draw a logic circuit for this fault condition.

Reflection Point:

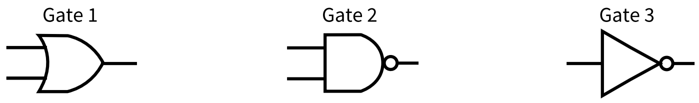
Looking back over the chapter content, what would you say is the central concept in the subject matter?

Summary

- A logic scenario can be described by a problem statement or a logic expression.
- A logic expression comprises logic propositions and Boolean operators.
- Logic circuits are constructed from logic gates.
- The operation of a logic gate matches that of a Boolean operator.
- The outcome of a logic expression or a logic circuit can be expressed as a truth table.
- A logic expression can be created from a truth table using the rows that provide a 1 output.

Exam-style Questions

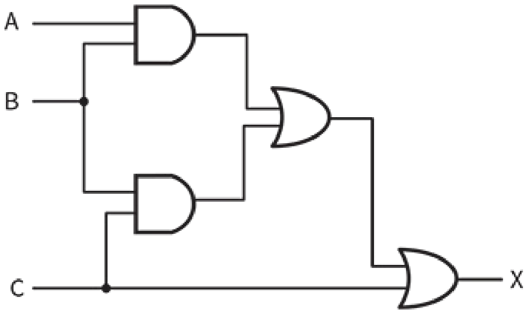
1 **a** The following are the symbols for three different logic gates.



i Identify each of the logic gates. [3]

ii Sketch the truth table for either Gate 1 or Gate 2. [2]

b Consider the following circuit:



i Complete the truth table for the circuit using the following template:

Inputs			**Workspace**			**Output**
A	**B**	**C**				**X**
0	0	0				
0	0	1				
0	1	0				
0	1	1				
1	0	0				
1	0	1				
1	1	0				
1	1	1				

[8]

ii There is an element of redundancy in this diagram. Explain what the problem is. [2]

2 **a** The definition of the NAND gate can be expressed as:

A NAND B is TRUE if A is FALSE or B is FALSE

Sketch the truth table for a NAND gate. [2]

b Consider the following statement:

In a competition, two teams play two matches against each other. One of the teams is declared the winner if one of the following results occurs:

- The team wins both matches.
- The team wins one match and loses the other but has the highest total score.

i Identify the **three** logic propositions in this statement. [3]

ii By assigning the symbols A, B and C to these three propositions give the outcome of the competition as a logic expression. [3]

iii Sketch a logic circuit to match this logic expression. [4]

3 A domestic heating system has a hot water tank and a number of radiators. There is a computerised management system which receives signals. These signals indicate whether or not the conditions for components are as they should be. The following table summarises the signals received:

Signal	Value	Component condition
A	0	Water flow in the radiators is too low
	1	Water flow in the radiators is within limits
B	0	Hot water tank temperature too high
	1	Hot water tank temperature within limits
C	0	Water level in hot water tank too low
	1	Water level in hot water tank within limits

a Consider the following fault condition. The water level in the hot water tank is too low and the temperature in the hot water tank is too high. The management system must output a signal to switch off the system.

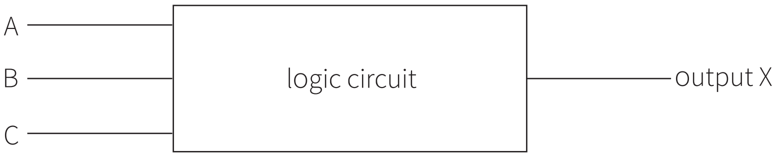
i Sketch a truth table for this fault condition including the A, B and C signals. [4]

ii Sketch the circuit diagram for this fault condition to match this truth table. [5]

b Consider the fault condition where the hot water tank temperature is within limits but the water flow in the radiators is too low and the water level in the hot water tank is too low. Sketch the circuit diagram for this fault condition which requires the management system to output a signal to increase water pressure. [5]

4 **a** Three digital sensors A, B and C are used to monitor a process. The outputs from the sensors are used as the inputs to a logic circuit.

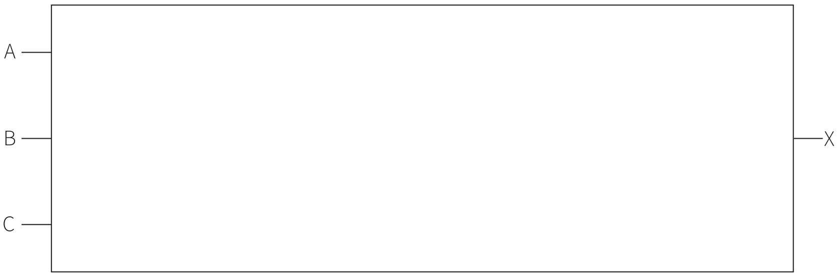
A signal, X, is output from the logic circuit:



Output, X, has a value of 1 if either of the following two conditions occur:

- sensor A outputs the value 1 OR sensor B outputs the value 0
- sensor B outputs the value 1 AND sensor C outputs the value 0

Draw a logic circuit to represent these conditions.



[5]

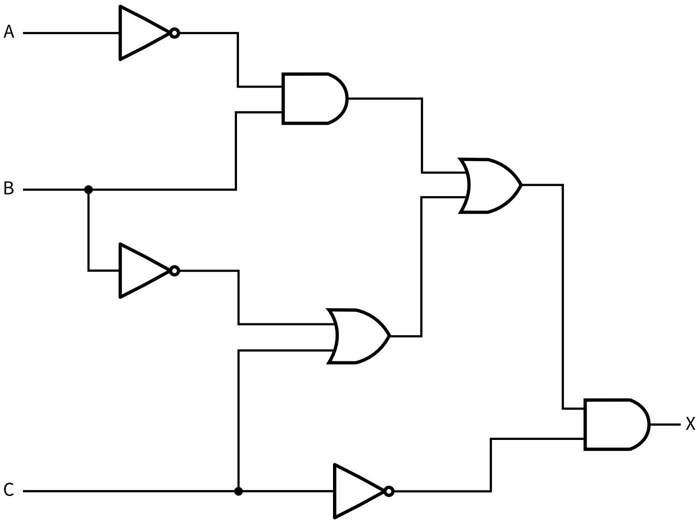
b Complete the truth table for the logic circuit described in **part (a)**.

A	B	C	Workspace	X
0	0	0		
0	0	1		
0	1	0		
0	1	1		
1	0	0		

1	0	1		
1	1	0		
1	1	1		

[4]

c Write a logic statement that describes the following logic circuit.



[3]

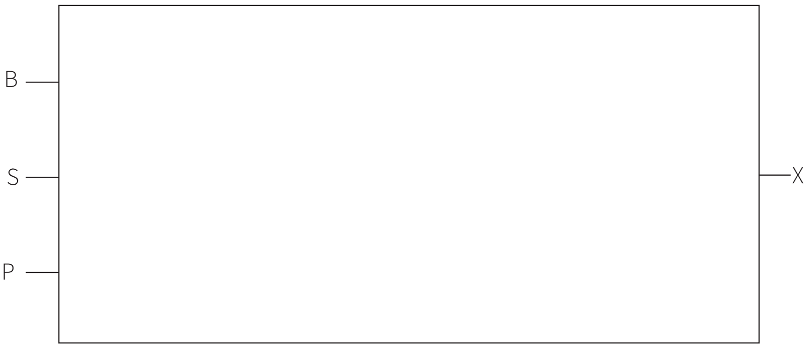
Cambridge International AS & A level Computer Science 9608 paper 13 Q6 June 2015

5 **a** A student writes the following logic expression:

X is 1 IF (B is NOT 1 AND S is NOT 1) OR (P is NOT 1 AND S is 1)

Draw a logic circuit to represent this logic expression.

Do **not** attempt to simplify the logic expression.



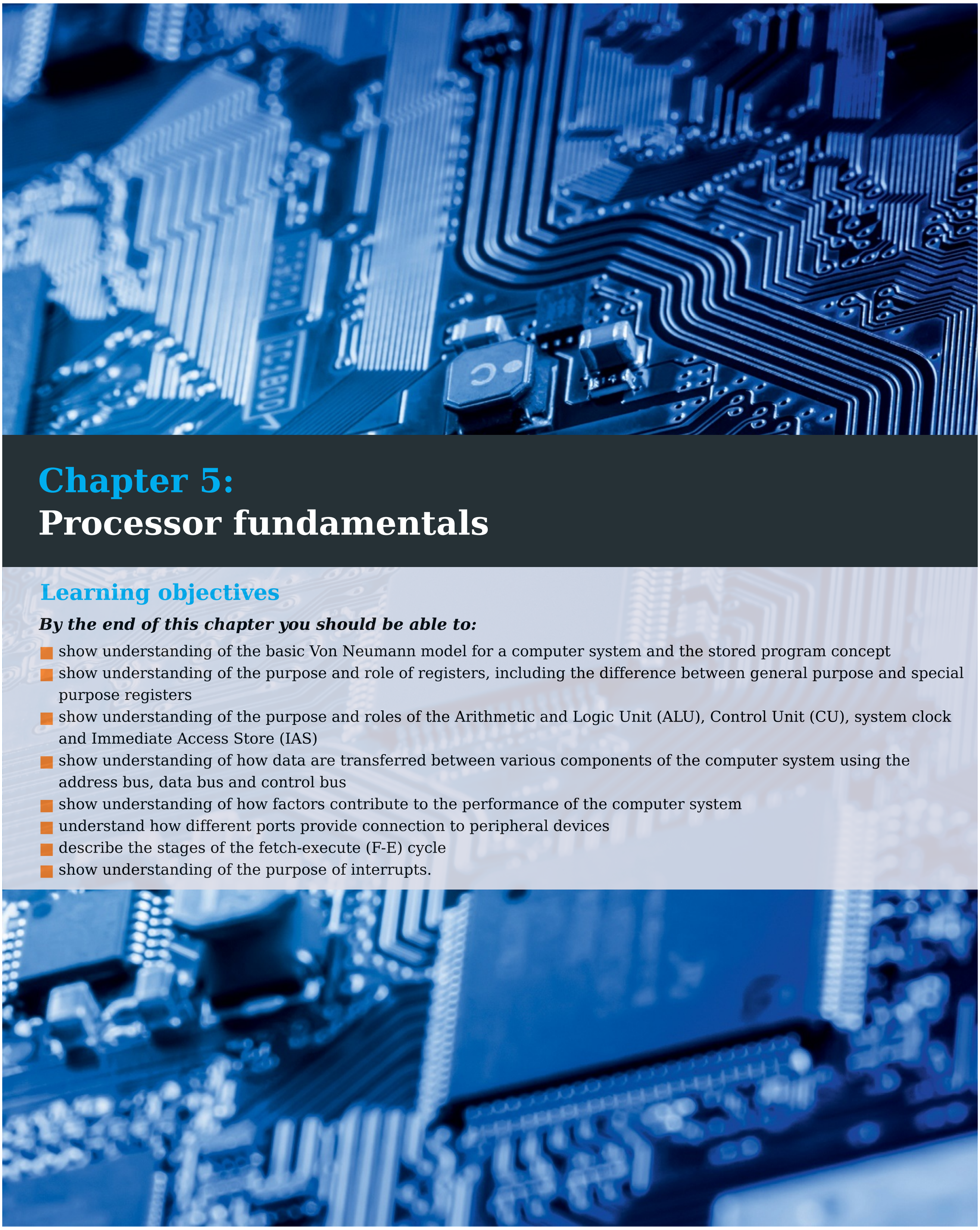
[6]

b Complete the truth table for the logic expression given in **part (a)**.

B	**S**	**P**	**Workspace**	**X**
0	0	0		
0	0	1		
0	1	0		
0	1	1		
1	0	0		
1	0	1		
1	1	0		
1	1	1		

[4]

Cambridge international AS & A Level Computer Science 9608 paper 12 Q1 November 2016



Chapter 5:
Processor fundamentals

Learning objectives

By the end of this chapter you should be able to:

- show understanding of the basic Von Neumann model for a computer system and the stored program concept
- show understanding of the purpose and role of registers, including the difference between general purpose and special purpose registers
- show understanding of the purpose and roles of the Arithmetic and Logic Unit (ALU), Control Unit (CU), system clock and Immediate Access Store (IAS)
- show understanding of how data are transferred between various components of the computer system using the address bus, data bus and control bus
- show understanding of how factors contribute to the performance of the computer system
- understand how different ports provide connection to peripheral devices
- describe the stages of the fetch-execute (F-E) cycle
- show understanding of the purpose of interrupts.

5.01 The von Neumann model of a computer system

John von Neumann was the first person to describe the basic principles of a computer system and its architecture in a publication.

The model von Neumann described has the following basic features.

- There is a processor - the central processing unit (CPU).
- The processor has direct access to memory.
- The memory contains a ‘stored program’ (which can be replaced by another at any time) and the data required by the program.
- The stored program consists of individual instructions.
- The processor executes instructions sequentially.

5.02 Central processing unit (CPU) architecture

In order to understand how the von Neumann model could be put into practice in a real computer system, we need first to identify the individual hardware components of a CPU and define their functions. Let's consider a system that has the minimum number of components needed. Figure 5.01 gives a simplified schematic diagram of a processor.

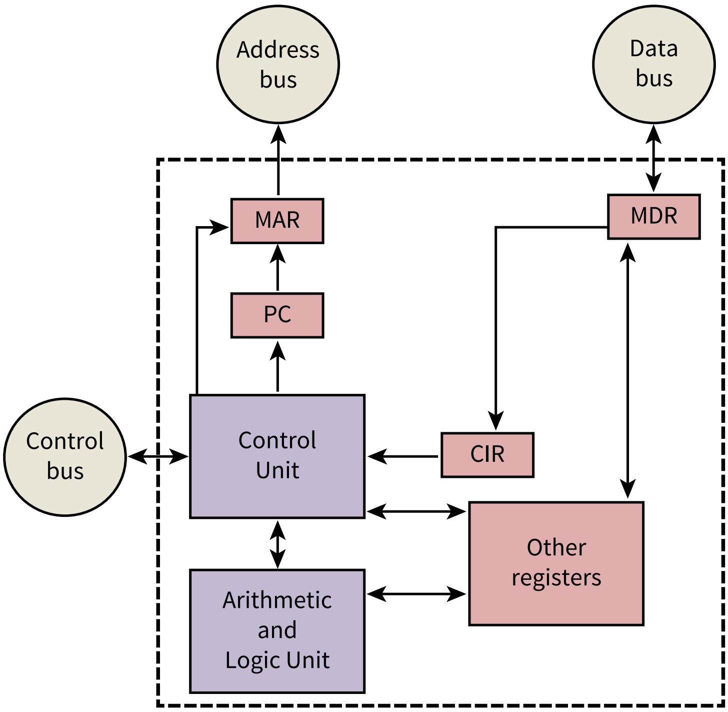


Figure 5.01 A schematic diagram of the architecture of a simple CPU

The dotted outline shows the boundary of the processor. The logical arrangement of some of the processor components is indicated. The arrows show possible directions of flow of data. As the following discussion will show, the data for some of the arrows is actually an address or an instruction. However, in general, data might be an address, an instruction or a value.

The active components of the CPU

The two components of the CPU that have an active role in its operation are the arithmetic and logic unit (ALU) (or Arithmetic Logic Unit) and the control unit. As its name implies, the ALU is responsible for the arithmetic or logic processing requirements of the instructions in a running program. The functions of the control unit are more diverse. One aspect is controlling the flow of data throughout the processor and the rest of the whole computer system. Another is ensuring that program instructions are handled correctly. A vital part of the control unit is a clock that is used by the unit to synchronise processes. Strictly speaking there are two clocks. The first is an **internal clock** that controls the cycles of activity within the processor. The other is the **system clock** that controls activities outside the processor. The CPU will have a defined frequency for its clock cycle, which is usually referred to as the clock speed. The frequency defines the minimum period of time that separates successive activities within the system.

Registers

The other components of the CPU are the registers. These are storage components which, because they are placed very close to the ALU, allow very short access times. Each register has limited storage capacity, typically 16, 32 or 64 bits. A register is either general purpose or special purpose. If there is only one general-purpose register it is referred to as the **Accumulator**. Here and in Chapter 6, we assume that the processor has just this one general-purpose register. The Accumulator is used to store a single value at any one time. A value is stored in the Accumulator that is to be used by the ALU for the execution of an instruction. The ALU can then store a different value in the Accumulator after the execution of the instruction.

Figure 5.01 shows some of the special-purpose registers as individual components. The box labelled ‘other registers’ can be considered to comprise the Accumulator plus the special-purpose registers not identified individually. The full names of the special-purpose registers included in the simple CPU that we are considering are given in Table 5.01 with a brief description of their function.

Register name	Abbreviation	Register’s function
Current instruction register	CIR	Stores the current instruction while it is being decoded and executed
Index register	IX	Stores a value; only used for indexed addressing
Memory address register	MAR	Stores the address of a memory location or an I/O component which is about to have a value read from or written to
Memory data register (memory buffer register)	MDR (MBR)	Stores data that has just been read from memory or is just about to be written to memory
Program counter	PC	Stores the address of where the next instruction is to be read from
Status register	SR	Contains bits that are either set or cleared which can be referenced individually

Table 5.01 Registers in a simple CPU

There are three important points to remember. The first is that the MDR must act as a buffer. This is because transfers of data inside the processor take place much more quickly than transfers outside the processor. The second point to note is that the index register (IX) can be abbreviated as IR but in some sources the current instruction register (CIR) is abbreviated as ‘IR’. This is a potential cause of confusion. In this book, the index register is always IX and the current instruction register is CIR. Finally, there is also possible confusion if the abbreviation PC is used. This will only be used in this book when register transfer notation is being used, as you will see later in the chapter. Everywhere else, a PC is a computer.

For all of the special-purpose registers, except for the status register, the contents represent one value. For the status register each individual bit is used as a logical flag. The bit is set to 1 if a condition is detected. Examples are the carry flag, the negative flag and the overflow flag.

Chapter 6 (Section 6.07) contains some examples of the use of the accumulator and the status register.

5.03 The system bus

A bus is a parallel transmission component with each separate wire carrying a single bit. It is important not to describe a bus as a storage device. A bus does not hold data. Instead it is a mechanism for data to be transferred from one system component to another.

There will be buses inside the CPU. These are not considered here. The system bus connects the CPU to the memory and to the I/O system. In the simple computer system described in this chapter there will be a system bus that comprises three distinct components: the address bus, the data bus and the control bus. The schematic diagram of the CPU in Figure 5.01 shows the logical connection between each bus and a CPU component. The address bus is connected to the MAR; the data bus to the MDR; and the control bus to the control unit. The system bus allows data flow between the CPU, the memory and input or output (I/O) devices as shown in the schematic diagram in Figure 5.02.

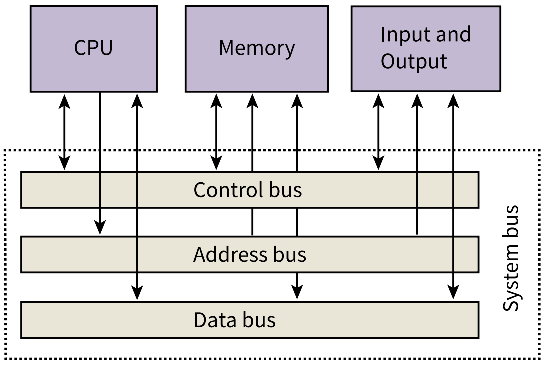


Figure 5.02 A schematic diagram of the system bus

The address bus

The sole function of the **address bus** is to carry an address. This address is loaded on to the bus from the MAR as and when directed by the control unit. The address specifies a location in memory or an I/O component which is due to receive data or from which data is to be read. The address bus is a 'one-way street'. It can only be used to send an address to a memory controller or an I/O controller. It cannot be used to carry an address back to the CPU.

The data bus

The function of the **data bus** is to carry data. This might be an instruction, an address or a value. As can be seen from Figure 5.02, the data bus is two-way (bidirectional): it might be carrying data from the CPU to the memory or carrying data to the CPU. However, another option is to carry data to or from an I/O device. The diagram does not make clear whether, for instance, data coming from an input device is carried first to the CPU or directly to the memory. There is a good reason for this. Some computer systems will only allow input to the CPU before the data can be stored in memory. Other systems will allow direct transfer to memory.

The control bus

The control bus is another bidirectional bus which transmits a signal from the control unit to any other system component or transmits a signal to the control unit. There is no need for extended width, so the control bus typically has just eight wires. A major use of the control bus is to carry timing signals. As described in Section 5.02, the system clock in the control unit defines the clock cycle for the computer system. The control bus carries timing signals at time intervals dictated by the clock cycle. This ensures that the time that one component transmits data is synchronised with the time that another component reads it.

5.04 Factors contributing to system performance

The processor clock speed is a very important factor governing the processing speed of the system. This is because one clock cycle defines the shortest possible time that any action can take. Actually, none of the components outside of the processor can work anywhere near as fast as the processor can. The components that are directly addressable by the processor, which can be referred to as the immediate access store (IAS), can only accept data from or provide data to the processor at speeds much slower than the processor speed.

Because of this problem modern processors are far more complex than the simple example that has been discussed in this chapter. One example of this complexity is that the CPU chip or integrated circuit will be multi-core. Each core is a separate processor. Performance improves with increasing number of cores. A further factor is the use of cache memory which was briefly discussed in Chapter 3 (Sections 3.01 and 3.03). Cache memory is the fastest component of the IAS. Performance improves with increased storage size for the cache and with increased rate of access. Fastest access is obtained by having all or part of the cache on the CPU chip.

Before considering other factors, it is useful to introduce the concept of a **word**. A word consists of a number of bytes and for any system the word length is defined. The significance of the word length is that it defines a grouping that the system can handle as one unit. The word length might be stated as a number of bytes or as a number of bits. Typical word lengths are 16, 32 or 64 bits; that is, 2, 4 or 8 bytes, respectively. The word length will influence the system architecture design in regard to the capacity of the components. For example, it is usual for the size of registers to match the word length. Word length also has to be considered when making decisions about bus widths.

For the address bus, the bus width defines the number of bits in the address's binary code. In a very simple computer system the bus width might be 16 bits, allowing 65 536 memory locations to be directly addressed. Such a memory size would, of course, be totally inadequate for a modern computer system. Even doubling the address bus width to 32 bits would only allow direct addressing of a little over four billion addresses. As a result, special techniques are used when the storage capacity of the memory is too large for direct addressing. Their use affects system performance.

Bus width is again an important factor in considering how the data bus is used. For a given computer system, the data bus width is ideally the same as the word length. If this is not possible, the bus width can be half the word length so that a full word can be transmitted by two consecutive data transfers. Clearly the performance of the system is affected if the latter case applies.

Extension Question 5.01

In an advertisement for a laptop computer, the system is described as 4 GB, 1 TB, 1.7 GHz.

a Which three components are being referred to here?

b Calculate the minimum time period that could separate successive activities on this system.

Extension Question 5.02

Can you find out the bus widths used in the computer system you are using?

5.05 I/O ports

The schematic diagram in Figure 5.02 slightly misrepresents the system architecture because it looks as if the CPU, the memory and the I/O devices have similar access to the data and control buses. The reality is different. Each I/O device is connected to an interface called a port. Each port is connected to the I/O or device controller. This controller handles the interaction between the CPU and an I/O device. A port is described as 'internal' if the connected I/O device is an integral part of the computer system. An external port allows the computer user to connect a peripheral I/O device.

The Universal Serial Bus (USB)

In the early days of the PC, the process of connecting a peripheral was not something the ordinary user would try to do; it required technical expertise. The aim of the plug-and-play concept was to remove the need for technical knowledge so that any computer user could connect a peripheral and start using it straight away. The plug-and-play concept was only fully realised by the creation of the Universal Serial Bus (USB) standard. Nowadays anyone buying a new peripheral device will expect it to connect to a USB port. There is an alternative technology known as FireWire, but this is not so commonly used in computer systems.

TIP

Don't forget that the USB is a bus. A USB drive stores data and is connected to a USB port which allows data to be transmitted along the bus.

The following is some information about the USB standard.

- A hierarchy of connections is supported.
- The computer is at the root of this hierarchy and can handle 127 attached devices.
- Devices can be attached while the computer is switched on and are automatically configured for use.
- The standard has evolved, with USB 3.2 being the latest version.

Discussion Point:

Carry out an investigation into storage devices that could be connected as a peripheral to a PC using the USB port.

For two representative devices find out which specific USB technology is being used and what the potential data transfer speed is. How do these speeds compare with the speed of access of a hard drive installed inside the computer?

Specialised multimedia ports

Despite the widespread use of USB ports there are some peripheral devices that require a different port, one that is specialised for the type of device. Although computer systems come packaged with a monitor for screen display there is sometimes a requirement for a second screen to be used. The connection of the second screen can be through a Video Graphics Array (VGA) port. This provides high-resolution screen display which is suitable for most display requirements. However, if the screen is needed to display a video, the VGA port is not suitable because it does not transmit the audio component.

A High Definition Multimedia Interface (HDMI) port will provide a connection to a screen and allow the transmission of high-quality video including the audio component.

5.06 The fetch-execute (F-E) cycle

The full name for this is the fetch, decode and execute cycle. This is illustrated by the flowchart in Figure 5.03.

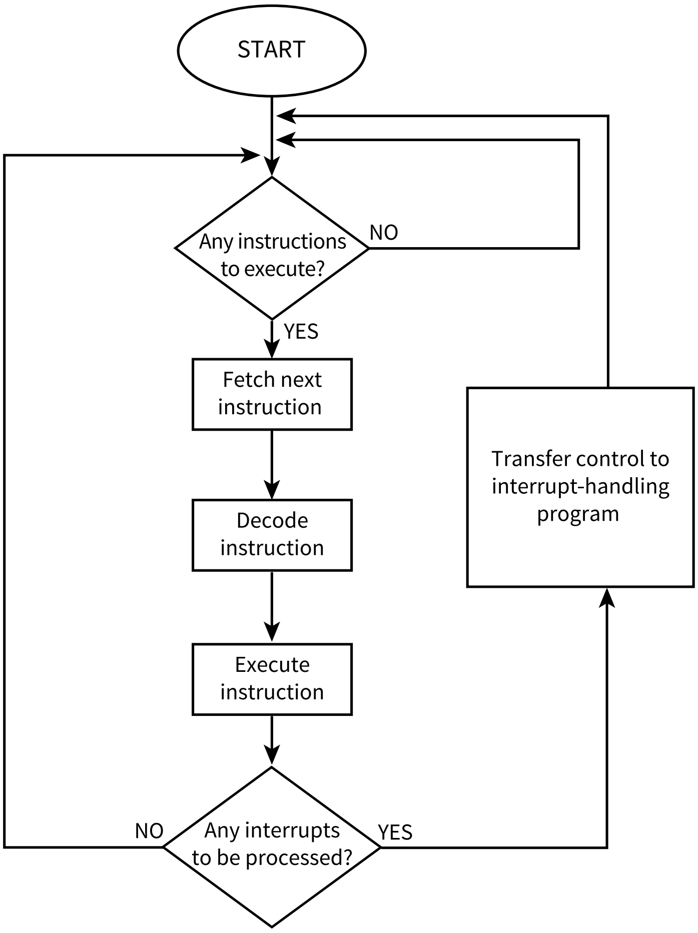

START
Any instructions to execute?
NO
YES
Fetch next instruction
Decode instruction
Execute instruction
Any interrupts to be processed?
NO
YES
Transfer control to interrupt-handling program

Figure 5.03 Flowchart for the fetch, decode and execute cycle

If we assume that a program is already running, then the program counter will already hold the address of the next instruction. In the fetch stage, the following steps will now happen.

1 This address in the program counter is transferred within the CPU to the MAR.

2 During the next clock cycle two things happen simultaneously:

   - the instruction held in the address pointed to by the MAR is fetched into the MDR
   - the address stored in the program counter is incremented.

3 The instruction stored in the MDR is transferred within the CPU to the CIR.

   There are two points to note here.

   - The clock cycle is the one controlled by the system clock which will have settings that allow one data transfer from memory to take place in the time defined for one cycle.
   - In the final step the program counter is incremented by 1. However, the instruction just loaded might be a jump instruction. In this case, the program counter contents will have to be updated in accordance with the jump condition. This can only happen after the instruction has been decoded.

In the decode stage, the instruction stored in the CIR is received as input by the circuitry within the control unit. Depending on the type of instruction, the control unit will send signals to the appropriate components so that the execute stage can begin. At this stage, the ALU will be activated if the instruction requires arithmetic or logic processing.

The description of the execute stage is given in Chapter 6, where a simple instruction set is introduced and discussed.

5.07 Register transfer notation

Operations involving registers can be described by register transfer notation. A simple example of this is a representation of the fetch stage of the fetch–execute cycle:

```
MAR ← [PC]
PC ← [PC] + 1; MDR ← [[MAR]]
CIR ← [MDR]
```

In register transfer notation the basic format for an individual data transfer is similar to that for variable assignment. The first item is the destination for the data. Here the appropriate abbreviation is used to identify the particular register. To the right of the arrow showing the transmission of data is the definition of this data. In this definition, the square brackets around a register abbreviation show that the content of the register is being moved. This movement might also include an arithmetical operation. When two data operations are placed on the same line separated by a semi-colon, this means that the two transfers take place simultaneously. The double pair of brackets around MAR on the second line needs careful interpretation. The content of the MAR is an address; it is the content of that address which is being transferred to the MDR.

5.08 Interrupt handling

There are many different reasons for an interrupt to be generated. Some examples are:

- a fatal error in a program
- a hardware fault
- a need for I/O processing to begin
- user interaction
- a timer signal.

Discussion Point:

Carry out an investigation into the different causes of an interrupt.

Interrupts are handled by a number of different mechanisms, but there are some clear overriding principles. Each different interrupt needs to be handled appropriately. Different interrupts might have different priorities. Therefore, the processor must have a means of identifying the type of interrupt. One way is to have an interrupt register in the CPU that works like the status register, with each individual bit operating as a flag for a specific type of interrupt.

As the flowchart in Figure 5.03 shows, the existence of an interrupt is only detected at the end of a fetch–execute cycle. This allows the current program to be interrupted and left in a defined state which can be returned to later. An interrupt is handled by the following steps.

- The contents of the program counter and any other registers are stored somewhere safe in memory.
- The appropriate interrupt handler or Interrupt Service Routine (ISR) program is initiated by loading its start address into the program counter.
- When the ISR program has been executed there is an immediate check to see if further interrupts need handling.
- Further interrupts are dealt with by repeated execution of the ISR program.
- If there are no further interrupts, the safely stored contents of the registers are restored to the CPU and the originally running program is resumed.

Reflection Point:

Have you worked out a method to remember all of the names and abbreviations for the special purpose registers?

Summary

- The von Neumann architecture for a computer system is based on the stored program concept.
- The CPU contains a control unit, an arithmetic and logic unit and registers.
- Registers can be special purpose or general purpose.
- The status register has individual bits acting as condition flags.
- The system bus contains the data, address and control buses.
- A universal serial bus (USB) port can be used to attach peripheral devices.
- Instructions are handled by the fetch–execute cycle.
- Register transfer notation is used to describe data transfers.
- If an interrupt is detected, control passes to an interrupt-handling routine.

Exam-style Questions

1 **a** A processor has just one general-purpose register. Give the name of this register. [1]

b The memory address register (MAR) is a special-purpose register. State:

i its function

ii the type of data stored in it

iii the register that supplies these data at the start of the fetch stage of the fetch–execute cycle. [3]

c The current instruction register (CIR) is another special-purpose register. State:

i its function

ii the type of data stored in it

iii the register that supplies this data at the end of the fetch stage of the fetch–execute cycle. [3]

d Explain **three** differences between the memory address register and the memory data register. [5]

2 The system bus comprises of three individual buses: the data bus, the address bus and the control bus.

a For each bus give a brief explanation of its use. [6]

b Each bus has a defined bus width.

i State what determines the width of a bus. [1]

ii Explain which bus will have the least width. [2]

iii Explain the effect of changing the address bus from a 32-bit bus to a 64-bit bus. [3]

3 The fetch stage of the fetch–decode–execute cycle can be represented by the following statements using register transfer notation:

```
MAR ← [PC]
PC ← [PC] + 1; MDR [[MAR]]
ICR ← [MDR]
```

a Explain the meaning of each statement. The explanation must include definitions of the following items: MAR, PC, [ ], , MDR, [[ ]], CIR. [10]

b Explain the use of the address bus and the data bus for two of the statements. [4]

4 **a** Name and describe **three** buses used in the von Neumann model. [6]

b The sequence of operations shows, in register transfer notation, the fetch stage of the fetch-execute cycle.

```
1  MAR ← [PC]
2  PC ← [PC] + 1
3  MDR ← [[MAR]]
4  CIR ← [MDR]
```

- `[register]` denotes contents of the specified register or memory location
- step 1 above is read as “the contents of the Program Counter are copied to the Memory Address Register”

i Describe what is happening at step 2. [1]

ii Describe what is happening at step 3. [1]

iii Describe what is happening at step 4. [1]

c Describe what happens to the registers when the following instruction is executed:

```
LDD 35
```

[2]

d **i** Explain what is meant by an interrupt. [2]

ii Explain the actions of the processor when an interrupt is detected. [4]

Cambridge International AS & A level Computer Science 9608 paper 13 Q5 June 2015

5 a Describe how special purpose registers are used in the fetch stage of the fetch-execute cycle. [4]

b Use the statements A, B, C and D to complete the description of how the fetch-execute cycle handles an interrupt.

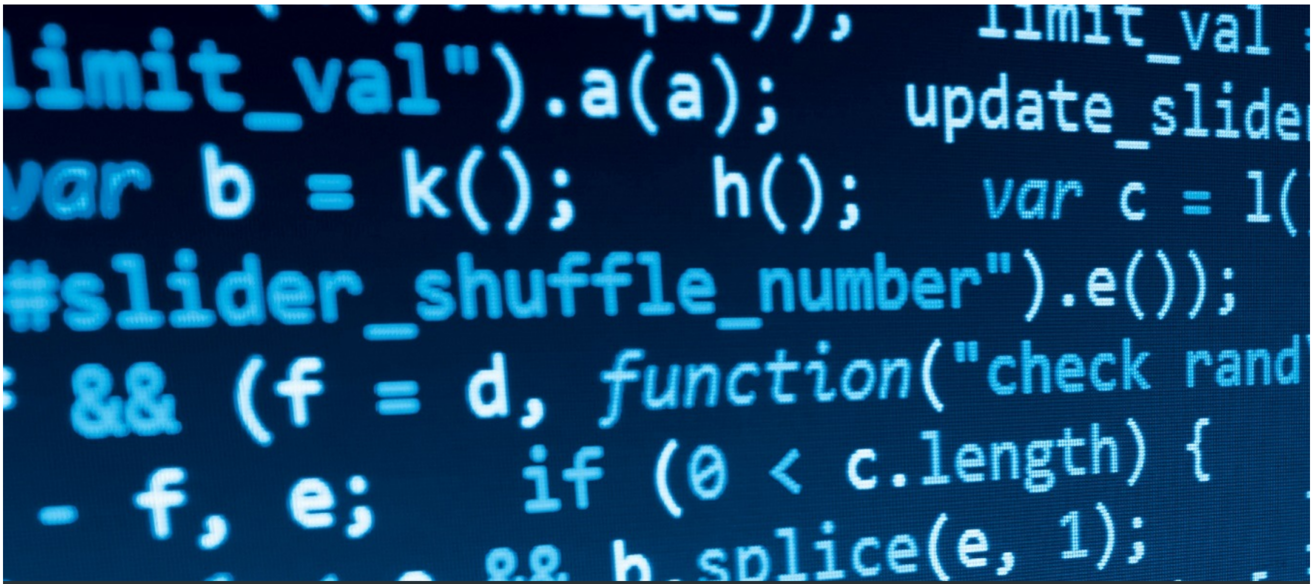
A	the address of the Interrupt Service Routine (ISR) is loaded to the Program Counter (PC).
B	the processor checks if there is an interrupt.
C	when the ISR completes, the processor restores the register contents.
D	the register contents are saved.

At the end of the cycle for the current instruction ....

If the interrupt flag is set, ..., ... and ....

The interrupted program continues its execution. [4]

Cambridge International AS & A level Computer Science 9608 paper 11 Q3 June 2016.



Chapter 6:
Assembly language programming

Learning objectives

By the end of this chapter you should be able to:

- show understanding of the relationship between assembly language and machine code
- describe the different stages of the assembly process for a two-pass assembler
- trace a given simple assembly language program
- show understanding that the set of instructions are grouped into instructions for:
    - data movement
    - input and output of data
    - arithmetic operations
    - unconditional and conditional jumps
    - comparisons
- show understanding of modes of addressing
- show understanding of and perform binary shifts.

6.01 Machine code instructions

We need to start with a few facts.

- The only language that the CPU recognises is machine code.
- Machine code consists of a sequence of instructions.
- An instruction contains an **opcode**.
- An instruction may not have an **operand** but up to three operands are possible.
- Different processors have different instruction sets associated with them.
- Different processors will have comparable instructions for the same operations, but the coding of the instructions will be different.

For a particular processor, the following must be defined for each individual **machine code instruction**:

- the total number of bits or bytes for the whole instruction
- the number of bits that define the opcode
- the number of operands that are defined in the remaining bits
- whether the opcode occupies the most significant or the least significant bits.

We will consider a simple system where there is either one or zero operands. This simple system will be assumed to have a 16-bit address bus width. Following on from the approach in Chapter 5 (Section 5.02), the system will have the accumulator as the only general purpose register.

The number of bits needed for the opcode depends on the number of different opcodes in the instruction set for the processor. The opcode can be structured with the first few bits defining the operation and the remaining bits associated with addressing. A sensible instruction format for our simple processor is shown in Figure 6.01.

	Opcode		Operand
Operation	**Address mode**	**Register addressing**	
4 bits	2 bits	2 bits	16 bits

Figure 6.01 A simple instruction format

This has an eight-bit opcode consisting of four bits for the operation, two bits for the address mode (discussed in Section 6.05) and the remaining two bits for addressing registers. This allows 16 different operations each with one of four addressing modes. This opcode will occupy the most significant bits in the instruction. Because in some circumstances the operand will be a memory address it is sensible to allocate 16 bits for it. This is in keeping with the 16-bit address bus.

When an instruction arrives in the CPU the control unit checks the opcode to see what action it defines. This first step in the decode stage of the fetch–execute cycle can be described using the register transfer notation which was introduced in Chapter 5 (Section 5.07). However, a slight amendment is needed to the format. The following shows the transfer of bits 16 to 23, which represent the opcode, from the current instruction register to the control unit:

```
CU ← [CIR(23:16)]
```

6.02 Assembly language

A programmer might wish to write a program where the actions taken by the processor are directly controlled. It is argued that this is the most efficient type of program. However, writing a substantial program as a sequence of machine code instructions would take a very long time and there would be inevitably lots of errors along the way. The solution for this type of programming is to use **assembly language**. As well as having a uniquely defined machine code language, each processor has its own assembly language.

The essence of assembly language is that for each machine code instruction there is an equivalent assembly language instruction which comprises:

- a mnemonic (a symbolic abbreviation) for the opcode
- a character representation for the operand.

If a program has been written in assembly language it has to be translated into machine code before it can be executed by the processor. The translation program is called an **assembler**.

Using an assembly language, the programmer has the advantage of the coding being easier to write than it would have been in machine code. In addition, the use of the assembler allows a programmer to include some special features in an assembly language program. Examples of some of these are:

- comments
- symbolic names for constants
- labels for addresses
- macros
- directives.

A macro is a sequence of instructions that is to be used more than once in a program. A **directive** is an instruction to the assembler as to how it should construct the final executable machine code. This might be to direct how memory should be used or to define files or procedures that will be used.

Discussion Point:

Although writing a program in assembly language is much easier than using machine code, many would argue that its use is no longer justified. Can you investigate the arguments for and against?

6.03 Symbolic, relative and absolute addressing

When considering how an assembler would convert an assembly language program into machine code it is necessary to understand the difference between symbolic, relative and absolute addressing. To explain these, we can consider a simple assembly language program which totals single numbers input at the keyboard. Table 6.01 shows the program as it would be written using symbolic addressing together with an explanation of each instruction.

Assembly language program using symbolic addressing	Explanation of each instruction
`IN`	A single number is input at the keyboard and its ASCII code is stored in the accumulator
`SUB #48`	This subtraction converts the ASCII code into the binary code for the number (see Task 6.01)
`STO MAX`	The number in the accumulator is stored at the address labelled MAX:
`LDM #0`	Loads zero into the accumulator
`STO TOTAL`	The zero in the accumulator is stored at the address labelled TOTAL:
`STO COUNT`	The zero in the accumulator is stored at the address labelled COUNT:
`STRTLP:IN`	A single number is input at the keyboard and its ASCII code is stored in the accumulator
`SUB #48`	This subtraction converts the ASCII code into the binary code for the number
`ADD TOTAL`	Adds the value at address labelled TOTAL: to the value in the accumulator and stores the sum in the accumulator
`STO TOTAL`	The number in the accumulator is stored at the address labelled TOTAL:
`LDD COUNT`	Loads the value stored at address COUNT: into the accumulator
`INC ACC`	Adds 1 to the value in the accumulator
`CMP MAX`	Compares the value in the accumulator with the value stored at address MAX:
`JPN STRTLP`	If the compared values are not equal the program jumps to the instruction labelled STRTLP:
`END`	The execution of the program has finished
`MAX:`	A labelled address where a value can be stored
`TOTAL:`	A labelled address where a value can be stored
`COUNT:`	A labelled address where a value can be stored

Table 6.01 An assembly program using symbolic addressing with explanations

The convention has been followed that a label is written with a following colon which is ignored when the label is referenced. Note how the code is dominated by the use of the accumulator.

TASK 6.01

Check the ASCII coding table to see why the subtraction in Table 6.01 works.

The use of symbolic addressing allows a programmer to write some assembly language code without having to bother about where the code will be stored in memory when the program is run. However, it is possible to write assembly language code where the symbolic addressing is replaced by either relative addressing or absolute addressing. Table 6.02 shows the simple code from Table 6.01 converted to use these alternative approaches.

Assembly language program using relative addressing	Assembly language program using absolute addressing	
(0) IN	(200)	IN
(1) SUB #48	(201)	SUB #48
(2) STO [BR] + 15	(202)	STO 215
(3) LDM #0	(203)	LDM #0
(4) STO [BR] + 16	(204)	STO 216
(5) STO [BR] + 17	(205)	STO 217
(6) IN	(206)	IN
(7) SUB #48	(207)	SUB #48
(8) ADD [BR] + 16	(208)	ADD 216
(9) STO [BR] + 16	(209)	STO 216
(10) LDD [BR] + 17	(210)	LDD 217
(11) INC ACC	(211)	INC ACC
(12) CMP [BR] + 15	(212)	CMP 215
(13) JPN [BR] + 7	(213)	JPN 207
(14) END	(214)	END
(15)	(215)	
(16)	(216)	
(17)	(217)	

Table 6.02 A simple assembly language program using relative and absolute addressing

For the relative addressing example, the assumption is that a special-function base register BR contains the base address. The contents of this register can then be used as indicated by [BR]. Note that there are no labels for the code. The left-hand column is just for illustration identifying the offset from the base address which is the address of the first instruction in the program.

For the absolute address example there are again no labels for the code. The left-hand column is again just for illustration but this time identifying actual memory addresses. This has been coded with the understanding that the first instruction in the program is to be stored at memory address 200.

6.04 The assembly process for a two-pass assembler

For any assembler there are a number of things that have to be done with the assembly language code before any translation can be done. Some examples are:

- removal of comments
- replacement of a macro name used in an instruction by the list of instructions that constitute the macro definition
- removal and storage of directives to be acted upon later.

A two-pass assembler is designed to handle programs written in the style of the one illustrated in Table 6.01. This program contains forward references. Some of the instructions have a symbolic address for the operand where the location of the address is not known at that stage of the program. A two-pass assembler is needed so that in the first pass the location of the addresses for forward references can be identified.

To achieve this during the first pass the assembler uses a symbol table. The code is read line by line. When a symbolic address is met for the first time its name is entered into the symbol table. Alongside the name a corresponding address has to be added as soon as that can be identified. Table 6.03 shows a possible format for the symbol table that would be created for the program shown in Table 6.01.

Symbol	Offset
MAX	+15
TOTAL	+16
COUNT	+17
STRTLP	+7

Table 6.03 A completed symbol table for the assembly language program in Table 6.01

Note that the assembler has to count the instructions as it reads the code. Then when it encounters a label it can enter the offset value into the symbol table. In this example the first entry made in the offset column is the +7 for STRPLP.

For the second pass the Assembler uses the symbol table and a lookup table that contains the binary code for each opcode. This table would have an entry for every opcode in the set defined for the processor. Table 6.04 shows entries only for the instructions used in the simple program we are using as an example. Note that the binary codes are just suggestions of codes that might be used.

Opcode mnemonic	Opcode binary
IN	0001 0000
SUB	0110 0001
STO	0100 0100
LDM	0010 0001
ADD	0100 0101
LDD	0010 0101
INC	0101 0101
CMP	1000 0100
JPN	1010 0100
END	1111 1111

Table 6.04 An opcode lookup table

Provided that no errors have been identified, the output from the second pass will be a machine code program. For our example, this code is shown in Table 6.05 along with the original assembly code for comparison.

Machine code		Assembly code	
Opcode	Operand		
0001 0000			`IN`
0110 0001 0000 0000 0011 0000			`SUB #48`
0100 0100 0000 0000 0000 1111			`STO MAX`
0010 0001 0000 0000 0000 0000			`LDM #0`
0100 0100 0000 0000 0001 0000			`STO TOTAL`
0100 0100 0000 0000 0001 0001			`STO COUNT`
0001 0000		`STRTLP:`	`IN`
0110 0001 0000 0000 0011 0000			`SUB #48`
0100 0101 0000 0000 0001 0000			`ADD TOTAL`
0100 0100 0000 0000 0001 0000			`STO TOTAL`
0010 0101 0000 0000 0001 0001			`LDD COUNT`
0101 0101			`INC ACC`
1000 0100 0000 0000 0000 1111			`CMP MAX`
1010 0100 0000 0000 0000 0110			`JPN STRTLP`
1111 1111			`END`
0000 0000		`MAX:`	
0000 0000		`TOTAL:`	
0000 0000		`COUNT:`	

Table 6.05 Machine code created from assembly code

Some points to note are as follows.

- Most of the instructions have an operand which is a 16-bit binary number.
- Usually this represents an address but for the `SUB` and `LDM` instructions the operand is used as a value.
- There is no operand for the `IN` and `END` instructions.
- The `INC` instruction is a special case. There is an operand in the assembly language code but this just identifies a register. In the machine code the register is identified within the opcode so no operand is needed.
- The machine code has been coded with the first instruction occupying address zero.
- This code is not executable in this form but it is valid output from the assembler.
- Changes will be needed for the addresses when the program is loaded into memory ready for it to be executed.
- Three memory locations following the program code have been allocated a value zero to ensure that they are available for use by the program when it is executed.

6.05 Addressing modes

When an instruction requires a value to be loaded into a register there are different ways of identifying the value. Each one is known as an **addressing mode**. In Section 6.01, it was stated that, for our simple processor, two bits of the opcode in a machine code instruction would be used to define the addressing mode. This allows four different modes which are described in Table 6.06.

Addressing mode	Use of the operand
Immediate	The operand is the value to be used in the instruction; `SUB #48` is an example.
Direct	The operand is the address which holds the value to be used in the instruction; `ADD TOTAL` is an example.
Indirect	The operand is an address that holds the address which has the value to be used in the instruction.
Indexed	The operand is an address to which must be added the value currently in the index register (IX) to get the address which holds the value to be used in the instruction.

Table 6.06 Addressing modes

For immediate addressing there are three options for defining the value:

- #48 specifies the denary value 48
- #B00110000 specifies the binary equivalent
- #&30 specifies the hexadecimal equivalent

6.06 Assembly language instructions

We continue to consider a simple processor with a limited instruction set. The examples described here do not correspond directly to those found in the assembly language for any specific processor. Individual instructions will have a match in more than one real-life set. The important point is that these examples are representative. In particular, there are examples of the most common categories of instruction.

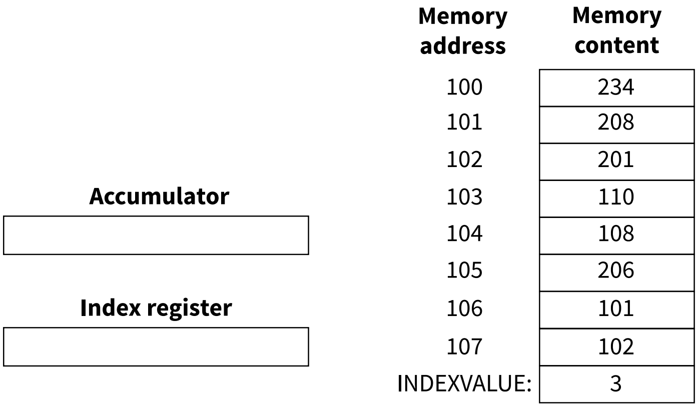
Data movement

These types of instruction can involve loading data into a register or storing data in memory. Table 6.07 contains a few examples of the format of the instructions with explanations.

Instruction opcode	Instruction operand	Explanation
`LDM`	`#n`	Immediate addressing. Load the number n to ACC.
`LDR`	`#n`	Immediate addressing. Load the number n to IX.
`LDD`	`<address>`	Direct addressing. Load the contents at the given address to ACC.
`LDI`	`<address>`	Indirect addressing. The address to be used is at the given address. Load the contents of this second address to ACC.
`LDX`	`<address>`	Indexed addressing. Form the address from <address> + the contents of the index register. Copy the contents of this calculated address to ACC.
`MOV`	`<register>`	Move the contents of the accumulator to the given register (IX).
`STO`	`<address>`	Store the contents of ACC at the given address.

Table 6.07 Some instruction formats for data movement

The important point to note is that the mnemonic defines the instruction type including which register is involved and, where appropriate, the addressing mode. It is important to read the mnemonic carefully! The instruction will have an actual address where `<address>` is shown, a register abbreviation where `<register>` is shown and a denary value for `n` where `#n` is shown. The explanations use ACC to indicate the accumulator. For explanations of `LDD`, `LDI` and `LDX`, refer back to Table 6.07.



Memory address	Memory content
100	234
101	208
102	201
103	110
104	108
105	206
106	101
107	102
INDEXVALUE:	3

Figure 6.02 Example of some data stored in memory

The following shows some examples of the effect of an instruction or a sequence of instructions based on the memory content shown in Figure 6.02.

LDD 103	the value 110 is loaded into the accumulator
LDI 106	the value 208 from address 101 is loaded into the accumulator
STO 106	the value 208 is stored in address 106

LDD INDEXVALUE	the value 3 is loaded into the accumulator
MOV IX	the value 3 from the accumulator is loaded into the index register
LDX 102	the value 206 from address 105 is loaded into the accumulator

Input and output

There are two instructions provided for input or output. In each case the instruction has only an opcode; there is no operand.

- The instruction with opcode `IN` is used to store in the ACC the ASCII value of a character typed at the keyboard.
- The instruction with opcode `OUT` is used to display on the screen the character for which the ASCII code is stored in the ACC.

Comparisons and jumps

A program might need an unconditional jump or might need a jump if a condition is met. In the second case, a compare instruction is executed first. Table 6.08 shows the format for these types of instruction.

Instruction opcode	Instruction operand	Explanation
JMP	<address>	Jump to the given address
CMP	<address>	Compare the contents of ACC with the contents of <address>
CMP	#n	Compare the contents of ACC with the number n
CMI	<address>	Indirect addressing. The address to be used is at the given address. Compare the contents of ACC with the contents of this second address
JPE	<address>	Following a compare instruction, jump to <address> if the compare was True
JPN	<address>	Following a compare instruction, jump to <address> if the compare was False

Table 6.08 Jump and compare instruction formats

Note that the comparison is restricted to asking if two values are equal.

The result of the comparison is recorded by a flag in the status register. The execution of the conditional jump instruction begins by checking whether or not the flag bit has been set. This jump instruction does not cause an immediate jump. This is because a new value has to be supplied to the program counter so that the next instruction is fetched from this newly specified address. The incrementing of the program counter that took place automatically when the instruction was fetched is overwritten.

Arithmetic operations

There are no instructions for general-purpose multiplication or division. General-purpose addition and subtraction are catered for. Table 6.09 contains the instruction formats used for arithmetic operations.

Instruction opcode	Instruction operand	Explanation
ADD	<address>	Add the contents of the given address to the ACC
ADD	#n	Add the denary number n to the ACC
SUB	<address>	Subtract the contents of the given address from the ACC
SUB	#n	Subtract the denary number n from the ACC

`INC`	`<register>`	Add 1 to the contents of the register (ACC or IX)
`DEC`	`<register>`	Subtract 1 from the contents of the register (ACC or IX)

Table 6.09 Instruction formats for arithmetic operations

Figure 6.03 shows a program to find out how many times 5 divides into 75.

The following should be noted concerning the program.

- The first three instructions initialise the count and the sum.
- The instruction in address 103 is the one that is returned to in each iteration of the loop; in the first iteration it is loading the value 0 into the accumulator when this value is already stored but this cannot be avoided.

Memory address	**Memory content**
100	`LDD 200`
101	`STO 202`
102	`STO 203`
103	`LDD 202`
104	`INC ACC`
105	`STO 202`
106	`LDD 203`
107	`ADD 201`
108	`STO 203`
109	`CMP 204`
110	`JPN 103`
111	`LDD 202`
112	`OUT`
113	`END`
200	`0`
201	`5`
202	
203	
204	`75`

Figure 6.03 A program to calculate the result of dividing 75 by 5

- The next three instructions are increasing the count by 1 and storing the new value.
- Instructions 106 to 108 add 5 to the sum.
- Instructions 109 and 110 check to see if the sum has reached 75 and if it has not the program begins the next iteration of the loop.
- Instructions 111 to 113 are only used when the sum has reached 75 which causes the value 15 stored for the count to be output.

Shift operations

There are two shift instructions available:

- `LSL    #n`

  where the bits in the accumulator are shifted logically n places to the left

- `LSR    #n`

  where the bits are shifted to the right.

In a **logical shift** no consideration is given as to what the binary code in the accumulator represents. Because a shift operation moves a bit from the accumulator into the carry bit in the status register this can be used to examine individual bits. For a left logical shift, the most significant bit is moved to the carry bit, the remaining bits are shifted left and a zero is entered for the least significant bit. For a right logical shift, it is the least significant bit that is moved to the carry bit and a zero is entered for the most significant bit.

If the accumulator content represents an unsigned integer, the left shift operation is a fast way to multiply by two. However, this only gives a correct result if the most significant bit is a zero. For an unsigned integer the right shift represents integer division by two. For example, consider:

00110001 (denary 49) gives if right shifted 00011000 (denary 24)

The remainder from the division can be found in the carry bit. Again, the division will not always give a correct result; continuing right shifts will eventually produce a zero for every bit. It should be apparent that a logical shift cannot be used for multiplication or division by two when a signed integer is stored. This is because the operation may produce a result where the sign of the number has changed.

As indicated earlier, only the two logical shifts are available for the simple processor considered here. However, in more complex processors there is likely to be a **cyclic shift** capability. Here a bit moves off one end into the carry bit then one step later moves in at the other end. All bit values in the original code are retained. Left and right **arithmetic shifts** are also likely to be available. These work in a similar way to logical shifts, but are provided for the multiplication or division of a signed integer by two. The sign bit is always retained following the shift.

Bitwise logic operation

The options for this are described in Table 6.10.

Instruction opcode	Instruction operand	Explanation
`AND`	`#Bn`	Bitwise AND operation of the contents of ACC with the binary number n
`AND`	`<address>`	Bitwise AND operation of the contents of ACC with the contents of <address>
`XOR`	`#Bn`	Bitwise XOR operation of the contents of ACC with the binary number n
`XOR`	`<address>`	Bitwise XOR operation of the contents of ACC with the contents of <address>
`OR`	`#Bn`	Bitwise OR operation of the contents of ACC with the binary number n
`OR`	`<address>`	Bitwise OR operation of the contents of ACC with the contents of <address>

Table 6.10 Bitwise logical operation instructions

The operand for a bitwise logic operation instruction is referred to as a mask because it can effectively cover some of the bits and only affect specific bits. Some examples of their use are given in Chapter 7 (Section 7.03).

6.07 Further consideration of assembly language instructions

Register transfer notation

Section 6.01 introduced an extension to register transfer notation. We can use this to describe the execution of an instruction. For example, the `LDD` instruction is described by:

```
ACC ← [[CIR(15:0)]]
```

The instruction is in the CIR and only the 16-bit address needs to be examined to identify the location of the data in memory. The contents of that location are transferred into the accumulator.

TASK 6.02

Use register transfer notation to describe the execution of an LDI instruction.

Computer arithmetic

In Chapter 1 (Section 1.03) we saw that computer arithmetic could lead to an incorrect answer if overflow occurred. In Chapter 5 (Section 5.02) we saw the possible uses of the Status Register. The following worked example illustrates how the values stored in the Status Register can identify a specific overflow condition.

The use of the following three flags is required:

- the carry flag, identified as C, which is set to 1 if there is a carry
- the negative flag, identified as N, which is set to 1 if a result is negative
- the overflow flag, identified as V, which is set to 1 if overflow is detected.

WORKED EXAMPLE 6.01

Using the status register during an arithmetic operation

1 Consider the addition of two positive values where the sum of the two produces an answer that is too large to be correctly identified with the limited number of bits used to represent the values. For example, Figure 6.04 shows what happens if we use an eight-bit binary integer representation and attempt to add denary 66 to denary 68.

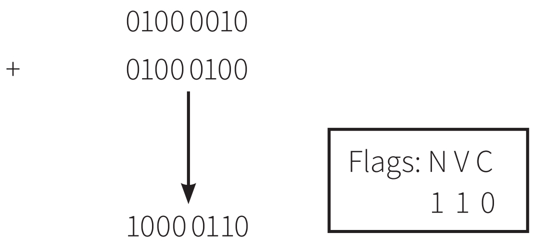


Figure 6.04 An attempted addition of denary 66 to denary 68

The answer produced is denary −122. Two positive numbers have been added to get a negative number. This impossibility is detected by the combination of the negative flag and the overflow flag being set to 1. The processor examines the flags, identifies the problem and generates an interrupt.

2 Consider using the same eight-bit binary integer representation but this time we add two negative numbers (−66 and −68 in denary). The result is shown in Figure 6.05.

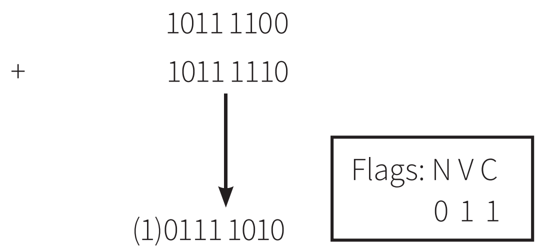


Figure 6.05 An attempted addition of denary −66 to denary −68

We get the answer +122. This impossibility is detected by the combination of the negative flag not being set and both the overflow and the carry flag being set to 1.

Extension Question 6.01

Carry out a comparable calculation for the addition in binary of −66 to +68. What do you think the processor should do with the carry bit?

Tracing an assembly language program

One way of checking to see if an assembly language program has errors is to carry out a dry (practice) run. The main feature of this will be to check how the contents of the accumulator change as the program runs. The following two worked examples illustrate the process.

WORKED EXAMPLE 6.02

Tracing an assembly language program

For this example the trace table needs a column for the accumulator, two for memory locations and one for the output.

The tracing is based on an initial user input of 15, a second input of 27 and a final input of 31.

The program is shown in Figure 6.06.

100	IN
101	STO 200
102	IN
103	STO 201
104	IN
105	ADD 200
106	STO 200
107	ADD 201
108	INC ACC
109	OUT
110	END

Figure 6.06 The assembly language program

The completed trace table is shown in Figure 6.07

Accumulator	Memory location 200	Memory location 201	Output
15			
	15		
27			
		27	
31			
46			
	46		

73			
74			
			74

Figure 6.07 The trace table showing the execution of the program

Note that in this presentation the decision has been made to use a new row in the trace table for each instruction in the program. This helps with checking. However, an alternative correct method is to enter a value in a column in the first available position. For example in the Memory location 200 column the first two rows could contain the 15 and 46. The other point to note is that if an instruction does not change an entry in a column it is not necessary to enter the value stored again. The trace table only needs to show activity; it does not have to record a complete set of values at each stage in the program execution.

WORKED EXAMPLE 6.03

Tracing an assembly language program

Some instructions for part of a program are contained in memory locations 100 upwards. Some 4-bit binary data values are stored in locations 200 upwards. For illustrative purposes the instructions are shown in assembly language form. At the start of a part of the program, the memory contents are as shown in Figure 6.08.

Address	Contents	Address	Contents
100	`LDD 200`	200	0001
101	`INC ACC`	201	0011
102	`ADD 201`	202	0101
103	`CMP 202`	203	
104	`JPE 106`		
105	`DEC ACC`		
106	`INC ACC`		
107	`STO 203`		

Figure 6.08 The contents of memory addresses before execution of the program begins

The completed trace table for this example is shown in Figure 6.09. Because the program contains a jump instruction it is necessary to record the values for the program counter as well as for the accumulator.

Program counter PC	Accumulator ACC	Memory location 203
100	1000	
101	0001	
102	0010	
103	0101	
104		
106		
107	0110	
108		0110

Figure 6.09 The contents of the program counter and accumulator during program execution

The entries in the table can be explained as follows.

- The first row shows the stored value before execution of this part of the program. There will be a value in the accumulator resulting from an earlier instruction.

- The second row shows the result of the execution of the instruction in location 100 which loads a value into ACC; this is followed by the PC being automatically incremented.
- The next two rows show the value being changed in the ACC by the instructions in 101 and 102 and the automatic incrementing of the PC each time.
- The fifth row has no new value in ACC because only a comparison is being done but there is an automatic increment of the PC.
- The sixth row shows a new value in the PC which has resulted from the execution of the jump instruction which tested for equality and found it to be True.
- The seventh row shows the result of the instruction in location 106 which has incremented the ACC.
- The final row shows the value stored in location 203.

Question 6.01

Can you follow through the changes in the trace table for Worked Example 6.03? Could it be possible for the program to change the content in one of the memory locations 100–107 during execution?

TASK 6.03

Without looking at the explanations provided, trace the assembly language program shown in Table 6.01. Use a value of 3 for MAX and then 7, 8 and 9 as input values.

Reflection Point:

There are several references in this chapter to the content in earlier chapters. Have you checked that you understand how the topics are related by revising the content in the earlier chapters?

Summary

- A machine code instruction consists of an opcode and an operand.
- An assembly language program contains assembly language instructions plus directives that provide information to the assembler.
- A two-pass assembler identifies relative addresses for symbolic addresses in the first pass.
- Processor addressing modes can be: immediate, direct, indirect or indexed.
- Assembly language instructions can be categorised as: data movement, input/output, compare and jump, arithmetic, shift and logical.

Exam-style Questions

1 Three instructions for a processor with an accumulator as the single general purpose register are:

`LDD <address>` for direct addressing

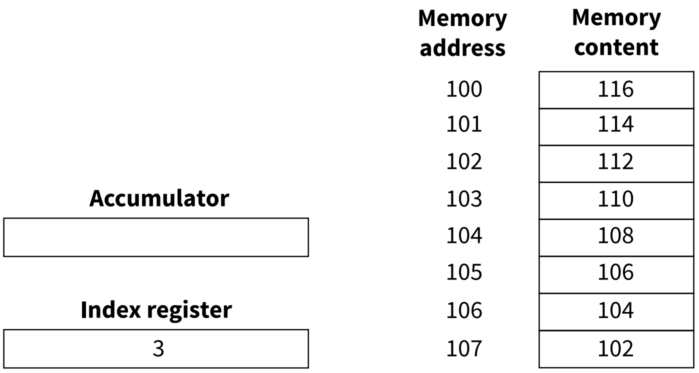
`LDI <address>` for indirect addressing

`LDX <address>` for indexed addressing

In the diagrams below, the instruction operands, the register content, memory addresses and the memory contents are all shown as denary values.

a Consider the instruction `LDD 103`.

i Draw arrows on a copy of the diagram below to explain execution of the instruction. [2]

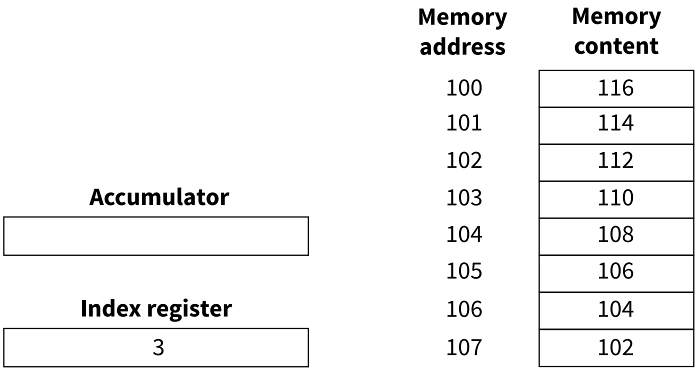


Memory address	Memory content
100	116
101	114
102	112
103	110
104	108
105	106
106	104
107	102

ii Give the contents of the accumulator as a denary value after execution of the instruction. [1]

b Consider the instruction `LDI 107`.

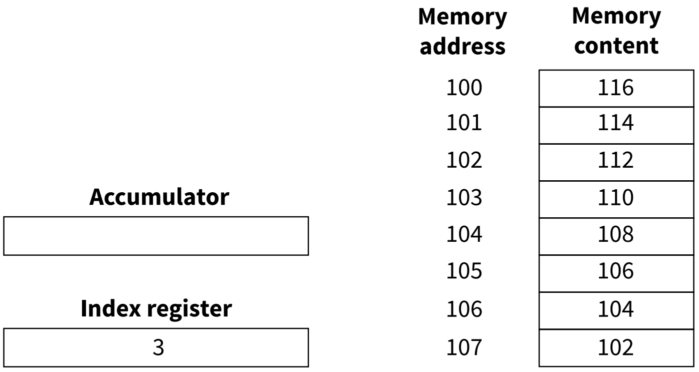
i Draw arrows on a copy of the diagram below to explain execution of the instruction. [3]



Memory address	Memory content
100	116
101	114
102	112
103	110
104	108
105	106
106	104
107	102

ii Give the contents of the accumulator as a denary value after execution of the instruction. [1]

c **i** Draw arrows on a copy of the diagram below to explain the execution of the instruction `LDX 103`. [3]



Memory address	Memory content
100	116
101	114
102	112
103	110
104	108
105	106
106	104
107	102

ii Give the contents of the accumulator as a denary value after the execution. [1]

2 Every machine code instruction has an equivalent in assembly language. An assembly language program will contain assembly language instructions. An assembly language program also contains components not directly transformed into machine code instructions when the program is assembled.

a Describe the use of three types of component of an assembly language program that are not intended to be directly transformed into machine code by the assembler. [6]

b Complete the trace table for the following assembly language program. Note that the LDI instruction uses indirect addressing. [6]

Assembly language program

Memory address	**Memory content**
100	LDD 201
101	INC ACC
102	STO 202
103	LDI 203
104	DEC ACC
105	STO 201
105	ADD 204
107	STO 201
108	END
201	10
202	0
203	204
204	5

Accumulator
0

Memory addresses			
201	**202**	**203**	**204**
10	0	204	5

3 Consider the following assembly language program:

```
<code>        IN
              STO CHARACTER
              IN
              SUB #48
START:        CMP #0
              JPN OUTPUT
              END
OUTPUT:       OUT
              DEC ACC
              JMP START
CHARACTER:
```

a Explain what the program takes as input. [4]

b Explain what the program outputs. [3]

c Complete the symbol table shown below which would be obtained from the first pass of a two-pass assembler. You can use denary numbers for addresses and you can assume that the first instruction is stored in address 0.

Label	**Address**

[4]

4 The table shows assembly language instructions for a processor which has one general purpose register, the Accumulator (ACC) and an index register (IX).

Instruction		**Explanation**
Op code	**Operand**	
`LDD`	`<address>`	Direct addressing. Load the contents of the given address to ACC.
`LDX`	`<address>`	Indexed addressing. Form the address from <address> + the contents of the index register. Copy the contents of this calculated address to ACC.
STO	`<address>`	Store contents of ACC at the given address.
`ADD`	`<address>`	Add the contents of the given address to ACC.
`INC`	`<register>`	Add 1 to the contents of the register (ACC or IX).
`DEC`	`<register>`	Subtract 1 from the contents of the register (ACC or IX).
`CMP`	`<address>`	Compare contents of ACC with contents of <address>.
`JPE`	`<address>`	Following a compare instruction, jump to <address> if the compare was True.
`JPN`	`<address>`	Following a compare instruction, jump to <address> if the compare was False.
`JMP`	`<address>`	Jump to the given address.
`OUT`		Output to screen the character whose ASCII value is stored in ACC.
`END`		Return control to the operating system.

a The diagram shows the current contents of a section of main memory and the index register:

60	0011 0010
61	0101 1101
62	0000 0100
63	1111 1001
64	0101 0101
65	1101 1111
66	0000 1101
67	0100 1101
68	0100 0101
69	0100 0011
. . .	
1000	0110 1001

Index register:

0	0	0	0	1	0	0	0

i Show the contents of the Accumulator after the execution of the instruction:

LDX 60

Accumulator:

Show how you obtained your answer. [2]

ii Show the contents of the index register after the execution of the instruction:

DEC IX

Index register: □□□□□□□□ [1]

Cambridge International AS & A level Computer Science 9608 paper 11 Q9a June 2016

Chapter 7:
Monitoring and control systems

Learning objectives

By the end of this chapter you should be able to:

- show an understanding of monitoring and control systems
- show understanding of how bit manipulation can be used to monitor/control a device.

7.01 Monitoring systems

A monitoring system can be used to create a record of the condition of a system over a period of time. A monitoring system is used more often to detect when a particular physical property of a system goes outside a desired range; for example, if the CPU is too hot.

Discussion Point:

Set yourself a time limit of one minute. During this minute, by considering what measurement will be involved, ask yourself how many different types of monitoring system you can identify.

Let's consider temperature as an example. If this was being monitored under human control, the measurement could be made with a standard mercury thermometer. However, in this chapter we are interested in systems where a computer or microprocessor is being used. These systems require a measuring device that records a value which can be transmitted to the computer. Such a measuring device is a called a **sensor**. An example of a sensor for measuring temperature is a thermocouple, which outputs an electrical voltage that changes with temperature.

It is important to understand that in a monitoring system, a sensor does not have any built-in intelligence, so it cannot take any action if there is a problem. If the temperature measured becomes dangerously high it is the computer that sounds an alarm.

There are a wide variety of sensors available. For some the name indicates the property being measured such as pressure, humidity, carbon monoxide, pH or sound. For others such as an infrared sensor there are different methods of use. A passive infrared sensor just measures the level of infrared light received. In other cases, there is transmission of infrared light with the sensor possibly measuring the level of the light that is reflected back. Other sensors are given a generic name such as a motion sensor, for which different examples will be measuring different physical properties.

Question 7.01

How many different types of motion sensor are you aware of?

7.02 Control systems

A control system has the monitoring activity plus the capability to control a system. The control element of a monitoring and control system needs a device called an **actuator**. An actuator is an electric motor that is connected to a controlling device. It might be used for switching on or off or for adjusting a setting.

Discussion Point:

Refer back to the examples you identified as monitoring systems. How many were actually control systems? If they were monitoring systems could they be modified to become control systems?

Figure 7.01 shows a schematic diagram of a computer-controlled environment.

Note that Figure 7.01 includes an analogue-to-digital converter (ADC) and a digital-to-analogue converter (DAC) as separate components. In a real system they are likely to be integral to the sensor or actuator device.

For the system shown in Figure 7.01 there is a continuing process where the computer at regularly timed intervals signals the sensor to provide a measurement. If the measurement value received by the computer is not in the desired range the computer initiates a control action. The next timed measurement will happen after this control action has taken place. In effect this next measurement provides feedback to the computer on the effect of the control action. Feedback is essential in a control system.

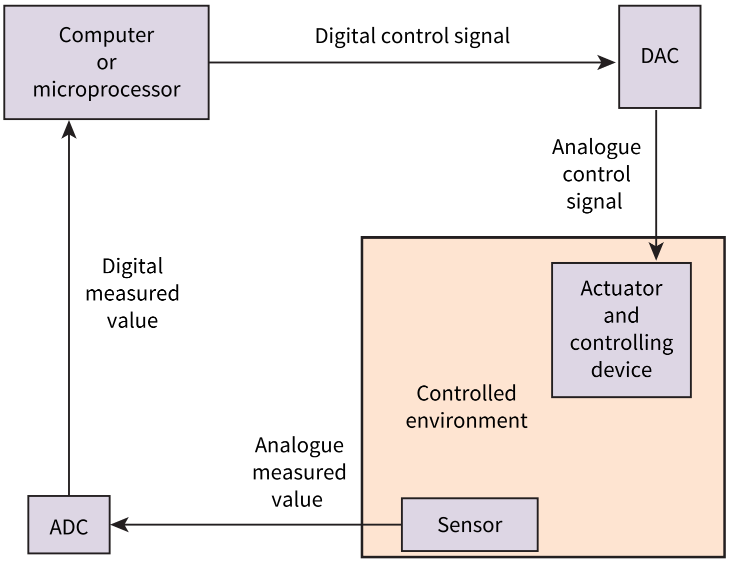


Figure 7.01 Computer-controlled environment

TIP

You need to remember that a sensor does not have any built-in intelligence so it cannot itself take any action if a problem occurs.

A closed-loop feedback control system is a special type of monitoring and control system where the feedback directly controls the operation. Figure 7.02 shows a schematic diagram of such a system. A microprocessor functions as the controller. This compares the value for the actual output, as read by the sensor, with the desired output. It then transmits a value to the actuator which depends on the difference calculated.

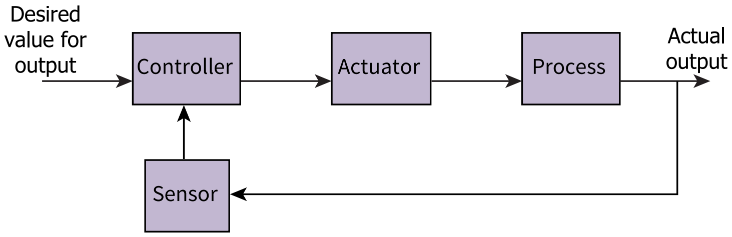


Figure 7.02 Closed-loop feedback control system

Question 7.02

Where would you be likely to find a closed-loop feedback control system?

7.03 Bit manipulation to control devices

The controlling computer or microprocessor has to have a real-time program running continuously. The program can set values for Boolean variables subject to what the sensors detect. For instance, if a controlled environment had two properties to be monitored and controlled, four Boolean variables could be used. Values could be set by assignment statements such as:

```
IF SensorDifference1 > 0 THEN Sensor1HighFlag ← TRUE
IF SensorDifference1 < 0 THEN Sensor1LowFlag ← TRUE
IF SensorDifference2 > 0 THEN Sensor2HighFlag ← TRUE
IF SensorDifference2 < 0 THEN Sensor2LowFlag ← TRUE
```

Another part of the monitoring and control program would then be checking whether any of the four flags were set. The machine code for running such a program could use individual bits to represent each flag. The way that flags could be set and read are illustrated by the following assembly language code fragments. In these code fragments the three least significant bits (positions 0, 1 and 2) of the byte are used as flags.

The following illustrates the setting of all bits to zero which might be used when the system is switched on.	
`LDD 0034`	Loads a byte into the accumulator from an address.
`AND #B00000000`	Uses a bitwise AND operation of the contents of the accumulator with the operand to convert each bit to 0.
`STO 0034`	Stores the altered byte in the original address.
The following illustrates the toggling of the value for one bit. This changes the value of the flag it represents. It might be needed because a problem has been encountered or alternatively because a problem has been solved.	
`LDD 0034`	Loads a byte into the accumulator from an address.
`XOR #B00000001`	Uses a bitwise XOR operation of the contents of the accumulator with the operand to toggle the value of the bit stored in position 0.
`STO 0034`	Stores the altered byte in the original address.
The following illustrates the setting of a bit to have value 1 irrespective of its existing value. This would be a simple way of just reporting a condition repetitively.	
`LDD 0034`	Loads a byte into the accumulator from an address.
`OR #B00000100`	Uses a bitwise OR operation of the contents of the accumulator with the operand to set the flag represented by the bit in position 2. All other bit positions remain unchanged.
`STO 0034`	Stores the altered byte in the original address.
The following illustrates setting all bits to zero except one bit which is of interest. Following this operation, a comparison can be made with a binary value to check if the bit is set. In this example the value would be compared to the binary equivalent of denary 2.	
`LDD 0034`	Loads a byte into the accumulator from an address.
`AND #B00000010`	Uses a bitwise AND operation of the contents of the accumulator with the operand to leave the value in position 1 unchanged but to convert every other bit to 0.
`STO 0034`	Stores the altered byte in the original address.

Reflection Point:

Are you clear that a bitwise logic operation acts on every bit individually; in effect all bits in the accumulator are processed simultaneously?

Summary

- A monitoring system requires sensors.
- A sensor measures a physical quantity; there are many examples, such as temperature, humidity, pH, infrared, pressure, sound and carbon monoxide.
- A monitoring and control system requires sensors and actuators.
- A program used for a monitoring and control system has to operate in real time with an infinite loop that accepts input from the sensors at timed intervals.
- The program transmits signals to the actuators if the values received from the sensors indicate a need for control measures to be taken.
- Bit manipulation can be used within an assembly language program to monitor or control devices.

Exam-style Questions

1 A farmer has a large barn to house poultry for the purpose of collecting the eggs that are laid. The environment inside the barn affects the egg-laying performance of the poultry. Traditionally, the farmer had routinely entered the barn to check that all was well with the environment. If there was a concern, the barn had facilities for correcting the problem.

a More recently a computer-based system has been installed. This allows the farmer to observe data on a computer screen. If any of the data is of concern the system has been programmed to show a flashing red sign on the screen.

i Identify the type of system that the farmer has had installed. [1]

ii Identify the type of devices that have been installed inside the barn. [1]

iii Describe **two** examples of this type of device that could be used and explain what their purpose is with respect to the functioning of the computer-based system. [6]

b The farmer has been told that there is no need for someone to be watching a screen all of the time. A different type of computer-based system could be installed.

i Identify the type of this new computer-based system. [1]

ii Identify the new type of device that would need to be installed inside the barn. (There would be more than one needed). [1]

iii Describe how the new computer-based system would interact with these devices. [2]

2 An assembly language program has been written for a monitoring and control system. The program uses a byte stored in a register in which the bits can be individually set or cleared. An example is:

0	1	1	0	0	0	1	0

Bits 0–3 are set to 0 initially but if one of the two sensors in the system sends a measurement that indicates a problem (measurement is too high or too low) the appropriate bit is set to value 1. Bits 4–7 are also set to 0 initially but if an actuator has to be switched on or off the appropriate bit is set to 1.

a All of the bits in the register need to be set to 0. State which logical bitwise operation is required to be performed on the register content and give the operand that would be used for this. Complete your answer by filling in the boxes.

Logical bitwise operation is:

Performed with the operand:

[2]

b A sensor has recorded a value that is too high so bit 2 must be set to 1 but the other bits must remain unaltered. State which logical bitwise operation is required to be performed on the register content and give the operand that would be used for this. Complete your answer by filling in the boxes.

Logical bitwise operation is:

Performed with the operand:

[2]

c Bit 4 is set to 1 and bit 5 set to 0 but the sensor reading now indicates that there has been an over-reaction so the action of the actuator has to be reversed. This requires bits 4 and 5 to have their values toggled. State which logical bitwise operation is required to be performed on the register content and give the operand that would be used for this. Complete your answer by filling in the boxes:

Logical bitwise operation is:

Performed with the operand:

[3]

3 A gardener grows vegetables in a greenhouse. For the vegetables to grow well, the temperature needs to always be within a particular range.

The gardener is not sure about the actual temperatures in the greenhouse during the growing season. The gardener installs some equipment. This records the temperature every hour during the growing season.

a Name the type of system described. [1]

b Identify **three** items of hardware that would be needed to acquire and record the temperature data. Justify your choice for each. [6]

Item 1

Justification

Item 2

Justification

Item 3

Justification

c Part of the assembly code is:

	Op code	**Operand**
`SENSORS:`		`B00001010`
`COUNT:`		`0`
`VALUE:`		`1`
`LOOP:`	`LDD`	`SENSORS`
	`AND`	`VALUE`
	`CMP`	`#0`
	`JPE`	`ZERO`
	`LDD`	`COUNT`
	`INC`	`ACC`
	`STO`	`COUNT`
`ZERO:`	`LDD`	`VALUE`
	`CMP`	`#8`
	`JPE`	`EXIT`
	`ADD`	`VALUE`
	`STO`	`VALUE`
	`JMP`	`LOOP`
`EXIT:`	`LDD`	`COUNT`
`TEST:`	`CMP`	`…`
	`JGT`	`ALARM`

i Dry run the assembly language code. Start at `LOOP` and finish when `EXIT` is reached.

BITREG	**COUNT**	**VALUE**	**ACC**
`B00001010`	`0`	`1`	

[4]

ii The operand for the instruction labelled TEST is missing.

State the missing operand.

.... [1]

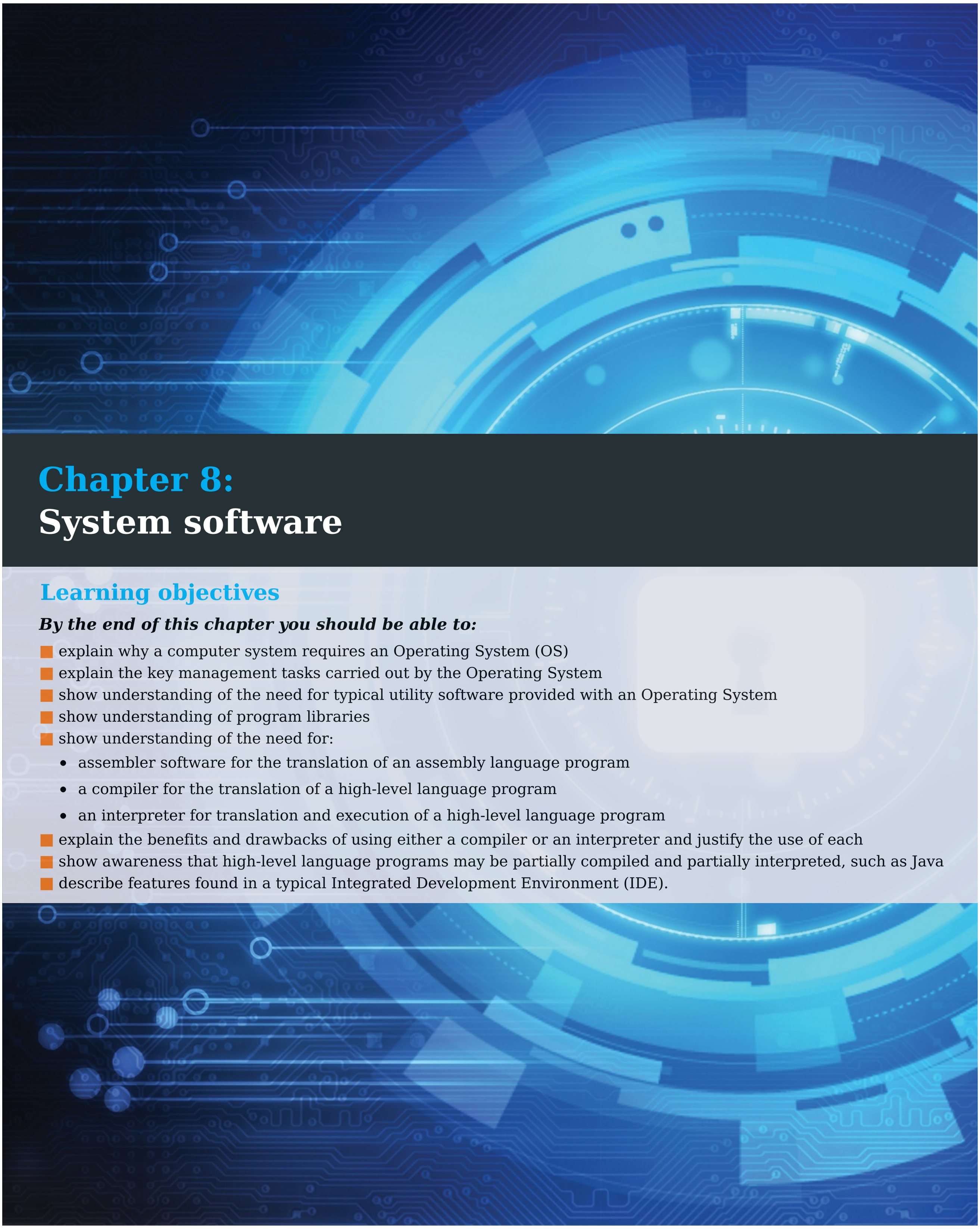
iii The intruder detection system is improved and now has eight sensors.

One instruction in the assembly language code will need to be amended.

Identify this instruction ....

Write the amended instruction .... [2]

Cambridge International AS & A Level Computer Science 9608 paper 31 Q6 June 2016



Chapter 8:
System software

Learning objectives

By the end of this chapter you should be able to:

- explain why a computer system requires an Operating System (OS)
- explain the key management tasks carried out by the Operating System
- show understanding of the need for typical utility software provided with an Operating System
- show understanding of program libraries
- show understanding of the need for:
    - assembler software for the translation of an assembly language program
    - a compiler for the translation of a high-level language program
    - an interpreter for translation and execution of a high-level language program
- explain the benefits and drawbacks of using either a compiler or an interpreter and justify the use of each
- show awareness that high-level language programs may be partially compiled and partially interpreted, such as Java
- describe features found in a typical Integrated Development Environment (IDE).

8.01 System software

In the 1960s, the likely arrangement for using a computer would be something like this.

1. Enter machine room with deck of punched cards and a punched paper tape reel.
2. Switch on computer.
3. Put deck of cards into card reader and press button.
4. Put paper tape into tape reader and press button.
5. Press button to run the program entered into memory from the punched cards using the data entered into memory from the paper tape.
6. Press button to get output printed on the line-printer.
7. Switch off computer.
8. Leave machine room with deck of cards, paper tape and line-printer output.

The user controlled the computer hardware by pressing buttons. Just try to imagine how many buttons would be needed if you had to control a computer in the same way today.

The missing component from the 1960s computer was, of course, an **operating system**; in other words, some software to control the hardware and interact with application software. An operating system is an example of a type of software called ‘system software’. This distinguishes it from application software, which is created to perform a specific task for a computer user rather than just helping to run the system.

8.02 Operating system activities

Operating systems are extremely complex and it is not possible to give a full description here of what an operating system is. However, what an operating system generally does is to provide an environment where programs can be run that are of benefit to a user.

The activities of an operating system can be sub-divided into different categories, some of which overlap with each other. We are going to look at each of the various tasks carried out by the operating system. Details of how some of them are carried out are discussed in Chapter 20 (Sections 20.01 to 20.05).

User–system interface

A user interface is needed to allow the user to get the software and hardware to do something useful. An operating system should provide at least the following for user input and output:

- a command-line interface
- a graphical user interface (GUI).

Discussion Point:

Have you any experience of using a command-line interface?

Program–hardware interface

Programmers write software and users run this software. The software uses the hardware. The operating system has to ensure that the hardware does what the software wants it to do. Program development tools associated with a programming language allow a programmer to write a program without needing to know the details of how the hardware, particularly the processor, actually works. The operating system then has to provide the mechanism for running the developed program.

Resource management

When a program has started to run it is described as a **process**. In a modern computer system, a process will not be able to run to completion without interruption. At any time there will be many processes running on the computer system. Each process needs access to the resources provided by the computer system.

The resource management provided by the operating system aims to achieve the best possible efficiency in computer system use. The two most important aspects of this are:

- scheduling of processes
- resolution of conflicts when two processes require the same resource.

Memory management

There are three important aspects of memory management.

- Memory protection ensures that one program does not try to use the same memory locations as another program.
- The memory organisation scheme is chosen to achieve the best usage of limited memory size, for example, virtual memory involving paging or segmentation.
- Memory usage optimisation involves decisions about which processes should be in main memory at any one time and where they are stored in this memory.

Device management

Every computer system has a variety of components that are categorised as ‘devices’. Examples include the monitor screen, the keyboard, the printer and the webcam. The management of these requires:

- installation of the appropriate device driver software
- control of usage by processes.

File management

Three major features here are the provision of:

- file naming conventions
- directory (folder) structures
- access control mechanisms.

Security management

Chapter 9 (Sections 9.02, 9.03 & 9.04) and Chapter 21 contain extensive accounts of security issues and measures used to prevent problems. For the operating system the many aspects of security management include:

- provision for recovery when data is lost
- prevention of intrusion
- ensuring data privacy.

Error detection and recovery

Errors can arise in the execution of a program either because it was badly written or because it has been supplied with inappropriate data. Other errors are associated with devices not working correctly. Whatever the cause of an error, the operating system should have the capability to interrupt a running process and provide error diagnostics where appropriate. In extreme cases, the operating system needs to be able to shut down the system in an organised fashion without loss of data.

TASK 8.01

For each of the above categories of operating system task, each point could be placed in a different category. Make an abbreviated list of these categories and add arrows to show different categories where each point could be placed.

Question 8.01

It is useful to describe the management tasks carried out by an operating system as being primarily one of the following types:

- those assisting the user of the system
- those concerned with the running of the system.

Considering the management tasks that have already been categorised, can you identify them as belonging to one or other of the above types? Are there any problems in doing this?

8.03 Utility software

A utility program can be provided by the operating system or it can be installed separately. It is a program that is not executed as part of the normal running of the operating system. Instead it is a program that the user or the operating system can decide to run when needed. For example, some utility programs manage hard disks.

Hard disk formatter and checker

A disk formatter will typically carry out the following tasks:

- removing existing data from a disk that has been used previously
- setting up the file system on the disk, based on a table of contents that allows a file recognised by the operating system to be associated with a specific physical part of the disk
- partitioning the disk into logical drives if this is required.

Another utility program, which can be a component of a disk formatter, performs disk contents analysis and, if possible, disk repair when needed. The program first checks for errors on the disk. Some errors arise from a physical defect resulting in what is called a 'bad sector'. There are a number of possible causes of bad sectors. However, they usually arise either during manufacture or from mishandling of the system. An example is moving the computer without ensuring that the disk heads are secured away from the disk surface.

Other errors arise from an event such as a loss of power or an error causing sudden system shutdown. As a result some of the files stored on the disk might no longer be useable. A disk repair utility program can mark bad sectors and ensure that the file system no longer tries to use them. When the integrity of files has been affected, the utility might be able to recover some of the data but otherwise it has to delete the files.

Hard disk defragmenter

A disk defragmenter utility also can be part of a disk repair utility program but it is not primarily concerned with errors. A perfectly functioning disk will, while in use, gradually become less efficient because the constant creation, editing and deletion of files leaves them in a fragmented state. The cause of this is the logical arrangement of data in sectors as discussed in Chapter 3 (Section 3.04), which does not allow a file to be stored as a single block of data.

A simple illustration of the problem is shown in Figure 8.01. Initially file A occupies three sectors fully and part of a fourth one. File B is small so occupies only part of a sector. File C occupies two sectors fully and part of a third. When File B is deleted, the sector remains unfilled because it would require too much system CPU time to rearrange the file organisation every time there is a change. When File A is extended it completely fills the first four sectors and the remainder of the extended file is stored in all of Sector 8 and part of Sector 9. Sector 4 will only be used again if a small file is created or if the disk fills up, when it might store the first part of a longer file.

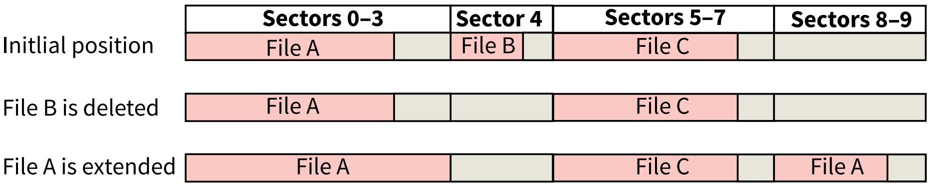


Figure 8.01 File fragmentation on a hard disk

A defragmenter utility program reorganises the file storage to return it to a state where all files are stored in one block across a sequence of sectors. For a large disk this will take some time. It will be impossible if the disk is too full because of the lack of working space for the rearrangement.

TASK 8.02

If you have never used a disk defragmenter or disk repair utility program, can you get access to a system where you can use one? If so, note the changes that are carried out and recorded by the utility program.

Backup software

It is quite likely that you perform a manual backup of your own files every now and then using a flash memory stick. However, an easier way to perform backup is to use a backup utility program. Such a program will:

- establish a schedule for backups
- only create a new backup file when there has been a change.

Question 8.02

In the systems that you use, the technical staff will have made provision for backup. Can you find out what procedures are followed and what hardware is used for this?

File compression

A file compression utility program can be used regularly by an operating system to minimise hard disk storage requirements. Whether or not the operating system does this, a user can still choose to install a suitable program. However, as was discussed in Chapter 1 (Section 1.07) compression is most important when transmitting data. In particular, it makes sense to compress (or ‘zip’) a file before attaching it to an email.

Virus checker

A virus-checking program should be installed as a permanent facility to protect a computer system. In an ideal world, it would only need to be used to scan a file when the file initially entered the system. Unfortunately, this ideal state can never be realised. When a new virus comes along there is a delay before it is recognised and a further delay before a virus checker has been updated to deal with it. As a result, it is necessary for a virus checker to be regularly updated and for it to scan all files on a computer system as a matter of routine.

8.04 Program libraries

The ‘programs’ in a program library are usually subroutines created to carry out particular tasks. A programmer can use these within their own programs.

All newly developed programs are likely to contain errors, which only become apparent as the programs are tested or used. It saves a programmer a lot of time and trouble to be able to include already tried and tested subroutines taken from a program library.

The most obvious examples of library routines are the built-in functions available for use when programming in a particular language. Examples of these are discussed in Chapter 14 (Section 14.07). Another example is the collection of over 1600 procedures for mathematical and statistics processing available from the Numerical Algorithms Group (NAG) library. This organisation has been creating routines since 1971 and they are universally accepted as being as reliable as software ever can be.

In Section 8.05, we will discuss the methods available for translation of source code. For now, we simply need an overview of what happens. The source code is written in a programming language of choice. If a compiler is used for the translation and no errors are found, the compiler produces object code (machine code). This code cannot be executed by itself. Instead it has to be linked with the code for any subroutines used by it. It is possible to carry out the linking before loading the full code into memory and running it.

There is a major disadvantage in linking library routines into the executable code. This is because every program using a routine has to have its own copy. This increases the storage space requirement for the executable file. It also increases memory usage when more than one process uses the routine.

The alternative is to use a routine from a dynamic linked library (DLL). When a DLL routine is available the executable code just requires a small piece of code to be included. This allows it to link to the routine, which is stored separately in memory, when execution of the program needs it. Many processes can be linked to the same routine. This has the advantage that the executable files for all programs need less storage space. Memory requirement is also minimised. Another advantage is that if a new version of the routine becomes available it can be loaded into memory so that any program using it is automatically upgraded.

The main disadvantage of using a DLL is that the program is relying on the routine being available and performing the expected function. If for some reason the DLL becomes corrupted or a new version has bugs not yet discovered the program will fail or produce an erroneous result. The user running the program will find it difficult to establish what needs to be done to get the program to run without error.

8.05 Language translators

The use of an assembler for translating a program written in assembly language has been discussed in Chapter 6 (Sections 6.02, 6.03 & 6.04). This chapter will introduce the translators that are used to translate a program written in a high-level procedural language.

Compilers and interpreters

The starting point for using either a compiler or an interpreter is a file containing source code, which is a program written in a high-level language.

For an interpreter the following steps apply.

1. The interpreter program, the source code file and the data to be used by the source code program are all made available.
2. The interpreter program begins execution.
3. The first line of the source code is read.
4. The line is analysed.
5. If an error is found, this is reported and the interpreter program halts execution.
6. If no error is found, the line of source code is converted to an intermediate code.
7. The interpreter program uses this intermediate code to execute the required action.
8. The next line of source code is read and Steps 4–8 are repeated.

For a compiler the following steps apply.

1. The compiler program and the source code file are made available but no data is needed.
2. The compiler program begins execution.
3. The first line of the source code is read.
4. The line is analysed.
5. If an error is found this is recorded.
6. If no error is found the line of source code is converted to an intermediate code.
7. The next line of source code is read and Steps 4–7 are repeated.
8. When the whole of the source code has been dealt with one of the following happens.
    - If no error is found in the whole source code the complete intermediate code is converted into object code.
    - If any errors are found a list of these is output and no object code is produced.

Execution of the program can only begin when the compilation has shown no errors. This can take place automatically under the control of the compiler program if data for the program is available. Alternatively, the object code is stored and the program is executed later with no involvement of the compiler.

Discussion Point:

What type of facility for language translation are you being provided with? Does your experience of using it match what has been described here?

The advantages and disadvantages to a programmer of creating interpreted or compiled programs.

- An interpreter has advantages when a program is being developed because errors can be identified as they occur and corrected immediately without having to wait for the whole of the source code to be read and analysed.
- An interpreter has a disadvantage in that during a particular execution of the program, parts of the

code which contain syntax errors may not be accessed so if errors are still present, they are not discovered until later.

- An interpreter has a disadvantage when a program is error free and is distributed to users because the source code has to be sent to each user.
- A compiler has the advantage that an executable file can be distributed to users, so the users have no access to the source code.

The advantages and disadvantages to the user of interpreted or compiled programs.

- For an interpreted program, the interpreter and the source code have to be available each time that an error-free program is run.
- For a compiled program, only the object code has to be available each time that an error-free program is run.
- Compiled object code will provide faster execution than is possible for an interpreted program.
- Compiled object code is less secure because it could contain a virus.

Whether an interpreter or a compiler is used, a program can only be run on a particular computer with a particular processor if the interpreter or compiler program has been written for that processor.

If there is an option available the choice of an interpreter is justified when a program is being developed because:

- one error in a program can lead to several other errors occurring
- an interpreter can detect and correct an early error so limiting subsequent ones
- the debugging facilities provided in association with the interpreter speed this process.

The choice of a compiler is justified when the programmer is confident that the program is as near error-free as possible because:

- an executable file can be created
- this can be distributed for general use
- execution of the program will be faster than if an interpreter were used.

Java

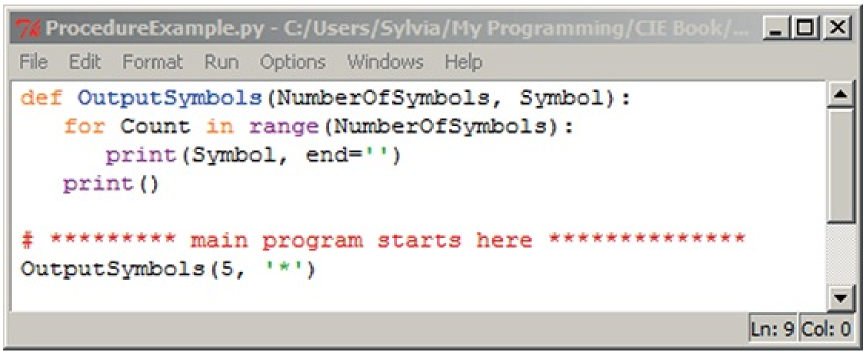
When the programming language Java was created, a different philosophy was applied to how it should be used. Each different type of computer has to have a Java Virtual Machine created for it. Then when a programmer writes a Java program this is compiled first of all to create what is called Java Byte Code. When the program is run, this code is interpreted by the Java Virtual Machine. The Java Byte Code can be transferred to any computer that has a Java Virtual Machine installed.

8.06 Features found in a typical Integrated Development Environment (IDE)

Whatever language is used for writing source code and whatever compiler or interpreter is being used there will be one or more IDEs available to assist the programmer. This section discusses the types of feature that should be provided by an IDE.

Prettyprinting

Prettyprint refers to the presentation of the program code typed into an editor. For example, the Python IDLE (see Figure 8.02) automatically colour-codes keywords, built-in function calls, comments, strings and the identifier in a function header. In addition, indentation is automatic.

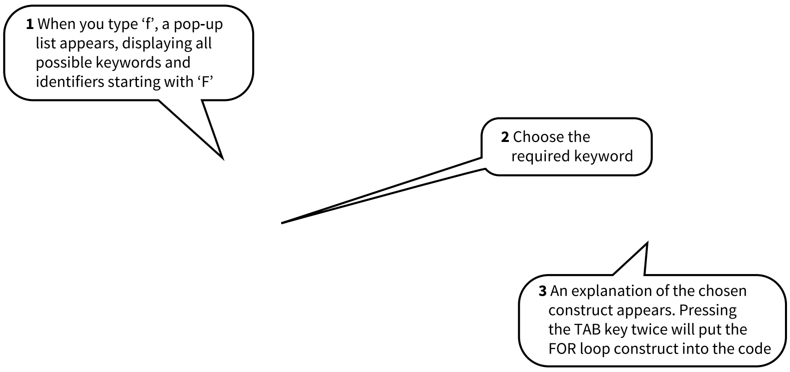

ProcedureExample.py - C:/Users/Sylvia/My Programming/CIE Book/...
File Edit Format Run Options Windows Help
Ln: 9 Col: 0

```
def OutputSymbols(NumberOfSymbols, Symbol):
    for Count in range(NumberOfSymbols):
        print(Symbol, end='')
    print()

# ********* main program starts here **************
OutputSymbols(5, '*')
```

Figure 8.02 Prettyprint in the Python IDLE

Context-sensitive prompts

This feature displays hints (or a choice of keywords) and available identifiers that might be appropriate at the current insertion point of the program code. Figure 8.03 shows an example of the Visual Studio editor responding to text typed in by the programmer.



1 When you type 'f', a pop-up list appears, displaying all possible keywords and identifiers starting with 'F'

2 Choose the required keyword

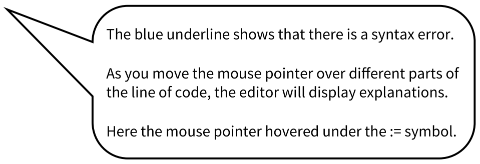
3 An explanation of the chosen construct appears. Pressing the TAB key twice will put the FOR loop construct into the code

Figure 8.03 Context-sensitive prompts in the Visual Studio editor

Dynamic syntax checks

When a line has been typed, some editors perform syntax checks and alert the programmer to errors.

Figure 8.04 shows an example of the Visual Studio editor responding to a syntax error.



The blue underline shows that there is a syntax error.

As you move the mouse pointer over different parts of the line of code, the editor will display explanations.

Here the mouse pointer hovered under the := symbol.

Figure 8.04 Dynamic syntax check in the Visual Studio editor

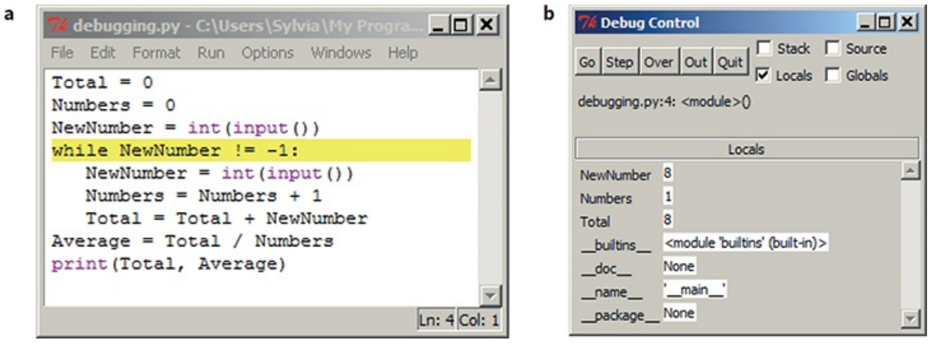
Expanding and collapsing code blocks

When working on program code consisting of many lines of code, it saves excessive scrolling if you can collapse blocks of statements.

Debugging

An IDE often contains features to help with **debugging**.

If a Debugger feature has been switched on it is possible to select a breakpoint. When the program starts running it will stop when it reaches the breakpoint. The program can then be stepped through, one instruction at a time. Figure 8.05 shows the windows presented to the user in the Python IDLE when this feature is being used.



```
Total = 0
Numbers = 0
NewNumber = int(input())
while NewNumber != -1:
    NewNumber = int(input())
    Numbers = Numbers + 1
    Total = Total + NewNumber
Average = Total / Numbers
print(Total, Average)
```

```
debugging.py:4: <module>()
```

NewNumber	8
Numbers	1
Total	8
__builtins__	<module 'builtins' (built-in)>
__doc__	None
__name__	'__main__'
__package__	None

Figure 8.05 (a) A Python program showing a breakpoint; (b) the Debug Control window

TASK 8.03

Investigate the facilities in the editors you have available. If you have a choice of editors, you may like to use the editor with the most helpful facilities.

Reflection Point:

Much of the discussion in this chapter only summarises what an operating system does. Do you think it would be helpful to move on immediately to have a look at some of the content in sections 20.01 to 20.05 of Chapter 20?

Summary

- Operating system tasks can be categorised in more than one way, for example, some are for helping the user, others are for running the system.
- Utility programs include hard disk utilities, backup programs, virus checkers and file compression utilities.
- Library programs, including Dynamic Link Library (DLL) files, are available to be incorporated into programs; they are usually subroutines and are very reliable.
- A high-level language can be translated using an interpreter or a compiler.

- A Java compiler produces Java Byte Code which is interpreted by a Java Virtual Machine.
- An integrated Development Environment (IDE) contains many features that provide support for a programmer when a program is being written and when it is being corrected.

Exam-style Questions

1 **a** One of the reasons for having an operating system is to provide a user interface to a computer system.

i Name **two** different types of interface that an operating system should provide. [2]

ii Identify for each type of interface a device that could be used to enter data. [2]

b Identify and explain briefly **three** other management tasks carried out by an operating system. [6]

2 **a** A PC operating system will make available to a user a number of utility programs.

i Identify **two** utility programs that might be used to deal with a hard disk problem. [2]

ii For each of these utility programs explain why it might be needed and explain what it does. [5]

iii Identify **two** other utility programs for a PC user. [2]

b Library programs are made available for programmers.

i Explain why a programmer should use library programs. [3]

ii Identify **two** examples of a library program. [2]

3 Assemblers, compilers and interpreters are examples of translation programs.

a State the difference between an assembler and a compiler or interpreter. [1]

b A programmer can choose to use an interpreter or a compiler.

i State **three** differences between how an interpreter works and how a compiler works. [3]

ii Discuss the advantages and disadvantages of an interpreter compared to a compiler. [4]

iii If a programmer chooses Java, a special approach is used. Identify **one** feature of this special approach. [1]

4 **a** Explain the meaning of the following terms:

Prettyprinting

Context-sensitive prompt

Dynamic syntax check

Debugging [8]

b Describe the features you would expect a debugger to provide. [4]

5 Before it is used, a hard disk is formatted using disk formatter software.

a Explain why formatting is needed. [2]

b Eventually, the performance of the hard disk deteriorates.

Name **three** other utility programs that might be required. State why each is needed. [6]

Cambridge International AS & A level Computer Science 9608 paper 12 Q10 November 2015

6 A programmer is writing a program that includes code from a program library.

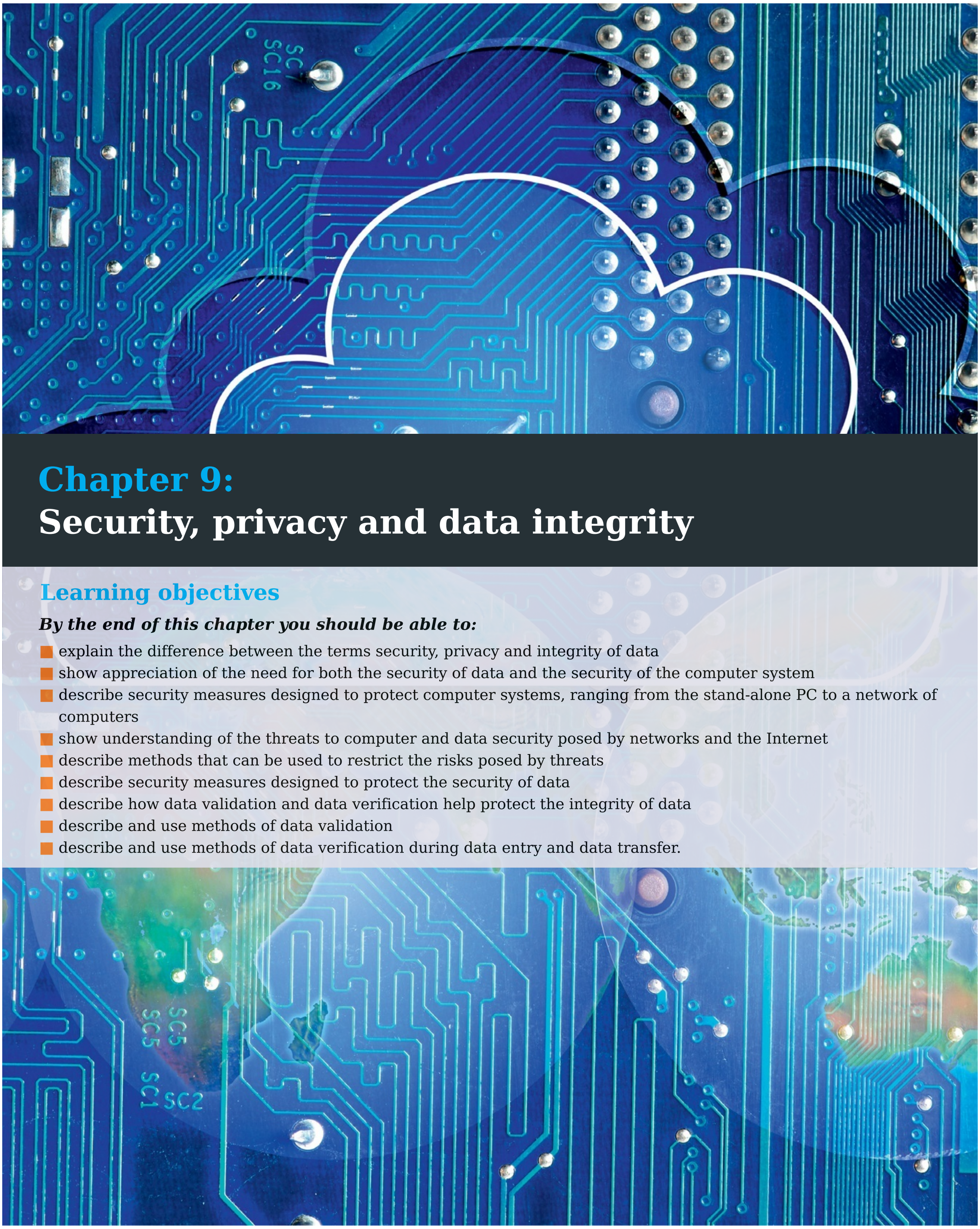
a Describe **two** benefits to the programmer of using one or more library routines. [4]

b The programmer decides to use a Dynamic Link Library (DLL) file.

i Describe **two** benefits of using DLL files. [4]

ii State **one** drawback of using DLL files. [2]

Cambridge international AS & A Level Computer Science 9608 paper 12 Q8 November 2016



Chapter 9:
Security, privacy and data integrity

Learning objectives

By the end of this chapter you should be able to:

- explain the difference between the terms security, privacy and integrity of data
- show appreciation of the need for both the security of data and the security of the computer system
- describe security measures designed to protect computer systems, ranging from the stand-alone PC to a network of computers
- show understanding of the threats to computer and data security posed by networks and the Internet
- describe methods that can be used to restrict the risks posed by threats
- describe security measures designed to protect the security of data
- describe how data validation and data verification help protect the integrity of data
- describe and use methods of data validation
- describe and use methods of data verification during data entry and data transfer.

9.01 Definitions of data integrity, privacy and security

Data integrity

It is easy to define integrity of data but far less easy to ensure it. Only accurate and up-to-date data has **data integrity**. Any person or organisation that stores data needs it to have integrity. We will discuss ways to achieve data integrity in this chapter, and also in Chapter 11 (Sections 11.01 & 11.02).

Data privacy

Data privacy is about keeping data private rather than allowing it to be available in the public domain. The term may be applied to a person or an organisation. Every individual has an almost limitless amount of data associated with their existence. Assuming that an individual is not engaged in criminal or subversive activities, he or she should be in control of which data about himself or herself is made public and which data remains private. An organisation can have data that is private to the organisation, such as the minutes of management meetings, but we will not discuss this here.

For an individual there is little chance of data privacy if there is not a legal framework in place to penalise offenders who breach this privacy. This framework is provided by a **data protection law**. The following aspects should be noted about such laws.

- The major focus relates to personal, therefore private, data that an individual supplies to an organisation.
- The data is supplied to allow the organisation to use it but only for purposes understood and agreed by the individual.
- Data protection laws oblige organisations to ensure the privacy and the integrity of this data.
- Unfortunately, having laws does not guarantee adherence to them but they do act as a deterrent if wrong-doers can be subject to legal proceedings.

Discussion Point:

What data protection laws are in place in your country? Are you familiar with any details of these laws?

Data protection normally applies to data stored in computer systems with the consent of the individual. Should these laws be extended to cover storage of data obtained from telephone calls or search engine usage?

Data security

Data are secure if they are available for use when needed and the data made available are the data that were stored originally. The security of data has been breached if any data have been lost or corrupted.

Data security must be achieved before either data integrity or data privacy can be achieved, but data security does not by itself guarantee either data integrity or data privacy. One of the requirements for protection of data is the security of the system used to store the data. System security does not just protect data. There are two primary aims of system security measures:

- to ensure that the system continues to carry out the tasks users need
- to ensure that only authorised users have access to the system.

9.02 Threats to the security of a computer system and of the data stored in it

The threats to the security of a system include the following types:

- individual user not taking appropriate care
- internal mismanagement
- natural disasters
- unauthorised intrusion into the system by an individual
- malicious software entering the system.

TIP

Try to keep a perspective with relation to hacking; it is only one of many security issues.

Threats to computer and data security posed by networks and the Internet

We are all continuously at risk from security threats to systems we use ourselves and to all of the systems used by organisations upon which we rely. The dominant factor is that none of these systems are stand-alone; all are connected to networks and through these networks to the Internet. One cause of concern is the hacker who is someone intent on gaining unauthorised access to a computer system. A hacker who achieves this aim might gain access to private data. Alternatively, a hacker might cause problems by deleting files or causing problems with the running of the system. The other major cause of concern is malicious software entering the system.

Types of malware

Malware is the everyday name for malicious software. It is software that is introduced into a system for a harmful purpose. One category of malware is where program code is introduced to a system. The various types of malware-containing program code are:

- virus: tries to replicate itself inside other executable code
- worm: runs independently and transfers itself to other network hosts
- logic bomb: stays inactive until some condition is met
- Trojan horse: replaces all or part of a previously useful program
- spyware: collects information and transmits it to another system
- bot: takes control of another computer and uses it to launch attacks.

The differences between the different types are not large and some examples come into more than one of these categories. The virus category is often subdivided according to the software that the virus attaches itself to. Examples are boot sector viruses and macro viruses.

Malware can also be classified in terms of the activity involved:

- phishing: sending an email or electronic message from an apparently legitimate source requesting confidential information
- pharming: setting up a bogus website which appears to be a legitimate site
- keylogger: recording keyboard usage by the legitimate user of the system.

Question 9.01

Carry out some research to find some examples of how phishing and pharming might be attempted.

System vulnerability arising from user activity

Many system vulnerabilities are associated directly with the activities of legitimate users of a system. Two examples which do not involve malware are as follows.

- The use of weak passwords and particularly those which have a direct connection to the user. A poor choice of password gives the would-be hacker a strong chance of guessing the password and thus being able to gain unauthorised access.
- A legitimate user not recognising a phishing or pharming attack and, as a result, giving away sensitive information.

A legitimate user with a grievance might introduce malware deliberately. More often, malware is introduced accidentally by the user. Typical examples of actions that might introduce malware are:

- attaching a portable storage device
- opening an email attachment
- accessing a website
- downloading a file from the Internet.

Vulnerability arising from within the system itself

Systems themselves often have security weaknesses. The following are examples of this.

1. Operating systems often lack good security. Over time, there is a tendency for operating systems to increase in complexity, which leads to more opportunities for weak security. Operating systems have regular updates, often because of a newly discovered security vulnerability.
2. In the past, commonly used application packages allowed macro viruses to spread, but this particular problem is now largely under control.
3. A very specific vulnerability is buffer overflow. Programs written in the C programming language, of which there are very many, do not automatically carry out array bound checks. A program can be written to deliberately write code to the part of memory that is outside the address range defined for the array, set up as a buffer. The program overwrites what is stored there so when a later program reads this overwritten section it will not execute as it should. Sometimes this only causes minor disruption, but a cleverly designed program can permit an attacker to gain unauthorised access to the system and cause serious problems.

9.03 Security measures for protecting computer systems

Disaster recovery

Continuity of operation is vital for large computer installations that are an integral part of the day-to-day operations of an organisation. Measures are needed to ensure that the system continues working whatever event occurs or, if there has to be a system shut-down, at the very least to guarantee that the service will start again within a very short time.

Such measures come under the general heading of disaster recovery contingency planning. The contingency plan should be based on a risk assessment. The plan will have provision for an alternative system to be brought into action. If an organisation has a full system always ready to replace the normally operational one, it is referred to as a 'hot site'. By definition, such a system has to be remote from the original system to allow recovery from natural disasters such as earthquake or flood.

Safe system update

A special case of system vulnerability arises when there is a major update of hardware and/or software. Traditionally, organisations had the luxury of installing and testing a new system over a weekend when no service was being provided. In the modern era, it is more usual to have systems that can be accessed at any time and (often) from any location. A company is never closed for business. As a result, organisations may need to have the original system and its replacement running in parallel for a period to ensure continuity of service.

Discussion Point:

Major failings of large computer systems are well documented. You could carry out research to find some examples. Find an example of where the crisis was caused by technology failure and a different example where some natural disaster was the cause.

User authentication

Even if a PC is used by only one person there should be a user account set up. User accounts are, of course, essential for a multi-user (timesharing) system. The main security feature of a user account is the **authentication** of the user. The normal method is to associate a password with each account. In order for this to be effective the password needs a large number of characters including a variety of those provided in the ASCII scheme.

TASK 9.01

1 Create an example of a secure password using eight characters (but not one you are going to use).

2 Assuming that each character is taken from the ASCII set of graphic characters, how many different possible passwords could be defined by eight characters?

3 Do you think this is a sufficient number of characters to assume that the password would not be encountered by someone trying all possible passwords in turn to access the system?

Alternative methods of authentication include biometric methods and security tokens. A biometric method might require examination of a fingerprint or the face or the eye. A security token can be a small item of hardware provided for each individual user that confirms their identity. Similar protection can be provided by software with the user required to provide further input after the password has been entered. Normal practice is to combine one of these alternative methods with the password system.

Good practice

General good practice that helps to keep a personal computer secure includes not leaving the computer switched on when unattended, not allowing someone else to observe you accessing the computer and not writing down details of how you access it.

Users may attach portable storage devices to a system, but this increases the risk of transferring malware into the system. This risk is reduced by an organisation having a policy banning the use of such devices or at least limiting their use. Unfortunately, this is difficult if normal business processes require portability of data.

Firewall

The primary defence to malware entering a system through a network connection is to install a **firewall**. Ideally a firewall will be a hardware device that acts like a security gate at an international airport. Nothing is allowed through without it being inspected. Alternatively, a firewall can run as software. Data must enter the system, but it can be inspected immediately. A firewall can inspect the system addresses identified in the transmission of data, but can sometimes also inspect the data itself to check for anything unusual or inappropriate.

Digital signature

If an incoming transmission is an email, you might want to check the identity of the sender. The solution is to insist that the sender attaches a digital signature to the email. Some details of this are discussed in Chapter 21 (Section 21.02).

Anti-virus software and intrusion detection

Security measures restricting access to a system do not guarantee success in removing all threats. It is therefore necessary to have, in addition, programs running on a system to check for problems. One option is to install what is normally referred to as anti-virus software but which is usually aimed at combating any type of malware. This carries out regular scans to detect any malware and to remove or deactivate it. Possibly special-purpose anti-spyware software might be installed. Another option is to install an intrusion detection system that will take as input an audit record of system use and look for examples that do not match expected system activity.

Unfortunately, people intent on causing damage to systems are using methods that are becoming ever more sophisticated. The defence methods have to be improved continually to counter these threats.

9.04 Security measures for protecting data

Recovering from data loss

In addition to problems arising from malicious activity there are a variety of reasons for accidental loss of data:

- a disk or tape gets corrupted
- a disk or tape is destroyed
- the system crashes
- the file is erased or overwritten by mistake
- the location of the file is forgotten.

For these reasons, every system should have a backup procedure to recover lost data. The system administrator decides on the details of the procedure. The principles for the procedure traditionally followed are straightforward:

- a full backup is made at regular intervals, perhaps weekly
- at least two generations of full backup are kept in storage
- incremental backups are made on a daily basis.

For maximum security the backup disks or tapes are stored away from the system in a fire-proof and flood-proof location.

This works well when an incremental backup occurs done overnight with the full backup handled at the weekend. With systems running 24/7, data in the system might be changing at any time, and a simple approach to backup would leave data in an inconsistent state.

One solution is to have a backup program that effectively freezes the file store while data is being copied. At the same time changes that are happening due to system use are recorded elsewhere within the system. The changes can then be implemented once the backup copy has been stored.

An alternative approach is to use a disk-mirroring strategy. In this case, data is simultaneously stored on two disk systems during the normal operation of the system. The individual disk systems might be at remote locations as part of a disaster recovery plan.

Restricting access to data

If a user has logged in, they have been authorised to use the computer system but not necessarily all of it. In particular, the system administrator may identify different categories of user with different needs with respect to the data they are allowed to see and use. A typical example is that one employee should be able to use the system to look up another employee’s internal phone number. This should not allow the employee at the same time to check the salary paid to the other employee.

The solution is to have an **authorisation** policy which gives different access rights to different files for different individuals. For a particular file, a particular individual might have no access at all or possibly read access but not write access. In another case, an individual might have read access but not unrestricted write access.

Protecting data content

Even with appropriate security measures in place, a system and its data might still be accessed by someone who overcomes security to break into the system. This type of access can still be made a waste of time and effort if the stored data cannot be read. Data can be encrypted to ensure this. Some details of encryption methods are discussed in Chapter 21 (Sections 21.01, 21.03, 21.04, 21.05 & 21.06).

9.05 Data validation and verification

Data integrity can never be guaranteed, but the chances are improved if appropriate measures are taken when data originally enters a system or when it is transmitted from one system to another.

Validation of data entry

The term **validation** is a somewhat misleading one. It seems to imply that data is accurate if it has been validated. This is far from the truth. For example, if entry of a name is expected but the wrong name is entered, it will still be recognised as a name and therefore accepted as valid. Validation can only prevent incorrect data if there is an attempt to input data that is of the wrong type, in the wrong format or out of range.

Data validation is implemented by software associated with a data entry interface. There are a number of different types of check that can be made. Typical examples are:

- a presence check to ensure that an entry field is not left blank
- a format check, for example a date has to be dd/mm/yyyy
- a length check, for example with a telephone number
- a range check, for example the month in a date must not exceed 12
- a limit check, for example a maximum number of years for a person’s age
- a type check, for example only a numeric value for the month in a date
- an existence check, for example that a file exists with the filename referred to in the data entry.

Verification of data entry

When data is entered into a system, **verification** means getting the user to confirm that the data entered was what was intended to be entered. Unfortunately, this still does not mean that the data entered is correct. Double entry is one method of verification. The most common example is when a user is asked to supply a new password. There will usually be a request for the password to be re-entered. A second method is to use a visual check of what has been entered. If a form has been filled in, it always makes sense to read through the data entered before sending the form off to its destination.

Check digit

When a series of numbers are used to identify something, it is possible to use a check digit method of verification. There are many different options here but they all require a calculation to be made with the numbers that have been entered. The final part of the calculation involves an integer division from which the remainder is added as an extra one or two digits at the end of the series of numbers. In one scheme where only a single check digit is allowed, the letter X is used when the remainder is calculated as 10. When the data is subsequently read the same calculation is carried out and the result is compared to the check digit that had been stored. This technique is used for a bar code or for the ISBN for a book.

Verification during data transfer

It is possible for data to be corrupted during transmission. Often, this happens when an individual bit is flipped from 1 to 0 or from 0 to 1. Verification techniques need to check on some property associated with the bit pattern.

The simplest approach is to use a simple one-bit parity check. This is particularly easy to do if data is transferred in bytes using a seven-bit code. Either even or odd parity can be implemented in the eighth bit of the byte. Assuming even parity, this is the procedure.

1 At the transmitting end, the number of 1s in the seven-bit code is counted.

2 If the count gives an even number, the parity bit is set to 0.

3 If the count gives an odd number, the parity bit is set to 1.

4 This is repeated for every byte in the transmission.

5 At the receiving end, the number of 1s in the eight-bit code is counted.

6 If the count gives an even number, the byte is accepted.

7 This is repeated for every byte in the transmission.

If no errors are found, the transmission is accepted. However, we cannot guarantee that the transmission is error free. It is possible for two bits to be flipped in an individual byte, which would mean that the transmission is incorrect but the parity check is passed. Fortunately, this is rather unlikely so it is sensible to assume no error. The limitation of the method is that it can only detect the presence of an error. It cannot identify the actual bit that is in error. If an error is detected, re-transmission has to be requested.

An alternative approach is to use the checksum method. At the transmitting end a block is defined as a number of bytes. Then, no matter what the bytes represent, the bits in each byte are interpreted as a binary number. The sum of these binary numbers in a block is calculated and supplied as a checksum value in the transmission. This is repeated for each block. The receiver does the same calculation and checks the sum of the numbers with the checksum value transmitted for each block in turn. Once again, an error can be detected but its position in the transmission cannot be determined.

TIP

Don’t confuse a check digit with a checksum. A check digit is used for stored data; a checksum is only used for transmitted data.

Detecting the exact position of an error so as to correct it is considerably more complex. One approach is to use the parity block check method. Like the checksum method this is a longitudinal parity meaning that it is used to check a sequence of binary digits contained in a number of bytes.

WORKED EXAMPLE 9.01

Using a parity block check

At the transmitting end, a program reads a group of seven bytes as illustrated in Figure 9.01. The data is represented by seven bits for each byte. The most significant bit in each byte, bit 7, is undefined so we have left it blank.

	Seven-bit codes						
	1	0	1	0	0	1	1
	0	1	1	0	0	0	1
	1	0	1	1	0	0	0
	0	0	1	1	1	0	0
	0	1	1	0	0	1	0
	0	1	1	0	0	0	1
	0	1	1	0	0	0	1

Figure 9.01 Seven bytes to be transmitted

The parity bit is set for each of the bytes, as in Figure 9.02. The most significant bit is set to achieve even parity.

Parity bits	**Seven-bit codes**						
0	1	0	1	0	0	1	1

1	0	1	1	0	0	0	1
1	1	0	1	1	0	0	0
1	0	0	1	1	1	0	0
1	0	1	1	0	0	1	0
1	0	1	1	0	0	0	1
1	0	1	1	0	0	0	1

Figure 9.02 Bytes with the parity bit set

An additional byte is then created and each bit is set as a parity bit for the bits at that bit position. This includes counting the parity bits in the seven bytes containing data. This is illustrated in Figure 9.03.

Parity bits	**Seven-bit codes**						
0	1	0	1	0	0	1	1
1	0	1	1	0	0	0	1
1	1	0	1	1	0	0	0
1	0	0	1	1	1	0	0
1	0	1	1	0	0	1	0
1	0	1	1	0	0	0	1
1	0	1	1	0	0	0	1
0	0	0	1	0	1	0	0

← **Parity byte**

Figure 9.03 Parity byte added

The program then transmits the eight bytes in sequence.

At the receiving end, a program takes the eight bytes as input and checks the parity sums for the individual bytes and for the bit positions.

Note that the method is handling a serial transmission so it includes longitudinal checking, but the checking algorithm is working on a matrix of bit values. If there is just one error in the seven bytes this method will allow the program at the receiving end to identify the position of the error. It can therefore correct the error so the transmission can be accepted.

Question 9.02

Assume that the seven bytes shown in Figure 9.04 contain data.

01001000 01000101 01110010 01100011

00101100 01010101 00110010

Figure 9.04 Seven bytes to be transmitted

The most significant bit is set to 0 but it is undefined at this stage because a seven-bit ASCII code represents character data. Choose a parity and change the value stored in the most significant bit to match this parity for each byte. Then create the eighth byte that would be used for transmission in a parity block check method.

Question 9.03

The eight bytes shown in Figure 9.05 have been received in a transmission using the parity block method. The first seven bytes contain the data and the last byte contains the parity check bits.

01001000 11000101 11110001 01100011

01001010 01010101 01110010 01110010

Figure 9.05 Eight bytes received in a transmission

a Identify what has gone wrong during the transmission.

b What would happen after the transmission is checked?

Reflection Point:

This chapter contains a lot of terminology. It is very easy to get confused about the definitions of the different terms used. Have you considered how you are going to attempt to remember all of the definitions and not get confused?

Summary

- Important considerations for the storage of data are: data integrity, data privacy and data security.
- Data protection laws relate to data privacy.
- Security measures for computer systems include authentication of users, prevention of unauthorised access, protection from malware and methods for recovery following system failure.
- Security methods for data include backup procedures, user authorisation and access control.
- Data entry to a system should be subject to data validation and data verification.
- Verification for data transmission may be carried out using: a parity check, a checksum or a parity block check method.

Exam-style Questions

1 **a** It is important that data has integrity.

i Identify the missing word in the sentence 'Concerns about the integrity of data are concerns about its [1]

ii Validation and verification are techniques that help to ensure data integrity when data is entered into a system.

Explain the difference between validation and verification. [3]

iii Define a type of validation and give an example. [2]

iv Even after validation has been correctly applied data may lack integrity when it comes to be used. Explain why that might happen. [2]

b Data should be protected from being read by unauthorised individuals.

Explain **two** policies that can be used to provide the protection. [4]

2 **a** Security of data is an important concern for a system administrator.

i Identify **three** reasons why data might not be available when a user needs it. [3]

ii Describe what could be features of a policy for ensuring data security. [3]

b It is important for mission-critical systems that there is a disaster recovery contingency plan in place.

i Define what type of disaster is under consideration here. [2]

ii Define what will be a major feature of the contingency plan. [2]

c Measures to ensure security of a computer system need to be in place on a daily basis if the system is connected to the Internet.

Describe **two** measures that could be taken to ensure security of the system. [4]

3 **a** When data is transmitted measures need to be applied to check whether the data has been transmitted correctly.

i If data consists of seven-bit codes transmitted in bytes, describe how a simple parity check system would be used. Your account should include a description of what happens at the transmitting end and what happens at the receiving end. [5]

ii An alternative approach is to use a checksum method. Describe how this works. [3]

b For either of these two methods there are limitations as to what can be achieved by them. Identify **two** of these limitations. [2]

c A different method which does not have all of these limitations is the parity block check method.

The following diagram represents eight bytes received where the parity block method has been applied at the transmitting end. The first seven bytes contain the data and the last byte contains parity bits.

Byte 1	0	1	0	1	0	0	1	1
Byte 2	1	0	1	1	0	0	0	1
Byte 3	1	1	0	1	0	0	0	0
Byte 4	1	0	0	1	1	1	0	0
Byte 5	1	0	1	1	0	0	1	0

Byte 6	1	0	1	1	0	0	0	1
Byte 7	1	0	1	1	0	0	0	1
Byte 8	0	0	0	1	0	1	0	0

Identify the problem with this received data and what would be done with it by the program used by the receiver. [4]

4 **a** Give the definition of the terms firewall and authentication. Explain how they can help with the security of data. [3]

b Describe **two** differences between data integrity and data security. [2]

c Data integrity is required at the input stage and also during transfer of the data.

i State **two** ways of maintaining data integrity at the input stage. Use examples to help explain your answer. [3]

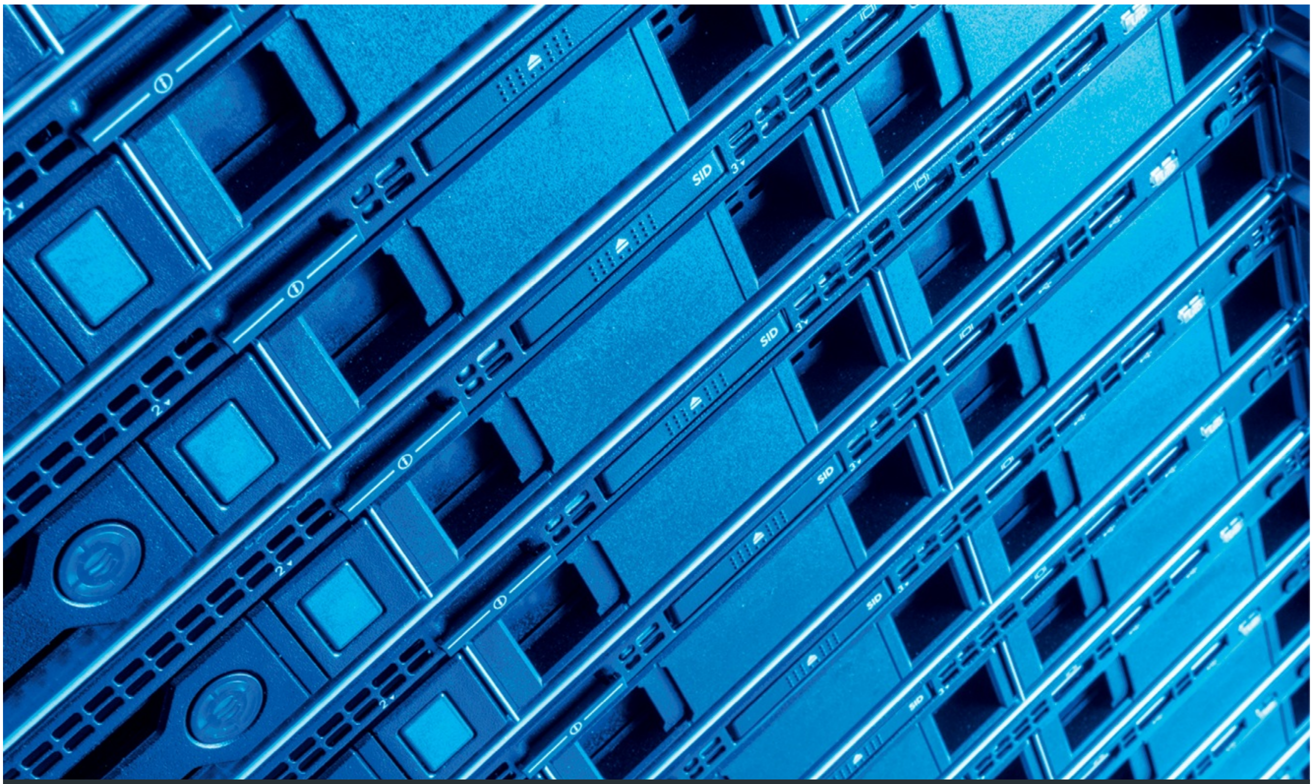
ii State **two** ways of maintaining data integrity during data transmission. Use examples to help explain your answer. [3]

Cambridge International AS & A level Computer Science 9608 paper 13 Q3 June 2015

5 Verification and validation can be applied during data entry.

Describe what is meant by these terms. For each method, explain why it is needed. [4]

Cambridge International AS & A level Computer Science 9608 paper 12 Q8 November 2015



Chapter 10:
Ethics and ownership

Learning objectives

By the end of this chapter you should be able to:

- show understanding of the need for and purpose of ethics as a computing professional
- show understanding of the need to act ethically and the impact of acting ethically or unethically for a given situation
- show understanding of the need for copyright legislation
- show understanding of the different types of software licencing and justify the use of a licence for a given situation
- show understanding of Artificial Intelligence (AI).

10.01 Ethics

You can find a number of definitions of ‘ethics’. The following three sentences are examples.

- Ethics is the field of moral science.
- Ethics are the moral principles by which any person is guided.
- Ethics are the rules of conduct recognised in a particular profession or area of human life.

In this book, we will not use the first definition. The third definition is the focus of this chapter. However, the rules of conduct of computer scientists and developers must reflect the moral principles of the second definition. Here are some observations that come to mind when considering moral principles.

Moral principles concern right or wrong. The concept of virtue is often linked to what is considered to be right. What is right and wrong can be considered from one of the following viewpoints: philosophical, religious, legal or pragmatic.

Philosophical debate has been going on for well over 2000 years. Early thinkers frequently quoted in this context are Aristotle and Confucius but there are many more. Religions have sometimes incorporated philosophies already existing or have introduced their own. Laws should reflect what is right and wrong. Pragmatism could be defined as applying common sense.

This chapter is not an appropriate place to discuss religious beliefs, but we should remember that religious beliefs do have to be considered in the working environment. Legal issues clearly impact on working practices but they are rarely the primary focus in rules of conduct. What remains as the foundation for rules of conduct are the philosophical views of right and wrong and the pragmatic views of what is common sense. These will constitute a frame of reference for what follows in this chapter.

10.02 The computing professional

No matter what their particular specialism is, any professional person is expected to act ethically. A professional can receive guidance on ethical behaviour by joining an appropriate professional organisation. Such an organisation will have a code of conduct that will include reference to ethical practice.

For example, the British Computer Society (BCS) has a code of conduct that gives guidance under four headings:

1. Public Interest
2. Professional Competence and Integrity
3. Duty to Relevant Authority
4. Duty to the Profession

The Association for Computing Machinery (ACM) and the Institute of Electrical and Electronics Engineers (IEEE) are both based in the USA but have a global perspective and global influence. The IEEE-CS/ACM Joint Task Force Software Engineering Code of Ethics defines eight principles defined as follows.

1. PUBLIC – Software engineers shall act consistently with the public interest.
2. CLIENT AND EMPLOYER – Software engineers shall act in a manner that is in the best interests of their client and employer consistent with the public interest.
3. PRODUCT – Software engineers shall ensure that their products and related modifications meet the highest professional standards possible.
4. JUDGEMENT – Software engineers shall maintain integrity and independence in their professional judgement.
5. MANAGEMENT – Software engineering managers and leaders shall subscribe to and promote an ethical approach to the management of software development and maintenance.
6. PROFESSION – Software engineers shall advance the integrity and reputation of the profession consistent with the public interest.
7. COLLEAGUES – Software engineers shall be fair to and supportive of their colleagues.
8. SELF – Software engineers shall participate in lifelong learning regarding the practice of their profession and shall promote an ethical approach to the practice of the profession.

Despite the differences in the detail the codes are consistent with regard to the following:

- the public interest or public good is a key concern
- the codes present fundamental principles
- the professional is expected to exercise their own judgement
- the professional should seek advice if unsure.

WORKED EXAMPLE 10.01

Applying ethics to a software engineering scenario

In a real-life scenario there might be many individual clauses that should be considered when a judgement is to be made. For example, let’s consider the following scenario.

You are employed by a company that develops software. You are working on a software engineering project to be delivered to a client. One day the project manager states that the project is running behind schedule. As a result, the time allocated for testing of the software will be limited to one week rather than the one month that was stated in the project plan.

As a professional you could be guided in your thinking by referring to the eight principles listed above from the IEEE-CS/ACM Joint Task Force Software Engineering Code of Ethics.

Considering them in turn your thinking might be as follows:

1. You could probably rule out any immediate need to consider public interest.
2. You would recognise that the end result might be the client being delivered a sub-standard product that would reflect badly on the reputation of your employer.
3. You would identify the primary cause of concern as being the likely poor quality product likely to be delivered.
4. You would realise that you needed to make a judgement as to what action, if any, you should take.
5. You might identify the secondary cause of concern as being one of poor management.
6. You would have some concern concerning how delivering a product with many errors would cause your profession to be judged badly but this would not be a primary concern.
7. You would be concerned about your colleagues being put under pressure to deliver in an unrealistically short timescale.
8. You would recognise that this was not an issue relating to your professional development.

You would now consider what action to take. This is where you would need to make a judgement. You might consider four possible scenarios:

1. You decide that this unprofessional behaviour by the project manager must be challenged. Following this challenge the decision is reversed.
2. You decide that this unprofessional behaviour by the project manager must be challenged. However, your protests are ignored.
3. You decide that no challenge is needed because although the testing will not be properly completed there will always be the opportunity to remedy the remaining errors in the code through routine maintenance following product installation.
4. You decide that an immediate protest would be useless but you intend to raise the matter at a later time when the errors in the product have become evident.

The first scenario is the ideal one where the appropriate ethical action leads to a fully-tested product. In the second scenario the professional has acted ethically but this has had no effect. The question is now whether anything else ought to be done. The remaining two scenarios are where unethical behaviour leaves the outcome of an unsatisfactory product being provided for the client.

Discussion Point:

Search the clauses for all eight principles of the IEEE-CS/ACM Joint Task Force Software Engineering Code of Ethics code and identify the ones that mention documentation.

Why is documentation mentioned so many times?

10.03 The public good

In different parts of the IEEE-CS/ACM Joint Task Force Software Engineering Code of Ethics code there is reference to:

- the health, safety and welfare of the public
- the public interest
- the public good
- public concern.

The BCS code has the statement that the professional should:

‘have due regard for public health, privacy, security and wellbeing of others and the environment’.

There is no further indication of how these should be interpreted. We can look at some individual cases to illustrate what might be considered.

Fortunately, there are very few examples which have involved loss of life and certainly none where large numbers of deaths were caused. However, there have been a number of incidents where extremely large sums of money were wasted because of rather simplistic errors.

The first example is the Ariane 5 rocket which exploded 40 seconds after blast-off in 1996. Approximately 500 million dollars’ worth of investment in development, scientific equipment and launch costs were wasted. The problem was caused by a line of code that tried to convert a 64-bit floating point number into a 16-bit integer. The resulting overflow crashed the program and as a result also the rocket.

The second example also relates to space exploration. The NASA Mars Climate Orbiter project centred on a space probe that was due to orbit Mars to study the climate. The probe reached Mars but unfortunately failed to get into orbit. The cause of the problem was that all of the software was supposed to use the SI system of units for all calculations. One group of software engineers used the Imperial system of units. This mismatch only caused a problem at the stage when the calculations concerned with achieving orbit around Mars were executed. This time the loss to the public purse was 125 million dollars.

These examples illustrate the public interest in successful software engineering. There is a strong argument that the correct application of the code of ethics with respect to specification, development and testing of software could have saved a lot of money.

A different type of disaster is the system that never gets built. In 2011 the UK government scrapped the National Programme for IT in the NHS (National Health Service), which had been commissioned in 2002. The project failed to produce a workable system. The estimated amount spent on the program was 12 billion pounds. The initial estimated cost was less than three billion pounds. In examples like this the software engineers are not to blame, but if correctly applied, the part of the code of ethics specifically targeted at project management would not have allowed this type of failure to occur.

In the three examples outlined above, the public concern was solely related to the costs associated with a failed project. There was no public concern relating to the ethics of the endeavour itself. In contrast there are many areas associated with computer-based systems where there is public concern about the nature of the endeavour or at least about what it has led to. Here are some examples:

- powerful commercial companies being able to exert pressure on less powerful companies to ensure that the powerful company’s products are used when alternatives might be more suitable or less costly
- companies providing systems that do not guarantee security against unauthorised access
- organisations that try to conceal information about a security breach that has occurred in their systems

- private data transmitted by individuals to other individuals being stored and made available to security services
- social media sites allowing abusive or illegal content to be transmitted
- search engines providing search results with no concern about the quality of the content.

There is by no means a consistent public attitude to concerns like this. This makes it difficult for an individual software engineer to make a judgement with respect to public good. Even if the judgement is that a company is not acting in the public good, it will always be difficult for an individual to exert any influence. There are recent examples where individuals have taken action which has resulted in their life being severely affected.

Discussion Point:

This section has deliberately been presented in generalisations. You should carry out a search for some individual examples and then consider actions that could have been taken and justified as being for the public good.

10.04 Ownership and copyright

Copyright is a formal recognition of ownership. If an individual creates and publishes some work that has an element of originality, the individual becomes the owner and can therefore claim copyright. An exception is if the individual is working for an organisation. An organisation can claim copyright for a published work if it is created by one or more individuals that work for the organisation. Copyright cannot apply to an idea and it cannot be claimed on any part of a published work that was previously published by a different individual or organisation.

Copyright can apply to any of:

- a literary (written) work
- a musical composition
- a film
- a music recording
- a radio or TV broadcast
- a work of art
- a computer program.

The justification for the existence of copyright has two components. The first is that the creation takes time and effort and requires original thinking, so the copyright holder should have the opportunity to earn money for it. The second is that it is unfair for some other individual or organisation to reproduce the work and to make money from it without any payment to the original creator.

As with the case of data protection discussed in Chapter 9 (Section 9.01), laws are needed to protect copyright. Different countries have different details in their laws but there is an international agreement that copyright laws cannot be avoided, for example by someone publishing the work in another country without the original copyright holder's permission.

Typical copyright laws will include:

- a requirement for registration recording the date of creation of the work
- a defined period when copyright will apply
- a policy to be applied if an individual holding copyright dies
- an agreed method for indicating the copyright, for example the use of the © symbol.

When copyright is in place there will be implications for how the work can be used. The copyright owner can include a statement concerning how the work might be used. For instance, the ACM has the following statement relating to the code of ethics discussed in Section 10.02:

This Code may be published without permission as long as it is not changed in any way and it carries the copyright notice. Copyright © 1999 by the Association for Computing Machinery, Inc. and the Institute for Electrical and Electronics Engineers, Inc.

This is one of several possible variations referring to permissions that are granted when the work has not been sold. If someone has bought a copy of a copyrighted product there is no restriction on copies being made provided that these are solely for the use of the individual. A general regulation relates to books in a library, where a library user can photocopy part of a book.

10.05 Software licensing

Commercial software

In one respect commercial software is no different to any other commercial product. It is created and made available for sale by a company that is aiming to make a profit. There is, however, a significant difference. If you buy a computer you become the owner but if you buy software you do not become the owner. The ownership remains with the vendor. As a buyer you have paid for an end-user licence that allows you to use the software. It is normal that the software license has to be paid for but there are a number of different options that might be available:

- A fee is paid for each individual copy of the software.
- A company might have the option of buying a site licence which allows a defined number of copies to be running at any one time.
- Special rates might be available for educational use.

A company that normally sells the software licence may sometimes provide a license free of charge. There are two possibilities. **Shareware** is commercial software which is made available on a trial basis for a limited time. It might be the full package available at the time or a limited version of it. A beta test version of new software might be considered to come in the shareware category. **Freeware** might be a limited version of a full package or possibly an earlier version. The difference is that there is no time limit for the licence.

Whatever license is obtained by the user of the software the source code will not be provided and the license will define limitations on the use of the software.

Examples of when using commercial software can be justified include:

- The software is available for immediate use and provides the functionality required
- The software has been created to be used in conjunction with already installed software
- There will be continuous maintenance and support provided
- Taking advantage of a shareware offer might allow suggestions to be made as to how the software could be improved
- Freeware can often offer sufficient functionality to serve a user’s limited needs.

Open or free licensing

For open licensing there are two major operations under way. Both are global non-profit organisations. They are very similar in what they provide but there is a difference in their underlying philosophies.

The Open Source Initiative makes **open source software** available. The philosophy here is that the use of open source software will allow collaborative development of software to take place. The software is normally made available free of charge. The source code is provided. The user of the software is free to use it, modify it, copy it or distribute it in accordance with the terms defined by the license.

The Free Software Foundation is so-named because the philosophy is that users should be free to use software in any way they wish. The software is not provided entirely free of charge; there is a small fee to cover distribution costs. The **free software** is still open source. However, there is a special feature of the license which is called ‘copyleft’. This is the condition that if the software is modified the source code for the modified version must be made available to other users under the same conditions of usage.

Examples of when using open source software can be justified include:

- The full functionality needed can be provided for at most a nominal cost
- The software could provide the required functionality with just a few modifications to the source

code

- A consortium of developers are collaborating in producing a new software suite
- The future development of the software or the continuous provision of the existing software is controlled by the user.

Discussion Point:

How often do you think that open licence software is being used? Should it be used more often?

TASK 10.01

Carry out a search to investigate some of the software available under an open licence.

10.06 Artificial intelligence (AI)

Artificial intelligence (AI) depends on and draws from many other disciplines including: philosophy, psychology, neuroscience, mathematics, linguistics and control engineering. The only definitions of AI that are acceptable are at the same time so generalised that they are not very practical. The following is a typical example:

AI concerns the use of a computer or computer-controlled device to perform tasks normally associated with intelligent behaviour by humans.

We will consider five aspects of intelligent human behaviour and discuss some applications of AI that mimic this human behaviour.

Problem solving

One example is the development of a system that can play chess. This can be considered as displaying artificial intelligence but this is only demonstrated because the rules of chess are limited. A computer with sufficient storage capacity and processing power can investigate so many options for a possible sequence of moves that a human cannot compete.

A second example is the traditional form of expert system that, for example, has been developed to aid medical diagnosis. This is supplied with data and rules from living medical experts. The expert system contains more knowledge than is possible for an individual doctor to have. However, if the expert system is given a new situation that is not covered by the data and rules it has been given, it cannot attempt a new or creative approach – unlike a human.

Linguistics

Voice recognition and voice synthesis techniques are already developed and in use. One example is if you phone a help line where you might be answered by a computer. Provided that you answer questions clearly the computer might be able to identify your needs and pass you on to an appropriate human who can help. However, this is a long way away from the computer itself creating new questions based on your answers and providing the help you need.

Perception

Traditionally robots have been used in manufacturing processes. Here the robot is programmed to perform repetitive tasks. The action of the robot each time is triggered by some mechanism. However, if anything unexpected happens the robot continues to operate as normal, regardless of any damage being caused.

There is now much research focussed on the development of autonomous robots. These have to be fitted with sensors to enable the robot to take appropriate action depending on the information received from the sensors. This is an example of perception in AI.

A development of this concept is the driverless car. There are several examples available or in development but so far they have only been able to perform limited tasks. An example is the capability for a car to park itself in a vacant parking space.

Reasoning

There are examples of the application of AI where a program has been able to draw inferences (reach conclusions based on evidence) which is a requirement for reasoning. The best examples concern the proving of mathematical theorems. Attempts have also been made to develop techniques that can verify that software that has been created does indeed correctly and fully match the documented specification.

Learning

This is currently a very active area for the application of AI techniques. Machine learning is said to take place if a system that has a task to perform is seen to improve its performance as it gains experience.

The AI system has access to ‘experience’ in the form of a massive set of data. By the use of appropriate statistical algorithms the system learns from this data.

One example is when the actions of users visiting websites to buy products are stored. The AI system then attempts to identify appropriate products to be advertised when a user returns to the website. If sales progressively increase there is evidence that learning is taking place.

Another example is the program that investigates incoming emails and makes decisions as to whether these can be classified as spam and therefore should be refused entry to the user inbox.

The impact of AI

The use of the Internet dominates the lives of a large proportion of the world’s population. Global organisations that provide the systems underpinning this user activity are collecting and storing massive amounts of data concerning how the Internet is being used. If this data is only being used to enable the organisation to increase its profits, this could be seen as normal business practice. However, if the data is not being securely stored it could get into the wrong hands and be used for criminal or subversive activity.

There are different concerns with respect to the introduction of autonomous mechanical products such as robots, robotic devices and driverless vehicles into our daily lives. There are arguments that technological developments lead to employment of more people to manufacture, service and install the new products. There is a further argument that more technology leads to less manual labour and therefore to increased leisure time. One counter argument is that more technology leads to fewer jobs because machines are doing the work. Another is that such developments simply make the rich richer and the poor poorer.

Some people are excited by the introduction of driverless vehicles, but other people believe that the potential for accidents will be increased and that there are not enough measures to prevent accidents.

Robots can be used in environments that would be dangerous for humans to enter. Giving the robot the capability to act autonomously would make it more useful in such environments.

The environmental impact of robot manufacture and disposal is probably the most significant issue. Robots are manufactured and require materials for their construction. There is only a limited supply of the raw materials needed. Also, all mechanical and electronic devices eventually end up on the scrap heap contributing to the already serious problem of waste products harming the environment and creatures living in this environment.

The use of improved expert systems to aid practising doctors and nurses is clearly a benefit. However, if these systems came to replace doctors and nurses the social consequences are difficult to predict.

Discussion Point:

Have you seen any recent information about new developments in AI?

Reflection Point:

Is there an organisation in your country for professional computer scientists? If so, does it encourage young people to join?

Summary

- There are different definitions of ethics.
- Professional organisations provide rules of conduct that include guidance on ethical behaviour.
- There is a history of software disasters that might have been prevented if good software engineering practice had been employed.
- Copyright is formal recognition of ownership.
- The use of software is controlled by a license
- Only open source software is provided with the source code which allows freedom of usage
- Artificial Intelligence is currently mainly focused on the development of autonomous mechanical products and machine learning based on access to massive data sets.

Exam-style Questions

1 **a** Complete the following sentences:

i As a computer professional your primary concern when faced with an issue should be

....

ii If an issue arises you should exercise your .... and possibly seek

....

iii You have a responsibility to act in accordance with the welfare of

....

iv You are expected to act in the interests of your .... and of your

....

v You should not accept .... for which you lack

.... [8]

b Explain **two** reasons why documentation is mentioned so often in the ACM/IEEE code of conduct. [4]

2 **a** Copyright is an important consideration when something is created.

i State what copyright primarily defines. [1]

ii When copyright is registered, some data will be recorded. Identify **two** examples of the type of data that would be recorded. [2]

iii Copyright legislation defines two conditions that will apply to the copyrighted work. Identify **one** of these. [1]

iv When copyright has been established there are options for how usage will be controlled. Give **two** alternatives for the instructions that could be included in the copyright statement for the created item. [2]

b When software is obtained there will be an associated licence defining how it can be used.

i For commercial software, describe **two** different ways in which the licence might be applied and explain the benefits to the customer of one of these. [4]

ii Define the difference between freeware and shareware. [2]

3 Identify **two** applications of artificial intelligence. For each one identify an aspect of human intelligence that the application mimics. Either explain how the application will be a benefit or explain why there would be concern about its use. [8]

4 Bobby is a senior programmer at a software house which produces intruder detection software.

He also runs his own software company which develops and sells various computer applications.

The following table shows seven activities which Bobby carries out.

Put a tick (✓) in the appropriate column to identify if the activity is ethical or unethical.

Activity	**Ethical**	**Unethical**
Gives away passwords used in the intruder detection software		
Uses source code developed at the software house for the software he develops for his own company		
Insists that staff work to deadlines		
Turns down training opportunities offered by his employer		

Writes and sells software that reads confidential data from client computers		
Fakes test results of safety-critical software		
Has the software applications developed overseas for sale in his own country		

[7]

Cambridge International AS & A level Computer Science 9608 paper 12 Q5 November 2015

5 A team of software engineers is developing a new e-commerce program for a client.

State **three** of the principles of the ACM/IEEE Software Engineering Code of Ethics. Illustrate each one, with an example, describing how it will influence their working practices. [6]

Cambridge International AS & A level Computer Science 9608 paper 11 Q6 June 2016

Chapter 11:
Databases

Learning objectives

By the end of this chapter you should be able to:

- show understanding of the limitations of using a file-based approach for the storage and retrieval of data
- describe the features of a relational database which address the limitations of a file-based approach
- show understanding of and use the terminology associated with a relational database model
- use an entity–relationship (E–R) diagram to document a database design
- show understanding of the normalisation process
- explain why a given set of database tables are, or are not, in 3NF
- produce a normalised database design for a description of a database, a given set of data, or a given set of tables
- show understanding of the features provided by a Database Management System (DBMS) that address the issues of a file based approach
- show understanding of how software tools found within a DBMS are used in practice
- show understanding that DBMS carries out all creation/modification of the database structure using its Data Definition Language (DDL)
- show understanding that the DBMS carries out all queries and maintenance of data using its DML
- show understanding that the industry standard for both DDL and DML is Structured Query Language (SQL)
- Understand given SQL (DDL) commands and be able to write simple SQL (DDL) commands using a sub-set of commands
- write an SQL script to query or modify data (DML) which are stored in (at most two) database tables.

11.01 Limitations of a file-based approach

Data integrity problems in a single file

Let’s consider a simple scenario. A theatrical agency makes bookings for bands and is setting up a computerised system. Text files are to be used. One of these text files is to store data about individual band members. Each line of the file is to contain the following data for one band member:

- Name
- Contact details
- Banking details
- Band name
- Band agent name
- Band agent contact details

The intention is that this file could be used if the agency needed to contact the band member directly or through the band’s agent. It could also be used after a gig when the band member has to be paid. Ignoring what would constitute contact details or banking details, we can look at a snapshot of some of the data that might be stored for the member’s given name, the member’s family name and the band name. The file might have a thousand or more lines of text. The following is a selection of some of the data that might be contained in various lines in the file:

```
Xiangfei   Jha        ComputerKidz

Mahesh     Ravuru     ITWizz

Dylan      Stoddart

Graham     Vandana    ITWizz

Vandana    Graham     ITWizz

Mahesh     Ravuru     ITWizz

Precious   Olsen      ComputerKidz

Precious   Olsen      ITWizz
```

It is clear that there are problems with this data. It would appear that when the data for Vandana Graham was first entered, her names were inserted in the wrong order. A later correct entry was made without deletion of the original incorrect data. This type of problem is not unique to a file-based system. There is no validation technique that could detect the original error. By contrast, validation should have led to the correction of the missing band name for Dylan Stoddart. The Precious Olsen data are examples of duplication of data and inconsistent data.

There is also possibly an error that is not evident from looking at the file contents. A band name could be entered here when that band doesn’t exist.

The above discussion shows how a file-based approach can lead to data integrity problems in an individual file. The reason is the lack of in-built control when data is entered. The database approach can prevent such problems or, at least, minimise the chances of them happening.

The data privacy issue with a single file

A different problem is a lack of data privacy. The file above was designed so that the finance section could find banking details and the recruitment section could find contact details. The problem is that there cannot be any control of access to part of a file, so for example, staff in the recruitment section would be able to access the banking details of band members. Data privacy would be properly handled by a database system.

Data redundancy and possible inconsistency in multiple files

Mindful of this privacy problem, the agency decides to store data in different files for different departments of the organisation. Table 11.01 summarises the main data to be stored in each department’s file.

Department	Data items in the section’s file				
Contract	Member names		Band name	Gig details	
Finance	Member names	Bank details		Gig details	
Publicity			Band name	Gig details	
Recruitment	Member names		Band name		Agent details

Table 11.01 Data to be held in the department files

There is now data duplication across the files. This is commonly referred to as **data redundancy**. This does not mean that the data is no longer of use. Rather, it is a recognition that once data has been stored in one file there should be no need for it to be stored again in a different file. Unfortunately, some data redundancy cannot be avoided in file-based systems. This can lead to data inconsistency, either because of errors in the original entry or because of errors in subsequent editing. This is a different cause of data lacking integrity. One of the primary aims of the database approach is the elimination of data redundancy.

Data dependency concerns

The above account has focused on the problems associated with storing the data in the files. We now need to consider the problems that might occur when programs access the files.

Traditionally a programmer wrote a program and at the same time defined the data files that the program would need. For the agency, each department would have its own programs that would access the department’s data files. When a programmer creates a program for a department, the programmer has to know how the data is organised in these files, for example, that the fourth item on a line in the file is a band name. This is an example of ‘data dependency’.

It is very likely that the files used by one department might have some data which is the same as the data in the files of other departments. However, in the scenario presented above there is no plan for file sharing.

A further issue is that the agency might decide that there is a need for a change in the data stored. For instance, they might see an increasing trend for bands to perform with additional session musicians. Their data will need to be entered into some files. This will require the existing files to be re-written. In turn, this will require the programs to be re-written so that the new files are read correctly. In a database scenario the existing programs could still be run even though additional data was added. The only programming change needed would be the writing of additional programs to use this additional data.

The other aspect of data dependency is that when file structures have been defined to suit specific programs, they may not be suited to supporting new applications. The agency might feel the need for an information system to analyse the success or otherwise of the gigs they have organised over a number of years. Extracting the data for this from the sort of file-based system described here would be a complex task that would take considerable time to complete.

11.02 The relational database

In the relational database model, each item of data is stored in a **relation** which is a special type of table. The strange choice of name comes from a mathematical theory. A relational database is a collection of relational tables.

When a table is created in a relational database it is first given a name and then the attributes are named. In a database design, a table would be given a name with the **attribute** names listed in brackets after the table name. For example, the design for a database for the theatrical agency might contain the table definitions shown in Figure 11.01.

Member(MemberID, MemberGivenName, MemberFamilyName, BandName, ...)
Band(BandName, AgentID, ...)

Figure 11.01 Two tables in a database design for the theatrical agency

A logical view of some data stored in these tables is given in Table 11.02 and Table 11.03. Each attribute is associated with one column in the table and is in effect a column header. The entries in the rows beneath this column header are attribute values.

MemberID	MemberGivenName	MemberFamilyName	BandName	...
0005	Xiangfei	Jha	ComputerKidz	...
0009	Mahesh	Ravuru	ITWizz	...
0001	Dylan	Stoddart	ComputerKidz	...
0025	Vandana	Graham	ITWizz	...

Table 11.02 Logical view of the Member table in a relational database

BandName	AgentID	...
ComputerKidz	01	...
ITWizz	07	...

Table 11.03 Logical view of the Band table in a relational database

This is described as a logical view because an underlying principle for a relational database is that there is no ordering defined for the attribute columns. At least one database product does allow a view of a table and its contents. However, this is not in keeping with the fundamental relational database concept that a query should be used to inspect the data in a table. Queries are discussed later in the chapter.

A row in a relation should be referred to as a **tuple** but this formal name is not always used. Often a row is called a ‘record’ and the attribute values ‘fields’. The tuple is the collection of data stored for one ‘instance’ of the relation. In Table 11.02, each tuple relates to one individual band member. A fundamental principle of a relational database is that a tuple is a set of ‘atomic’ values; each attribute has one value or no value.

The most important feature of the relational database concept is the **primary key**. A primary key may be a single attribute or a combination of attributes. Every table must have a primary key and each tuple in the table must have a value for the primary key and that value must be unique.

Once a table and its attributes have been defined, the next task is to choose the primary key. In some cases there may be more than one attribute for which unique values are guaranteed. In this case, each one is a **candidate key** and one will be selected as the primary key. A candidate key that is not selected as the primary key is then referred to as a **secondary key**. Often there is no candidate key and so a primary key has to be created. The design in Figure 11.01 illustrates this with the introduction of the attribute MemberID as the primary key for the Member table. Note that the primary key is underlined in the database design.

The primary key ensures integrity within the table. The DBMS will not allow an attempt to insert a value for a primary key when that value already exists. Therefore, each tuple automatically becomes unique. This is one of the features of the relational model that helps to ensure data integrity. The primary key also provides a unique reference to any attribute value that a query selects.

A database can contain stand-alone tables, but it is more usual for each table to have some relationship to another table. This relationship is implemented by using a **foreign key**.

Let's discuss the use of a foreign key using the database design shown in Figure 11.01. When the database is being created, the Band table is created first. BandName is chosen as the primary key because unique names for bands can be guaranteed. Then the Member table is created. MemberID is defined as the primary key and the attribute BandName is identified as a foreign key referencing the primary key in the Band table. Once this relationship between primary and foreign keys has been established, the DBMS will prevent any entry for BandName in the Member table being made if the corresponding value does not exist in the Band table. This provides **referential integrity** which is another reason why the relational database model helps to ensure data integrity.

Question 11.01

BandName is a primary key for the Band table. Does this mean that as a foreign key in the Member table it must have unique values? Explain your reasoning.

11.03 Entity-relationship modelling

We can use a top-down method called stepwise refinement to break down the process of database design into simple steps (see also Chapter 12, Section 12.08). At each step more detail is added to the design. In database design this approach uses an entity-relationship (E–R) diagram. Typically, this can be created either by a database designer or a systems analyst working with the designer. We introduced the term ‘relationship’ earlier in connection with the use of a foreign key. An entity (strictly speaking an entity type) could be a thing, a type of person, an event, a transaction or an organisation. Most importantly, there must be a number of ‘instances’ of the entity. An entity is something that will become a table in a relational database.

WORKED EXAMPLE 11.01

Creating an entity-relationship diagram for the theatrical agency

Let’s consider a scenario for the theatrical agency which will be sufficient to model a part of the final database they would need. The starting point for a top-down design is a statement of the requirement:

The agency needs a database to handle bookings for bands. Each band has a number of members. Each booking is for a venue. Each booking might be for one or more bands.

Step 1: Choose the entities

You look for the nouns. You ignore ‘agency’ because there is only the one. You choose Booking, Band, Member and Venue. For each of these there will be more than one instance. You are aware that each booking is for a gig at a venue but you ignore this because you think that the Booking entity will be sufficient to hold the required data about a gig.

Step 2: Identify the relationships

This requires experience, but the aim is not to define too many. You choose the following three:

- Booking with Venue
- Booking with Band
- Band with Member.

You ignore the fact that there will be, for example, a relationship between Member and Venue because you think that this will be handled through the other relationships that indirectly link them. You can now draw a preliminary E–R diagram as shown in Figure 11.02.

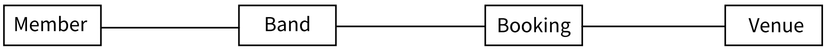


Figure 11.02 A preliminary entity–relationship diagram

Step 3: Decide the cardinalities of the relationships

Now comes the crucial stage of deciding on what are known as the ‘cardinalities’ of the relationships. At present we have a single line connecting each pair of entities. This line actually defines two relationships which might be described as the ‘forward’ one and the ‘backward’ one on the diagram as drawn. However, this only becomes apparent at the final stage of drawing the relationship. First, we have to choose one of the following descriptions for the cardinality of each relation:

- one-to-one or 1:1
- one-to-many or 1:M
- many-to-one or M:1
- many-to-many or M:M.

Let’s consider the relationship between Member and Band. We argue that one Member is a member of only one Band. (This needs to be confirmed as a fact by the agency.) We then argue that one Band has more than one Member so it has many. Therefore, the relationship between Member and Band is M:1. In its simplest form, this relationship can be drawn as shown in Figure 11.03.

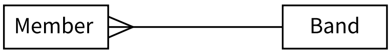


Figure 11.03 The M:1 relationship between Member and Band

This can be given more detail by including the fact that a member must belong to a Band and a Band must have more than one Member. To reflect this, the relationship can be drawn as shown in Figure 11.04.

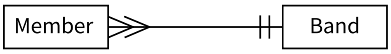


Figure 11.04 The M:1 relationship with more detail

At each end of the relationship there are two symbols. One of the symbols shows the minimum cardinality and the other the maximum cardinality. In this particular case, the minimum and maximum values just happen to be the same. However, using the diagram to document that a Member must belong to a Band is important. It indicates that when the database is created it must not be possible to create a new entry in the Member table unless there is a valid entry for BandName in that table.

For the relationship between Booking and Venue we argue that one Booking is for one Venue (there must be a venue and there cannot be more than one) and that one Venue can be used for many Bookings so the relationship between Booking and Venue is M:1. However, a Venue might exist that has so far never had a booking so the relationship can be drawn as shown in Figure 11.05.

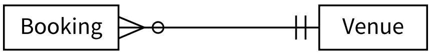


Figure 11.05 The M:1 relationship between Booking and Venue

Finally for the relationship between Band and Booking we argue that one Booking can be for many Bands and that one Band has many Bookings (hopefully!) so the relationship is M:M. However, a new band might not yet have a booking. Also, there might be only one Band for a booking so the relationship can be drawn as shown in Figure 11.06.

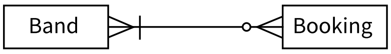


Figure 11.06 The M:M relationship between Band and Booking

Step 4: Create the full E–R diagram

At this stage we should name each relationship. The full E–R diagram for the limited scenario that has been considered is as shown in Figure 11.07.

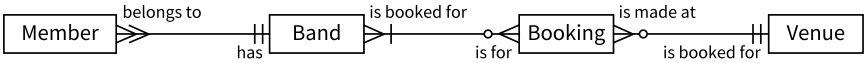


Figure 11.07 The E–R diagram for the theatrical agency’s booking database

To illustrate how the information should be read from such a diagram we can look at the part shown in Figure 11.08. Despite the fact that there is a many-to-many relationship, a reading of a relationship always considers just one entity to begin the sentence. So, reading forwards and then backwards, we say that:

One Band is booked for zero or many Bookings

One Booking is for one or many Bands

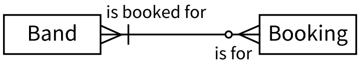


Figure 11.08 Part of the annotated E–R diagram

Question 11.02

If you are deciding on the cardinality of the relationship between two entities does it matter which one is put on the left and which on the right?

TIP

Be careful not to confuse the two completely different terms relation and relationship.

11.04 A logical entity–relationship model

A fully annotated E–R diagram of the type developed in Section 11.03 holds all of the information about the relationships that exist for the data that is to be stored in a system. It can be defined as a conceptual model because it does not relate to any specific way of implementing a system. If the system is to be implemented as a relational database, the E–R diagram has to be converted to a logical model. To do this we can start with a simplified E–R diagram that just identifies cardinalities.

If a relationship is 1:M, no further refinement is needed. The relationship shows that the entity at the many end needs to have a foreign key referencing the primary key of the entity at the one end.

If there were a 1:1 relationship there are options for implementation. However, such relationships are extremely rare and we do not need to consider them here.

The problem relationship is the M:M, where a foreign key cannot be used. A foreign key attribute can only have a single value, so it cannot handle the many references required. Another way of looking at this problem is to argue that a foreign key is required in each entity but neither table could be created first because the other table needed to exist for the foreign key to be defined. The solution for the M:M relationship is to create a link entity. For Band and Booking, the logical entity model will contain the link entity shown in Figure 11.09.

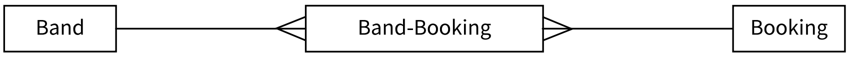


Figure 11.09 A link entity inserted to resolve a M:M relationship

Extension Question 11.01

Is it possible to annotate these relationships?

With the link entity in the model it is now possible to have two foreign keys in the link entity; one referencing the primary key of Band and one referencing the primary key of Booking.

Each entity in the logical E–R diagram will become a table in the relational database. It is therefore possible to choose primary keys and foreign keys for the tables. These can be summarised in a key table. Table 11.04 shows sensible choices for the theatrical agency’s booking database.

Table name	Primary key	Foreign key
Member	MemberID	BandName
Band	BandName	
Band-Booking	BandName & BookingID	BandName, BookingID
Booking	BookingID	VenueName
Venue	VenueName	

Table 11.04 A key table for the agency booking database

The decisions about the primary keys are determined by the uniqueness requirement. The link entity cannot use either BandName or BookingID alone but the combination of the two in a compound primary key will work.

TASK 11.01

Consider the following scenario. An organisation books cruises for passengers. Each cruise visits a number of ports. Create a conceptual E–R diagram and convert it to a logical E–R diagram. Create a key table for the database that could be implemented from the design.

11.05 Normalisation

Normalisation is a design technique for constructing a set of table designs from a list of data items. It can also be used to improve on existing table designs.

TIP

Unfortunately, you will be coming across a completely different use of the term normalisation in Chapter 16.

WORKED EXAMPLE 11.02

Normalising data for the theatrical agency

To illustrate the technique, let’s consider the document shown in Figure 11.10. This is a booking data sheet that the theatrical company might use.

Booking data sheet: 2016/023

Venue:

Cambridge International Theatre
Camside
CA1

Booking date: 23.06.2016

Bands booked	Number of band members	Headlining
ComputerKidz	5	Y
ITWizz	3	N
DeadlyDuo	2	N

Figure 11.10 Example booking data sheet

The data items on this sheet (ignoring headings) can be listed as a set of attributes:

(BookingID, VenueName, VenueAddress1, VenueAddress2, Date,
(BandName, NumberOfMembers, Headlining))

The list is put inside brackets because we are starting a process of table design. The extra set of brackets around BandName, NumberOfMembers, Headlining is because they represent a **repeating group**. If there is a repeating group, the attributes cannot sensibly be put into one relational table. A table must have single rows and atomic attribute values so the only possibility would be to include tuples such as those shown in Table 11.05. There is now data redundancy here with the duplication of the BookingID, venue data and the date.

Booking ID	Venue Name	Venue Address1	Venue Address2	Date	Band Name	Number Of Members	Headlining
2016/023	Cambridge International Theatre	Camside	CA1	23.06.2016	Computer Kidz	5	Y
2016/023	Cambridge International Theatre	Camside	CA1	23.06.2016	ITWizz	3	N
2016/023	Cambridge International	Camside	CA1	23.06.2016	DeadlyDuo	2	N

	Theatre						

Table 11.05 Data stored in an unnormalised table

Step 1: Conversion to first normal form (1NF)

The conversion to first normal form (1NF) requires splitting the data into two groups. At this stage we represent the data as table definitions. Therefore, we have to choose table names and identify a primary key for each table. One table contains the non-repeating group attributes, the other the repeating group attributes. For the first table a sensible design is:

Booking(BookingID, VenueName, VenueAddress1, VenueAddress2, Date)

The table with the repeating group is not so straightforward. It needs a compound primary key and a foreign key to give a reference to the first table. The sensible design is:

Band-Booking(BandName, BookingID(fk), NumberOfMembers, Headlining)

Again, the primary key is underlined but also the foreign key has been identified, with (fk). Because the repeating groups have been moved to a second table, these two tables could be implemented with no data redundancy in either. This is one aspect of 1NF. Also, we can say that for each table the attributes are dependent on the primary key.

Step 2: Conversion to second normal form (2NF)

For conversion to second normal form (2NF), the process is to examine each non-key attribute and ask if it is dependent on both parts of the compound key. Any attributes that are dependent on only one of the attributes in the compound key must be moved out into a new table. In this case, NumberOfMembers is only dependent on BandName. In 2NF there are now three table definitions:

Booking(BookingID, VenueName, VenueAddress1, VenueAddress2, Date)

Band-Booking(BandName(fk), BookingID(fk), Headlining)

Band(BandName, NumberOfMembers)

Note that the Booking table is unchanged from 1NF. The Booking table is automatically in 2NF; only tables with repeating group attributes have to be converted. The Band-Booking table now has two foreign keys to provide reference to data in the other two tables. The characteristics of a table in 2NF is that it either has a single primary key or it has a compound primary key with any non-key attribute dependent on both components.

Step 3: Conversion to third normal form (3NF)

For conversion to third normal form (3NF) each table has to be examined to see if there are any non-key dependencies; that means we must look for any non-key attribute that is dependent on another non-key attribute. If there is, a new table must be defined.

In our example, VenueAddress1 and VenueAddress2 are dependent on VenueName. With the addition of the fourth table we have the following 3NF definitions:

Band(BandName, NumberOfMembers)

Band-Booking(BandName(fk), BookingID(fk), Headlining)

Booking(BookingID, Date, VenueName(fk))

Venue(VenueName, VenueAddress1, VenueAddress2)

Note that once again a new foreign key has been identified to keep a reference to data in the newly created table. These four table definitions match four of the entities in the logical E–R model for which the keys were identified in Table 11.04. This will not always happen. A logical E–R diagram will describe a 2NF set of entities but not necessarily a 3NF set.

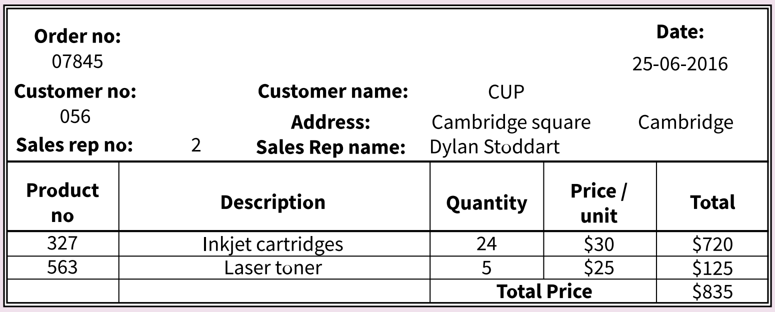
To summarise, if a set of tables are in 3NF it can be said that each non-key attribute is dependent on the key, the whole key and nothing but the key.

Question 11.03

In Step 2 of Worked Example 11.02, why is the Headlining attribute not placed in the Band table?

TASK 11.02

Normalise the data shown in Figure 11.11.



Order no: 07845

Date: 25-06-2016

Customer no: 056

Customer name: CUP

Address: Cambridge square Cambridge

Sales rep no: 2

Sales Rep name: Dylan Stoddart

Product no	Description	Quantity	Price / unit	Total
327	Inkjet cartridges	24	$30	$720
563	Laser toner	5	$25	$125
		Total Price		$835

Figure 11.11 An order form

11.06 The Database Management System (DBMS)

The database approach

It is vital to understand that a database is not just a collection of data. A database is an implementation according to the rules of a theoretical model. The basic concept was proposed some 40 years ago by ANSI (American National Standards Institute) in its three-level model. The three levels are:

- the external level
- the conceptual level
- the internal level.

The architecture is illustrated in Figure 11.12 in the context of a database to be set up for our theatrical agency.

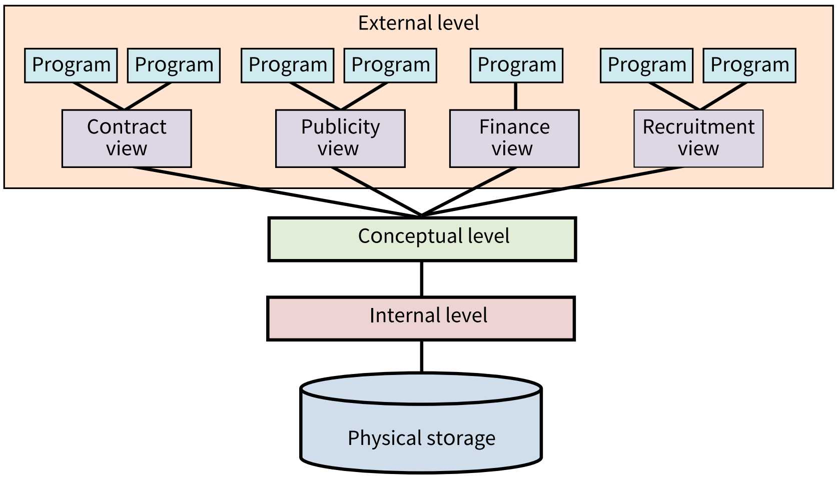


Figure 11.12 The ANSI three-level architecture for the theatrical agency database

The physical storage of the data is represented here as being on disk. The details of the storage (the internal schema) are known only at the internal level, the lowest level in the ANSI architecture. This is controlled by the **database management system (DBMS)** software.

The programmers who wrote this software are the only ones who know the structure for the storage of the data on disk. The software will accommodate any changes that might be needed in the storage medium.

At the next level, the conceptual level, there is a single universal view of the database. This is controlled by the **database administrator (DBA)** who has access to the DBMS. In the ANSI architecture the conceptual level has a conceptual schema describing the organisation of the data as perceived by a user or programmer. This may also be described as a logical schema.

At the external level there are individual user and programmer views. Each view has an external schema describing which parts of the database are accessible. A view can support a number of user programs.

An important aspect of the provision of views is that they can be used by the DBA as a mechanism for ensuring security. Individual users or groups of users can be given appropriate access rights to control what actions are allowed for that view. For example, a user may be allowed to read data but not to amend data. Alternatively, there may only be access to a limited number of the tables in the database.

The facilities provided by a DBMS

You need to remember that databases come in a variety of forms ranging from a simple system created for one individual through to the central database for some large organisation. Some of the facilities provided by a DBMS are only relevant for large organisations, when their use will be controlled by a DBA.

Whatever the size of the database, one option for its creation is to use the special-purpose language SQL which is discussed in the next section of this chapter. There are alternatives to SQL for most types of DBMS. The DBMS provides software tools through a **developer interface**. These allow for tables to be created and attributes to be defined together with their data types. In addition, the DBMS provides facilities for a programmer to develop a user interface. It also provides a **query processor** that allows a query to be created and processed. The **query** is the mechanism for extracting and manipulating data from the database. The other feature likely to be provided by the DBMS is the capability for creating a report to present formatted output. A programmer can incorporate access to queries and reports in the user interface.

DBMS functions likely to be used by a DBA

The DBA is responsible for setting up the user and programmer views and for defining the appropriate, specific access rights.

An important feature of the DBMS is the data dictionary which is part of the database that is hidden from view from everyone except the DBA. It contains metadata about the data. This includes details of all the definitions of tables, attributes and so on but also of how the physical storage is organised.

There are a number of features that can improve performance. Of special note is the capability to create an **index** for a table. This is needed if the table contains a large number of attributes and a large number of tuples. An index is a secondary table that is associated with an attribute that has unique values. The index table contains the attribute values and pointers to the corresponding tuples in the original table. The index can be on the primary key or on a secondary key. Searching an index table is much quicker than searching the full table.

The integrity of the data in the database is a key concern. One potential cause of problems occurs when a transaction is started but a system problem prevents its completion. The result would be a database in an undefined state. The DBMS should have a built-in feature that prevents this from happening. As with all systems, regular backup is a requirement. The DBA will be responsible for backup of the stored data.

Discussion Point:

How many of the above concepts are recognisable in your experience of using a database?

11.07 Structured Query Language (SQL)

SQL is the programming language provided by a DBMS to support all of the operations associated with a relational database. Even when a database package offers high-level software tools for user interaction, they create an implementation using SQL.

Data Definition Language (DDL)

Data Definition Language (DDL) is the part of SQL provided for creating or altering tables. These commands only create the structure. They do not put any data into the database.

The following are some examples of DDL that could be used in creating the database designed in Worked example 11.02 for the theatrical agency:

```
CREATE DATABASE BandBooking;
CREATE TABLE Band (
     BandName varchar(25),
     NumberOfMembers integer);
ALTER TABLE Band ADD PRIMARY KEY (BandName);
ALTER TABLE Band-Booking ADD FOREIGN KEY (BandName REFERENCES
Band(BandName);
```

These examples illustrate a number of general points regarding the writing of SQL.

- The SQL consists of a sequence of commands.
- Each command is terminated by;
- A command can occupy more than one line.
- There is no case sensitivity.
- There has been a decision made here to use upper case for the commands and lower case for table names, attribute names and datatypes.
- When a command contains a list of items these are separated by a comma.
- For the CREATE TABLE command this list is enclosed in parentheses

These examples show that once the database has been created, the tables can be created and the attributes defined. It is possible to define a primary key and a foreign key within the `CREATE TABLE` command but the `ALTER TABLE` command can be used as shown (it can also be used to add extra attributes).

When an attribute is defined, its data type must be specified. As with procedural languages there can be different data types or different names for data types depending on which DBMS is being used. One feature common to all databases is that the number of characters allowed for an attribute can be defined by including the number in brackets. In the above example `BandName varchar(25)` allows up to 25 characters for the band name.

The following list shows some of the names that might be used to define a data type: character, varchar, boolean, integer, real, date, time. In this chapter these will be written in lower case, but you might see them written in upper case in other sources.

TASK 11.03

For the database defined in Worked Example 11.02, complete the DDL for creating the four tables. Use varchar(8) for BookingID, integer for NumberOfMembers, date for Date, character for Headlining and varchar(25) for all other data.

Data Manipulation Language (DML)

There are three categories of use for Data Manipulation Language (DML)

- The insertion of data into the tables when the database is created
- The modification or removal of data in the database
- The reading of data stored in the database

The following illustrate the two possible ways that SQL can be written to populate a table with data:

```
INSERT INTO Band (‘ComputerKidz’, 5);
INSERT INTO Band-Booking (BandName, BookingID)
VALUES (‘ComputerKidz’,‘2016/023’);
```

The first example shows a simpler version that can be used if the order of the attributes is known. The second shows the safer method; the attributes are defined then the values are listed. The following are some points to note.

- Parentheses are used in both versions.
- A separate INSERT command has to be used for each tuple in the table.
- There is an order defined for the attributes.
- Although the SQL will have a list of INSERT commands the subsequent use of the table has no concept of the tuples being ordered.

The main use of DML is to obtain data from a database using a query. A query always starts with the `SELECT` command.

The simplest form for a query has the attributes for which values are to be listed as output identified after SELECT and the table name identified after FROM. For example:

```
SELECT BandName FROM Band;
```

Note that the components of the query are separated by spaces.

The Band table only has two attributes. To list the values for both there are two options:

```
SELECT BandName, NumberOfMembers
FROM Band;
```

or

```
SELECT * FROM Band;
```

which uses * to indicate all attributes. Note that in the first example the attributes are separated by commas but no parentheses are needed.

It is possible to include instructions in the SQL to control the presentation of the output. The following uses ORDER BY to ensure that the output is sorted to show the data with the band names in alphabetical order.

```
SELECT BandName, NumberOfMembers
FROM Band
ORDER BY BandName;
```

In this query there is no question of duplicate entries because BandName is the primary key of the BandName table. However, in the Band-Booking table an individual value for BandName will occur many times. If a query were being used to find which bands already had a booking there would be repeated names in the output. This can be prevented by the use of GROUP BY as shown here:

```
SELECT BandName
FROM Band-Booking
GROUP BY BandName;
```

An extension of the control of the output from a query is to include a condition to limit the selected data. This is provided by a WHERE clause. The following are examples:

```
SELECT BandName
FROM Band-Booking
WHERE Headlining = ‘Y’
GROUP BY BandName;
```

which produces a single output for each band that has headlined. Note how a query can have several component parts which are best presented on separate lines.

```
SELECT BandName, NumberOfMembers
FROM Band
WHERE NumberOfMembers > 2
ORDER BY BandName;
```

which excludes any duo bands.

It is possible to qualify the SELECT statement by using a function. SUM, COUNT and AVG are examples of functions that work on data held in several tuples for a particular attribute and return one value. For this reason, these functions are called aggregate functions. As an example, the following code displays the number of members in a band:

```
SELECT Count(*)
FROM Band;
```

This is a special case because there is no need to specify the attribute. An example using a specific attribute would be:

```
SELECT AVG(NumberOfMembers)
FROM Band;
```

another example is:

```
SELECT SUM(NumberOfMembers)
FROM Band;
```

A query can be based on a ‘join condition’ between data in two tables. The most frequently used is an inner join which is illustrated by:

```
SELECT VenueName, Date
FROM Booking
WHERE Band-Booking.BookingID = Booking.BookingID
AND Band-Booking.BandName = ‘ComputerKidz’;
```

The SQL uses the full definitive name for each attribute with the table name and attribute name separated by a dot. The query contains two conditions. The way that the query works is as follows.

- The Band-Booking table is searched for instances where the BandName is ComputerKidz.
- For each instance the BookingID is noted.
- Then there is a search of the Booking table to find the examples of tuples having this value for BookingID.
- For each one found the VenueName and Date are presented in the output.

Some versions of SQL require the explicit use of INNER JOIN. The following is a possible generic syntax:

```
SELECT table1.column1, table2.column2...
FROM table1
INNER JOIN table2
ON table1.common_field = table2.common_field;
```

The other use of DML is to modify the data stored in the database. The UPDATE command is used to change the data. If the band ComputerKidz recruited an extra member the following SQL would make the change needed.

```
UPDATE Band
SET NumberOfMembers = 6
WHERE BandName = ‘ComputerKidz’;
```

Note the use of the WHERE clause. If you forgot to include this the UPDATE command would change the number of band members to 6 for all of the bands.

The DELETE command is used to remove data from the database. This has to be done with care. If the ITWizz band decided to disband the following SQL would remove the name from the database.

```
DELETE FROM Band-Booking
WHERE BandName = ‘ITWizz’;
DELETE FROM Band
```

```
    WHERE BandName = 'ITWizz';
```

Note that if an attempt was made to carry out the deletion from Band first there would be an error. This is because BandName is a foreign key in Band-Booking. Any entry for BandName in Band-Booking must have a corresponding value in Band.

Reflection Point:

Did you find normalisation difficult? It would be surprising if you didn't. Are you going to get as much practice as possible? There are many questions from previous exam papers that contain examples to try.

Summary

- A database offers improved methods for ensuring data integrity compared to a file-based approach.
- A relational database comprises tables of a special type; each table has a primary key and may contain foreign keys.
- Entity–relationship modelling is a top-down approach to database design.
- Normalisation is a database design method that starts with a collection of attributes and converts them into first normal form then into second normal form and, finally, into third normal form.
- A database architecture provides, for the user, a conceptual level interface to the stored data.
- Features provided by a database management system (DBMS) include: a data dictionary, indexing capability, control of user access rights and backup procedures.
- Structured Query Language (SQL) includes data definition language (DDL) commands for establishing a database and data manipulation language (DML) commands for creating queries.

Exam-style Questions

1 **a** A relational database has been created to store data about subjects that students are studying. The following is a selection of some data stored in one of the tables. The data represents the student’s name, the personal tutor group, the personal tutor, the subject studied, the level of study and the subject teacher but there are some data missing:

Xiangfei	3	MUB	Computing	A	DER
Xiangfei	3	MUB	Maths	A	BNN
Xiangfei	3	MUB	Physics	AS	DAB
Mahesh	2	BAR	History	AS	IJM
Mahesh	2	BAR	Geography	AS	CAB

i Define the terms used to describe the components in a relational database table using examples from this table. [2]

ii If this represented all of the data, it would have been impossible to create this table. Identify what has not been shown here and must have been defined to allow the creation as a relational database table? Explain your answer and suggest a solution to the problem. [4]

iii Is this table in first normal form (1NF)? Explain your reason. [2]

b It has been suggested that the database design could be improved. The design suggested contains the following two tables:

Student(StudentName, TutorGroup, Tutor)

StudentSubject(StudentName, Subject, Level, SubjectTeacher)

i Identify features of this design which are characteristic of a relational database. [3]

ii Explain why the use of StudentName here is a potential problem. [2]

iii Explain why the Student table is not in third normal form (3NF). [2]

2 Consider the following scenario:

A company provides catering services for clients who need special-occasion, celebratory dinners. For each dinner, a number of dishes are to be offered. The dinner will be held at a venue. The company will provide staff to serve the meals at the venue.

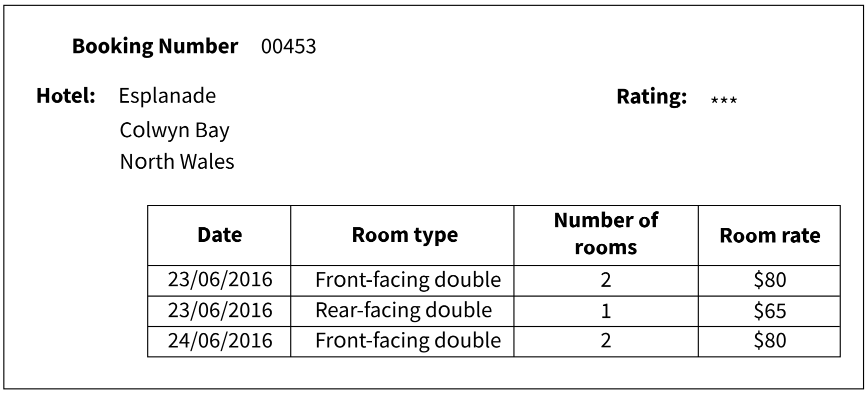
The company needs a database to store data related to this business activity.

a An entity–relationship model is to be created as the first step in a database design. Identify a list of entities. [4]

b Identify pairs of entities where there is a direct relationship between them. [4]

c For each pair of entities, draw the relationship and justify the choice of cardinality illustrated by the representation. [6]

3 Consider the following booking form used by a travel agency.



Booking Number 00453

Hotel: Esplanade
Colwyn Bay
North Wales

Rating: ***

Date	Room type	Number of rooms	Room rate
23/06/2016	Front-facing double	2	$80
23/06/2016	Rear-facing double	1	$65
24/06/2016	Front-facing double	2	$80

a Identify an unnormalised list of attributes using the data shown in this form. Make sure that you distinguish between the repeating and non-repeating attributes. [5]

b Demonstrate the conversion of the data to first normal form (1NF). The design of two tables should be defined with the keys identified. [3]

c Identify the appropriate table and demonstrate the conversion of the table to two tables in second normal form (2NF). Explain your choice of table to modify. Explain your identification of the keys for these two new tables. [5]

d Identify which part of your design is not in Third Normal Form (3NF). [2]

4 A small database is to be created with the following three tables:

STUDENT(StudentID, StudentName, StudentOtherName, DateOfbirth)

SUBJECT(SubjectName, SubjectTeacher)

TUTORIAL(StudentID(fk), Subjectname(fk), WeekNumber, Day, PeriodNumber)

a Using the appropriate datatypes from the following list:

CHARACTER, VARCHAR, BOOLEAN, INTEGER, REAL, DATE

Write the SQL scripts to create two of the tables using the CREATE TABLE command. Do not at this stage identify any keys. [5]

b Assuming that all three tables have been created, write the SQL scripts to assign the primary key in the SUBJECT table and the two foreign keys in the TUTORIAL table. [3]

c Write the SQL script that will list all of the student names in age order. [5]

d There is an aspect of the design of the tables that could cause problems. Explain this problem. [2]

5 A school stores a large amount of data. This includes student attendance, qualification and contact details. The school's software uses a file-based approach to store this data.

a The school is considering changing to a DBMS.

i State what DBMS stands for. [1]

ii Describe **two** ways in which the database Administrator (DBA) could use the DBMS software to ensure the security of the student data. [4]

iii A feature of the DBMS software is a query processor.

Describe how the school secretary could use this software. [2]

iv The DBMS has replaced software that used a file-based approach with a relational database.

Describe how using a relational database has overcome the previous problems associated with a file-based approach. [3]

b The database design has three tables to store the classes that students attend.

```
STUDENT (StudentID, FirstName, LastName, Year, TutorGroup)
CLASS (ClassID, Subject)
CLASS -GROUP (StudentID, ClassID)
```

Primary keys are not shown.

There is a one-to-many relationship between CLASS and CLASS –GROUP.

i Describe how this relationship is implemented. [2]

ii Describe the relationship between CLASS –GROUP and STUDENT. [1]

iii Write an SQL script to display the `StudentID` and `FirstName` of all students who are in the tutor group 10B. Display the list in alphabetical order of `LastName`. [4]

iv Write an SQL script to display the `LastName` of all students who attend the class whose `ClasstID` is CS1 [4]

Cambridge International AS & A level Computer Science 9608 paper 11 Q8 June 2016

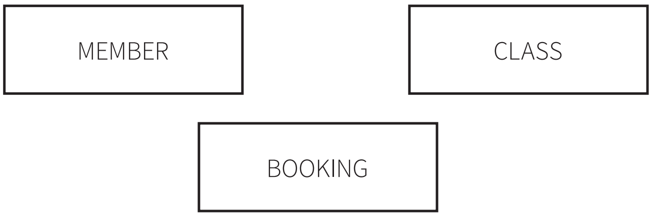
6 A health club offers classes to its members. A member needs to book into each class in advance.

a The health club employs a programmer to update the class booking system. The programmer has to decide how to store the records. The choice is between using a relational database or a file-based approach.

Give **three** reasons why the programmer should use a relational database. [6]

b The programmer decides to use three tables: `MEMBER`, `BOOKING` and `CLASS`.

Complete the entity–relationship (E–R) diagram to show the relationships between these tables.



[2]

c The `Class` table has primary key `Class ID` and stores the following data:

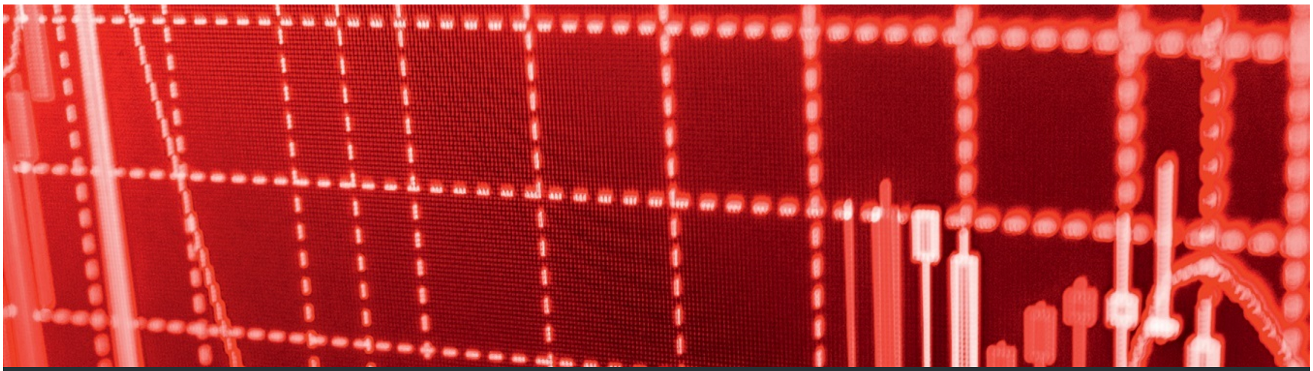
`ClassID`	`Description`	`StartDate`	`ClassTime`	`NoOfSessions`	`Adultsonly`
DAY01	Yoga beginners	12/01/2016	11:00	5	TRUE
EVE02	Yoga beginners	12/01/2016	19:00	5	FALSE
DAY16	Circuits	30/06/2016	10:30	4	FALSE

Write an SQL script to create the `CLASS` table. [6]

Cambridge international AS & A Level Computer Science 9608 paper 12 Q9 November 2016

Part 2

Fundamental problem-solving and programming skills

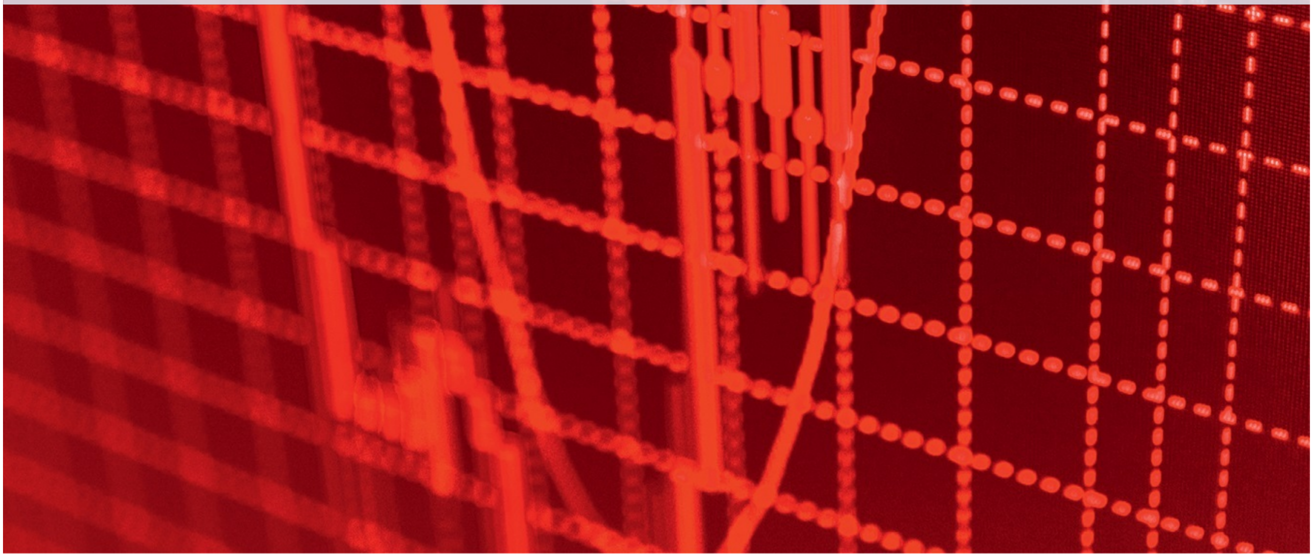


Chapter 12:
Algorithm design and problem-solving

Learning objectives

By the end of this chapter you should be able to:

- show an understanding of abstraction
- describe the purpose of abstraction
- produce an abstract model of a system by only including essential details
- describe and use decomposition
- break down problems into sub-problems leading to the concept of a program module
- show understanding that an algorithm is a solution to a problem expressed as a sequence of defined steps
- use suitable identifier names for the representation of data used by a problem and represent these using an identifier table
- write pseudocode that contains input, process and output
- write pseudocode using the four basic constructs of assignment, sequence, selection and repetition
- document a simple algorithm using pseudocode
- write pseudocode from a structured English description or a flowchart
- describe and use the process of stepwise refinement to express an algorithm to a level of detail from which the task may be programmed
- use logic statements to define parts of an algorithm solution.



12.01 What is computational thinking?

Computational thinking is a problem-solving process where a number of steps are taken in order to reach a solution. It is a logical approach to analysing a problem, producing a solution that can be understood by humans and used by computers.

Computational thinking involves five key strands: abstraction, decomposition, data modelling, pattern recognition and algorithmic thinking.

Abstraction

Abstraction involves filtering out information that is not necessary to solve a problem. There are many examples in everyday life where abstraction is used. Figure 12.02 shows part of the underground map of London, UK. The purpose of this map is to help people plan their journey within London. The map does not show a geographical representation of the tracks of the underground train network nor does it show the streets above ground. It shows the stations and which train lines connect the stations. In other words, the information that is not necessary when planning how to get from one landmark to another is filtered out. The essential information we need to be able to plan our route is clearly represented.

Abstraction gives us the power to deal with complexity. An algorithm is an abstraction of a process that takes inputs, executes a sequence of steps, and produces outputs. An abstract data type defines an abstract set of values and operations for manipulating those values.

TASK 12.01

Use the aerial photograph in Figure 12.01 and draw a map just showing the essential details for finding a route from landmark A to landmark B.

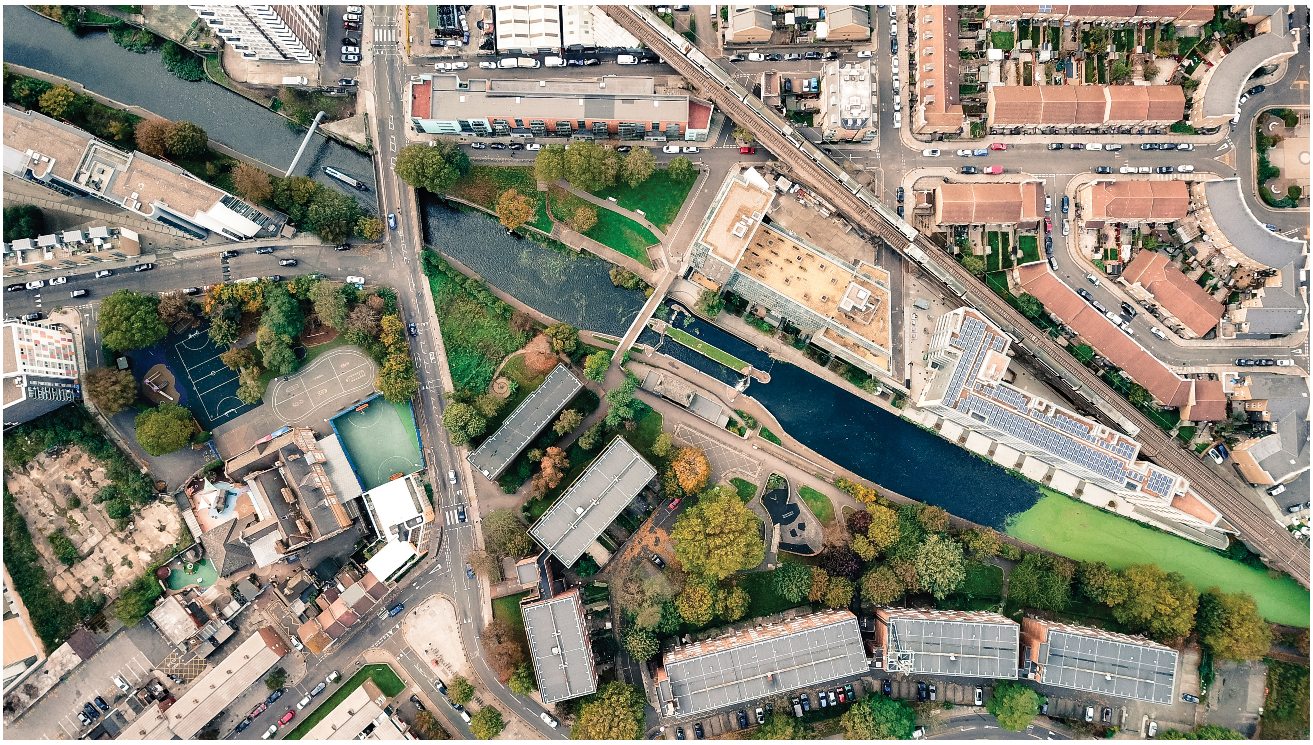


Figure 12.01 Aerial photograph of part of a city

Decomposition

Decomposition means breaking problems down into sub-problems in order to explain a process more clearly. Decomposition leads us to the concept of program modules and using procedures and functions.

Data modelling

Data modelling involves analysing and organising data (see Chapter 13). We can set up abstract data types to model real-world concepts, such as queues or stacks. When a programming language does not have such data types built-in, we can define our own by building them from existing data types. There are more ways to build data models. In Chapter 27 we cover object-oriented programming where we build data models by defining classes. In Chapter 29 we model data using facts and rules. In Chapter 26 we cover random files.

Pattern recognition

Pattern recognition means looking for patterns or common solutions to common problems and using these to complete tasks in a more efficient and effective way. There are many standard algorithms to solve standard problems, such as sorting (see Section 13.03 and 23.03) or searching (see Section 13.02 and 23.04).

Algorithm design

Algorithm design involves developing step-by-step instructions to solve a problem.

12.02 What is an algorithm?

We use algorithms in everyday life. If you need to change a wheel on a car, you might need to follow instructions (the algorithm) from a manual.

1. Take a spanner and loosen the wheel nuts.
2. Position a jack in an appropriate place.
3. Raise the car.
4. Take off the wheel nuts and the wheel.
5. Lift replacement wheel into position.
6. Replace wheel nuts and tighten by hand.
7. Lower the car.
8. Fully tighten wheel nuts.

This might sound all very straightforward. However, if the instructions are not followed in the correct logical sequence, the process might become much more difficult or even impossible. For example, if you tried to do Step 1 after Step 3, the wheel may spin and you can't loosen the wheel nuts. You can't do Step 4 before Step 3.

If you want to bake a cake, you follow a recipe.

1. Measure the following ingredients: 200g sugar, 200g butter, 4 eggs, 200g flour, 2 teaspoons baking powder and 2 tablespoons of milk.
2. Mix the ingredients together in a large bowl, until the consistency of the mixture is smooth.
3. Pour the mixture into a cake tin.
4. Bake in the oven at 190° C for 20 minutes.
5. Check it is fully cooked.
6. Turn cake out of the tin and cool on a wire rack.

The recipe is an algorithm. The ingredients are the input and the cake is the output. The process is mixing the ingredients and cooking the mixture in the oven.

Sometimes a step might need breaking down into smaller steps. For example, Step 2 can be more detailed.

2.1 Beat the sugar and butter together until fluffy.

2.2 Add the eggs, one at a time, mixing constantly.

2.3 Sieve the flour and baking powder and stir slowly into the egg mixture.

2.4 Add milk and mix to give a creamy consistency.

Sometimes there might be different steps depending on some other conditions. For example, consider how to get from one place to another using the map of the London Underground system in Figure 12.02.

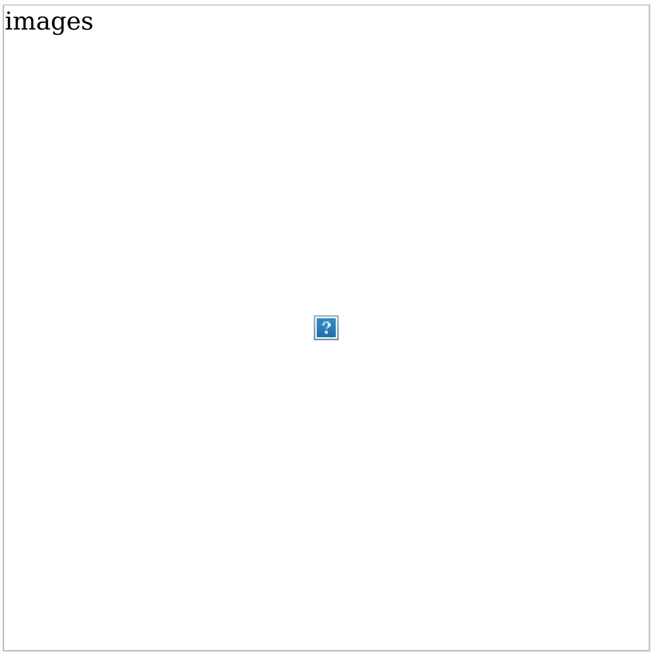


Figure 12.02 Underground map of London, UK

To travel from King’s Cross St. Pancras to Westminster, we consider two routes:

- Route A: Take the Victoria Line to Green Park (4 stations); then take the Jubilee Line to Westminster (1 station)
- Route B: Take the Piccadilly Line to Green Park (6 stations); then take the Jubilee Line to Westminster (1 station).

Route A looks like the best route. If there are engineering works on the Victoria Line and trains are delayed, Route B might turn out to be the quicker route.

The directions on how to get from King’s Cross St. Pancras to Westminster can be written as:

```
IF there are engineering works on the Victoria Line
   THEN
     Take the Piccadilly Line to Green Park (6 stations)
     Take the Jubilee Line to Westminster (1 station)
   ELSE
     Take the Victoria Line to Green Park (4 stations)
     Take the Jubilee Line to Westminster (1 station)
```

TASK 12.02

Write the steps to be followed to:

- make a sandwich
- walk from your school/college to the nearest shop
- log on to your computer.

Many problems have more than one solution. Sometimes it is a personal preference which solution to choose. Sometimes one solution will be measurably better than another.

12.03 Expressing algorithms

TIP

Computer scientists are interested in finding good solutions. A good solution gives the correct results, takes up as little computer memory as possible and executes as fast as possible. The solution should be concise, elegant and easy to understand.

In computer science, when we design a solution to a problem we express the solution (the algorithm) using sequences of steps written in **structured English** or **pseudocode**. Structured English is a subset of the English language and consists of command statements. Pseudocode resembles a programming language without following the syntax of a particular programming language. A **flowchart** is an alternative method of representing an algorithm. A flowchart consists of specific shapes, linked together.

An algorithm consists of a sequence of steps. Under certain conditions we may wish not to perform some steps. We may wish to repeat a number of steps. In computer science, when writing algorithms, we use four basic types of construct.

- **Assignment:** a value is given a name (identifier) or the value associated with a given identifier is changed.
- **Sequence:** a number of steps are performed, one after the other.
- **Selection:** under certain conditions some steps are performed, otherwise different (or no) steps are performed.
- **Repetition:** a sequence of steps is performed a number of times. This is also known as iteration or looping.

Many problems we try to solve with a computer involve data. The solution involves inputting data to the computer, processing the data and outputting results (as shown in Figure 12.03).

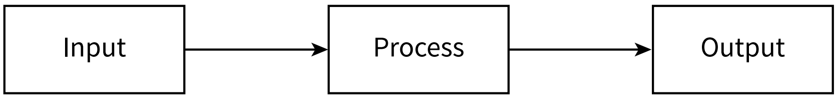


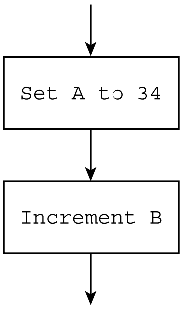
Figure 12.03 Input–process–output

We therefore also need input and output statements.

We need to know the constructs so we know how detailed our design has to be. These constructs are represented in each of the three notations as shown in Table 12.01.

In this book, algorithms and program code are typed using the `Courier` font.

	Structured English	Pseudocode	Flowchart

Assignment and Sequence	`SET A TO 34`
	`INCREMENT B`
		`A ← 34`
		`B ← B + 1`
			

Selection	`IF A IS GREATER THAN B`
	`THEN ...`
	`ELSE ...`
		`IF A > B`
		`THEN ...`
		`ELSE ...`
		`ENDIF`	

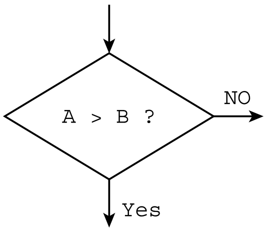
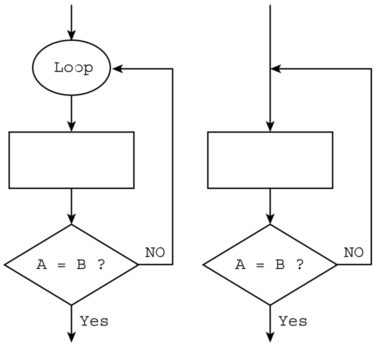
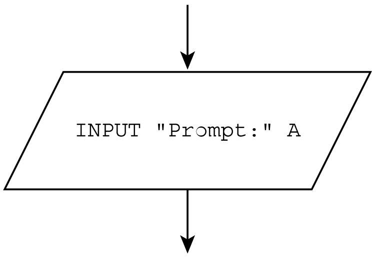
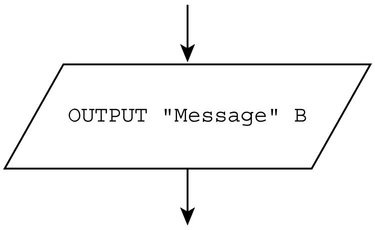
			
Repetition	REPEAT UNTIL A IS EQUAL TO B ...	REPEAT ... UNTIL A = B	Alternative construct: 
Input	INPUT A	INPUT "Prompt: " A	
Output	OUTPUT "Message" OUTPUT B	OUTPUT "Message", B	

Table 12.01 Constructs for computing algorithms

12.04 Variables

When we input data for a process, individual values need to be stored in memory. We need to be able to refer to a specific memory location so that we can write statements of what to do with the value stored there. We refer to these named memory locations as **variables**. You can imagine these variables like boxes with name labels on them. When a value is input, it is stored in the box with the specified name (identifier) on it.

For example, the variable used to store a count of how many guesses have been made in a number guessing game might be given the identifier `NumberOfGuesses` and the player's name might be stored in a variable called `ThisPlayer`, as shown in Figure 12.04.

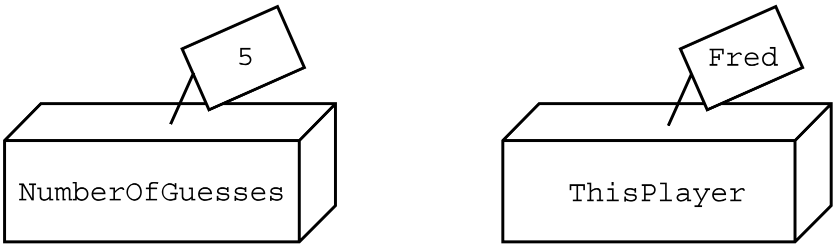


Figure 12.04 Variables

Variable identifiers should not contain spaces, only letters, digits and _ (the underscore symbol). To make algorithms easier to understand, the naming of a variable should reflect the variable's use. This means often that more than one word is used as an identifier. The formatting convention used here is known as CamelCaps. It makes an identifier easier to read.

12.05 Assignments

Assigning a value

The following pseudocode stores the value that is input (for example 15) in a variable with the identifier `Number` (see Figure 12.05(a)).

```
INPUT Number
```

The following pseudocode stores the value 1 in the variable with the identifier `NumberOfGuesses` (see Figure 12.05(b)).

```
NumberOfGuesses ← 1
```

15
Number
a
1
NumberOfGuesses
b

Figure 12.05 Variables being assigned a value

Updating a value

The following pseudocode takes the value stored in `NumberOfGuesses` (see Figure 12.06 (a)), adds 1 to that value and then stores the new value back into the variable `NumberOfGuesses` (see Figure 12.06 (b)).

```
NumberOfGuesses ← NumberOfGuesses + 1
```

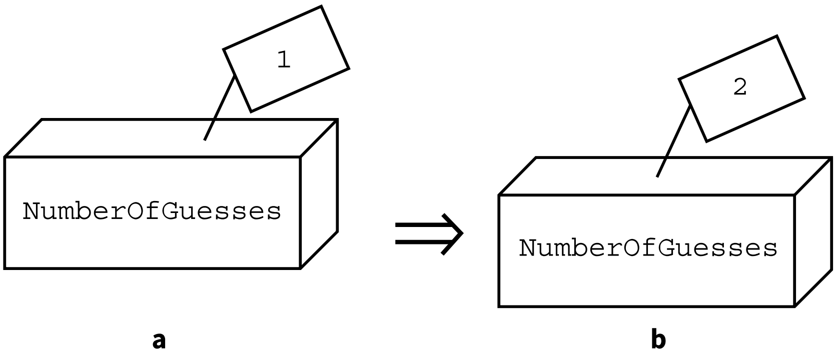


Figure 12.06 Updating the value of a variable

Copying a value

Values can be copied from one variable to another.

The following pseudocode takes the value stored in `Value1` and copies it to `Value2` (see Figure 12.07).

```
    Value2 ← Value1
```

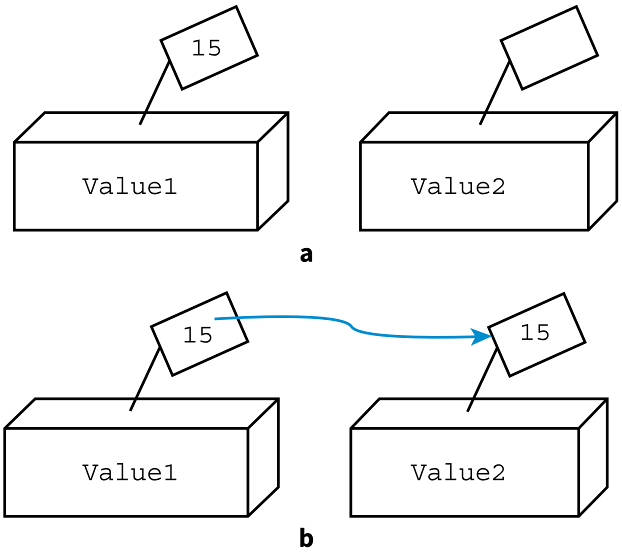


Figure 12.07 Copying the value of a variable

The value in `Value1` remains the same until it is assigned a different value.

Swapping two values

If we want to swap the contents of two variables, we need to store one of the values in another variable temporarily. Otherwise the second value to be moved will be overwritten by the first value to be moved.

In Figure 12.08(a), we copy the content from `Value1` into a temporary variable called `Temp`. Then we copy the content from `Value2` into `Value1` Figure 12.08(b)). Finally, we can copy the value from `Temp` into `Value2` (Figure 12.08(c)).

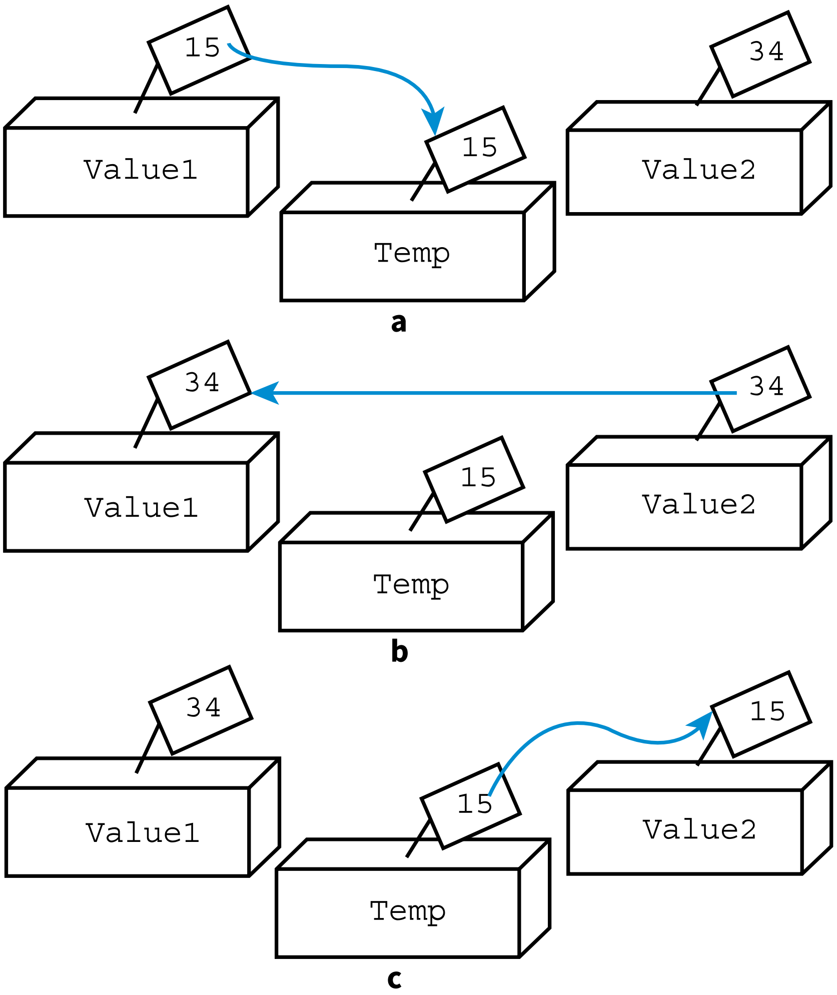


Figure 12.08 Swapping the values of two variables

Using pseudocode we write:

```
Temp ← Value1
Value1 ← Value2
Value2 ← Temp
```

WORKED EXAMPLE 12.01

Using input, output, assignment and sequence constructs

The problem to be solved: Convert a distance in miles and output the equivalent distance in km.

Step 1: Write the problem as a series of structured English statements:

```
INPUT number of miles
Calculate number of km
OUTPUT calculated result as km
```

Step 2: Analyse the data values that are needed.

We need a variable to store the original distance in miles and a variable to store the result of multiplying the number of miles by 1.61. It is helpful to construct an **identifier table** to list the variables.

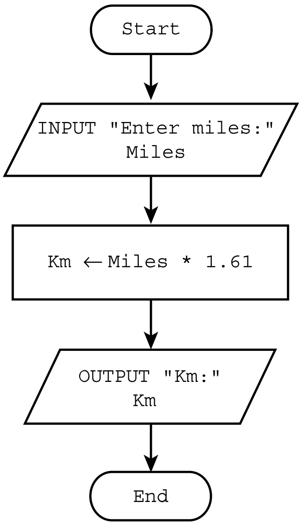
Identifier	Explanation
`Miles`	Distance as a whole number of miles
`Km`	The result from using the given formula: Km = Miles * 1.61

Table 12.02 Identifier table for miles to km conversion

Step 3: Provide more detail by drawing a flowchart or writing pseudocode.

The detail given in a flowchart should be the same as the detail given in pseudocode. It should use the basic constructs listed in Table 12.01.

Figure 12.09 represents our algorithm using a flowchart and the equivalent pseudocode.



```
INPUT "Enter miles:" Miles
Km ←Miles * 1.61
OUTPUT "km:", Km
```

Figure 12.09 Flowchart and pseudocode for miles to km conversion

TASK 12.03

Consider the following algorithm steps.

1 Input a length in inches.

2 Calculate the equivalent in centimetres.

3 Output the result.

List the variables required in an identifier table.

Write pseudocode for the algorithm.

12.06 Logic statements

In Section 12.02, we looked at an algorithm with different steps depending on some other condition:

```
IF there are engineering works on the Victoria Line
   THEN
     Take the Piccadilly Line to Green Park (6 stations)
     Take the Jubilee Line to Westminster (1 station)
   ELSE
     Take the Victoria Line to Green Park (4 stations)
     Take the Jubilee Line to Westminster (1 station)
```

The selection construct in Table 12.01 uses a condition to follow either the first group of steps or the second group of steps (see Figure 12.10).

A condition consists of at least one logic proposition (see Chapter 4, Section 4.01). Logic propositions use the relational (comparison) operators shown in Table 12.03.

Simple condition

```
IF A < B
  THEN
    <statement(s)>
  ELSE
    <statement(s)>
ENDIF
```

Figure 12.10 Pseudocode for the selection construct

Operator	Comparison
=	Is equal to
<	Is less than
>	Is greater than
<=	Is less than or equal to
>=	Is greater than or equal to
<>	Is not equal to

Table 12.03 Relational operators

Conditions are either TRUE or FALSE. In pseudocode, we distinguish between the relational operator = (which tests for equality) and the assignment symbol ←.

A person is classed as a child if they are under 13 and as an adult if they are over 19. If they are between 13 and 19 inclusive they are classed as teenagers. We can write these statements as logic statements.

- If Age < 13 then person is a child.
- If Age > 19 then person is an adult.
- If Age >= 13 AND Age <= 19 then person is a teenager.

TASK 12.04

A town has a bus service where passengers under the age of 12 and over the age of 60 do not need to pay a fare. Write the logic statements for free fares.

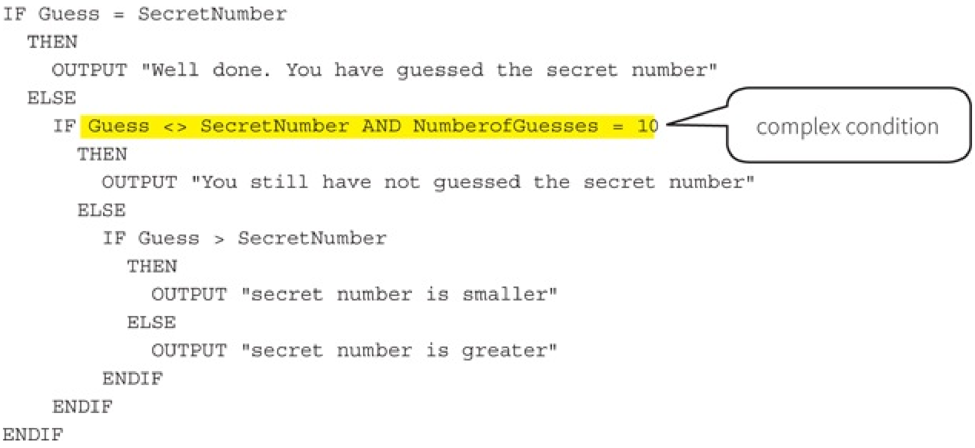
A number-guessing game follows different steps depending on certain conditions. Here is a description of the algorithm.

- The player inputs a number to guess the secret number stored.
- If the guess was correct, output a congratulations message.
- If the number input was larger than the secret number, output message “secret number is smaller”.
- If the number input was smaller than the secret number, output message “secret number is greater”.

We can re-write the number-guessing game steps as an algorithm in pseudocode:

```
SET value for secret number
INPUT Guess
IF Guess = SecretNumber
   THEN
      OUTPUT "Well done. You have guessed the secret number"
   ELSE
      IF Guess > SecretNumber
        THEN
          OUTPUT "secret number is smaller"
        ELSE
          OUTPUT "secret number is greater"
      ENDIF
ENDIF
```

More complex conditions can be formed by using the logical operators AND, OR and NOT. For example, the number-guessing game might allow the player multiple guesses; if the player has not guessed the secret number after 10 guesses, a different message is output.



```
IF Guess = SecretNumber
  THEN
    OUTPUT "Well done. You have guessed the secret number"
  ELSE
    IF Guess <> SecretNumber AND NumberofGuesses = 10
      THEN
        OUTPUT "You still have not guessed the secret number"
      ELSE
        IF Guess > SecretNumber
          THEN
            OUTPUT "secret number is smaller"
          ELSE
            OUTPUT "secret number is greater"
        ENDIF
    ENDIF
ENDIF
```

complex condition

WORKED EXAMPLE 12.02

Using selection constructs

The problem to be solved: Take three numbers as input and output the largest number.

There are several different methods (algorithms) to solve this problem. Here is one method.

1. Input all three numbers at the beginning.
2. Store each of the input values in a separate variable (the identifiers are shown in Table 12.04).
3. Compare the first number with the second number and then compare the bigger one of these with the third number.
4. The bigger number of this second comparison is output.

See Worked Example 12.03 for another solution.

Identifier	Explanation

`Number1`	The first number to be input
`Number2`	The second number to be input
`Number3`	The third number to be input

Table 12.04 Identifier table for biggest number problem

The algorithm can be expressed in the following pseudocode:

```
INPUT Number1
INPUT Number2
INPUT Number3
IF Number1 > Number2
  THEN
  // Number1 is bigger
    IF Number1 > Number3
      THEN
        OUTPUT Number1
      ELSE
        OUTPUT Number3
    ENDIF
  ELSE
  // Number2 is bigger
    IF Number2 > Number3
      THEN
        OUTPUT Number2
      ELSE
        OUTPUT Number3
    ENDIF
ENDIF
```

When an `IF` statement contains another `IF` statement, we refer to these as **nested `IF` statements**.

Question 12.01

What changes do you need to make to output the smallest number?

WORKED EXAMPLE 12.03

Using selection constructs (alternative method)

The problem to be solved: Take three numbers as input and output the largest number.

This is an alternative method to Worked Example 12.02.

1. Input the first number and store it in `BiggestSoFar`
2. Input the second number and compare it with the value in `BiggestSoFar`.
3. If the second number is bigger, assign its value to `BiggestSoFar`
4. Input the third number and compare it with the value in `BiggestSoFar`
5. If the third number is bigger, assign its value to `BiggestSoFar`
6. The value stored in `BiggestSoFar` is output.

The identifiers required for this solution are shown in Table 12.05.

Identifier	Explanation
`BiggestSoFar`	Stores the biggest number input so far

`NextNumber`	The next number to be input

Table 12.05 Identifier table for the alternative solution to the biggest number problem

The algorithm can be expressed in the following pseudocode:

```
INPUT BiggestSoFar
INPUT NextNumber
IF NextNumber > BiggestSoFar
   THEN
      BiggestSoFar ← NextNumber
ENDIF
INPUT NextNumber
IF NextNumber > BiggestSoFar
   THEN
      BiggestSoFar ← NextNumber
ENDIF
OUTPUT BiggestSoFar
```

Note that when we input the third number in this method the second number gets overwritten as it is no longer needed.

There are several advantages of using the method in Worked Example 12.03 compared to the method in Worked Example 12.02.

- Only two variables are used.
- The conditional statements are not nested and do not have an ELSE part. This makes them easier to understand.
- This algorithm can be adapted more easily if further numbers are to be compared (see Worked Example 12.04).

The disadvantage of the method in Worked Example 12.03 compared to the method in Worked Example 12.02 is that there is more work involved with this algorithm. If the second number is bigger than the first number, the value of `BiggestSoFar` has to be changed. If the third number is bigger than the value in `BiggestSoFar` then the value of `BiggestSoFar` has to be changed again. Depending on the input values, this could result in two extra assignment instructions being carried out.

12.07 Loops

Look at the pseudocode algorithm in Worked Example 12.03. The two IF statements are identical. To compare 10 numbers, we would need to write this statement nine times. Moreover, if the problem changed to having to compare, for example, 100 numbers, our algorithm would become very tedious. If we use a repetition construct (a loop) we can avoid writing the same lines of pseudocode over and over again.

WORKED EXAMPLE 12.04

Repetition using REPEAT...UNTIL

The problem to be solved: Take 10 numbers as input and output the largest number.

We need one further variable to store a counter, so that we know when we have compared 10 numbers.

Identifier	Explanation
BiggestSoFar	Stores the biggest number input so far
NextNumber	The next number to be input
Counter	Stores how many numbers have been input so far

Table 12.06 Identifier table for the biggest number problem using REPEAT...UNTIL

The algorithm can be expressed in the following pseudocode:

```
INPUT BiggestSoFar
Counter ← 1
REPEAT
    INPUT NextNumber
    Counter ← Counter + 1
    IF NextNumber > BiggestSoFar
      THEN
        BiggestSoFar ← NextNumber
    ENDIF
UNTIL Counter = 10
OUTPUT BiggestSoFar
```

Note that when we input the next number in this method the previous number gets overwritten as it is no longer needed.

Question 12.02

What changes do you need to make to the algorithm in Worked Example 12.04:

a to compare 100 numbers?

b to take as a first input the number of numbers to be compared?

There is another loop construct that does the counting for us: the FOR...NEXT loop.

WORKED EXAMPLE 12.05

Repetition using FOR...NEXT

The problem to be solved: Take 10 numbers as input and output the largest number.

We can use the same identifiers as in Worked Example 12.04. Note that the purpose of Counter has changed.

Identifier	Explanation
BiggestSoFar	Stores the biggest number input so far
NextNumber	The next number to be input
Counter	Counts the number of times round the loop

Table 12.07 Identifier table for biggest number problem using a FOR loop

The algorithm can be expressed in the following pseudocode:

```
INPUT BiggestSoFar
FOR Counter ← 2 TO 10
      INPUT NextNumber
      IF NextNumber > BiggestSoFar
        THEN
          BiggestSoFar ← NextNumber
      ENDIF
NEXT Counter
OUTPUT BiggestSoFar
```

The first time round the loop, Counter is set to 2. The next time round the loop, Counter has automatically increased to 3, and so on. The last time round the loop, Counter has the value 10.

A **rogue value** is a value used to terminate a sequence of values. The rogue value is of the same data type but outside the range of normal expected values.

WORKED EXAMPLE 12.06

Repetition using a rogue value

The problem to be solved: A sequence of non-zero numbers is terminated by 0. Take this sequence as input and output the largest number.

Note: In this example the rogue value chosen is 0. It is very important to choose a rogue value that is of the same data type but outside the range of normal expected values. For example, if the input might normally include 0 then a negative value, such as −1, might be chosen.

Look at Worked Example 12.05. Instead of counting the numbers input, we need to check whether the number input is 0 to terminate the loop. The identifiers are shown in Table 12.08.

Identifier	Explanation
BiggestSoFar	Stores the biggest number input so far
NextNumber	The next number to be input

Table 12.08 Identifier table for biggest number problem using a rogue value

A possible pseudocode algorithm is:

```
INPUT BiggestSoFar
REPEAT
    INPUT NextNumber
    IF NextNumber > BiggestSoFar
      THEN
        BiggestSoFar ← NextNumber
    ENDIF
UNTIL NextNumber = 0
OUTPUT BiggestSoFar
```

This algorithm works even if the sequence consists of only one non-zero input. However, it will not work if the only input is 0. In that case, we don't want to perform the statements within the loop at all. We can use an alternative construct, the `WHILE...ENDWHILE` loop.

```
INPUT NextNumber
BiggestSoFar ← NextNumber
WHILE NextNumber <> 0 DO // sequence terminator not encountered
    INPUT NextNumber
    IF NextNumber > BiggestSoFar
      THEN
        BiggestSoFar ← NextNumber
    ENDIF
ENDWHILE
OUTPUT BiggestSoFar
```

Before we enter the loop, we check whether we have a non-zero number. To make this work for the first number, we store it in `NextNumber` and also in `BiggestSoFar`. If this first number is zero, we don't follow the instructions within the loop. For a non-zero first number this algorithm has the same effect as the algorithm using `REPEAT...UNTIL`.

WORKED EXAMPLE 12.07

Implementing the number-guessing game with a loop

Consider the number-guessing game again, this time allowing repeated guesses.

1. The player repeatedly inputs a number to guess the secret number stored.
2. If the guess is correct, the number of guesses made is output and the game stops.
3. If the number input is larger than the secret number, the player is given the message to input a smaller number.
4. If the number input is smaller than the secret number, the player is given the message to input a larger number.

The algorithm is expressed in structured English, as a flowchart and in pseudocode.

Algorithm for the number-guessing game in structured English:

```
SET value for secret number
REPEAT the following UNTIL correct guess
   INPUT guess
   count number of guesses
   COMPARE guess with secret number
   OUTPUT comment
OUTPUT number of guesses
```

We need variables to store the following values:

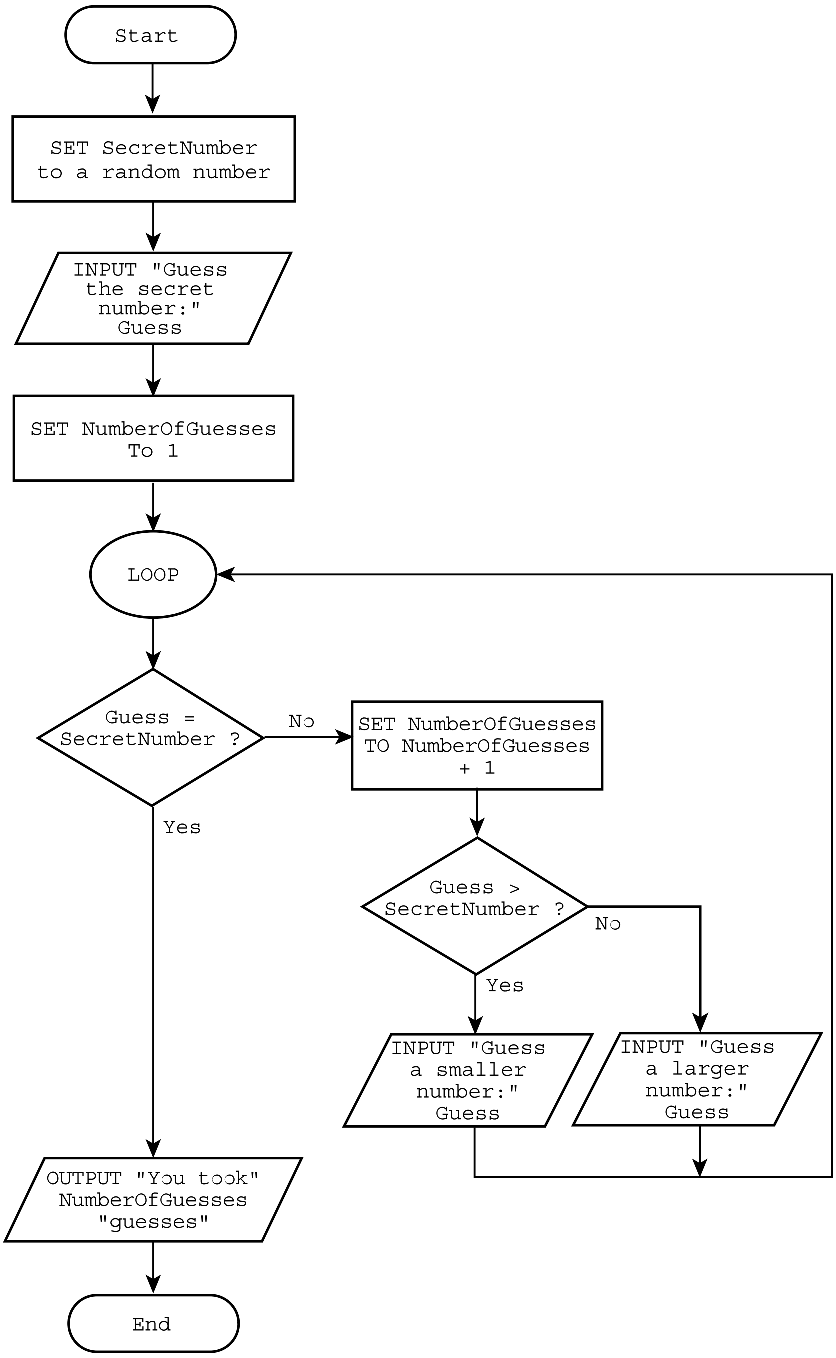
- the secret number (to be set as a random number)
- the number input by the player as a guess
- the count of how many guesses the player has made so far.

We represent this information in the identifier table shown in Table 12.09.

Identifier	Explanation
`SecretNumber`	The number to be guessed
`NumberOfGuesses`	The number of guesses the player has made
`Guess`	The number the player has input as a guess

Table 12.09 Identifier table for number-guessing game

Algorithm for the number-guessing game as a flowchart


Start
SET SecretNumber to a random number
INPUT "Guess the secret number:" Guess
SET NumberOfGuesses To 1
LOOP
Guess = SecretNumber ?
No
SET NumberOfGuesses TO NumberOfGuesses + 1
Guess > SecretNumber ?
No
Yes
INPUT "Guess a smaller number:" Guess
INPUT "Guess a larger number:" Guess
Yes
OUTPUT "You took" NumberOfGuesses "guesses"
End

Pseudocode for the number-guessing game with a post-condition loop

```
SecretNumber ← Random
NumberOfGuesses ← 0
REPEAT
    INPUT Guess
    NumberOfGuesses ← NumberOfGuesses + 1
    IF Guess > SecretNumber
      THEN
        // the player is given the message to input a smaller number
    ENDIF
    IF Guess < SecretNumber
      THEN
        // the player is given the message to input a larger number
    ENDIF
UNTIL Guess = SecretNumber
OUTPUT NumberOfGuesses
```

Pseudocode for the number-guessing game with a pre-condition loop

```
SecretNumber ← Random
INPUT Guess
NumberOfGuesses ← 1
WHILE Guess <> SecretNumber DO
    IF Guess > SecretNumber
      THEN
        // the player is given the message to input a smaller number
    ENDIF
    IF Guess < SecretNumber
      THEN
        // the player is given the message to input a larger number
    ENDIF
    INPUT Guess
    NumberOfGuesses ← NumberOfGuesses + 1
ENDWHILE
OUTPUT NumberOfGuesses
```

WORKED EXAMPLE 12.08

Calculating running totals and averages

The problem to be solved: Take 10 numbers as input and output the sum of these numbers and the average.

Identifier	Explanation
RunningTotal	Stores the sum of the numbers input so far
Counter	How many numbers have been input
NextNumber	The next number input
Average	The average of the numbers input

Table 12.10 Identifier table for running total and average algorithm

The following pseudocode gives a possible algorithm:

```
RunningTotal ← 0
FOR Counter ← 1 TO 10
      INPUT NextNumber
      RunningTotal ← RunningTotal + NextNumber
NEXT Counter
OUTPUT RunningTotal
Average ← RunningTotal / 10
OUTPUT Average
```

It is very important that the value stored in `RunningTotal` is initialised to zero before we start adding the numbers being input.

TIP

Which type of loop? If it is known how many repetitions are required, choose a FOR loop. If the statements inside the loop might never be executed, choose a WHILE loop. If the statements inside the loop are to be executed at least once, a REPEAT loop might be more sensible.

TASK 12.05

Change the algorithm in Worked Example 12.08 so that the sequence of numbers is terminated by a rogue value of 0.

WORKED EXAMPLE 12.09

Using nested loops

The problem to be solved: Take as input two numbers and a symbol. Output a grid made up entirely of the chosen symbol, with the number of rows matching the first number input and the number of columns matching the second number input.

For example the three input values 3, 7 and &, result in the output:

```
&&&&&&&
&&&&&&&
&&&&&&&
```

We need two variables to store the number of rows and the number of columns. We also need a variable to store the symbol. We need a counter for the rows and a counter for the columns.

Identifier	Explanation
`NumberOfRows`	Stores the number of rows of the grid
`NumberOfColumns`	Stores the number of columns of the grid
`Symbol`	Stores the chosen character symbol
`RowCounter`	Counts the number of rows
`ColumnCounter`	Counts the number of columns

Table 12.11 Identifier table for the nested loop example

```
INPUT NumberOfRows
INPUT NumberOfColumns
INPUT Symbol
FOR RowCounter ← 1 TO NumberOfRows
    FOR ColumnCounter ← 1 TO NumberOfColumns
        OUTPUT Symbol // without moving to next line
    NEXT ColumnCounter
    OUTPUT Newline   // move to the next line
NEXT RowCounter
```

Each time round the outer loop (counting the number of rows) we complete the inner loop, outputting a symbol for each count of the number of columns. This type of construct is called a **nested loop**.

12.08 Stepwise refinement

Many problems that we want to solve are bigger than the ones we met so far. To make it easier to solve a bigger problem, we break the problem down into smaller steps. These might need breaking down further until the steps are small enough to solve easily.

For a solution to a problem to be programmable, we need to break down the steps of the solution into the basic constructs of sequence, assignment, selection, repetition, input and output.

We can use a method called **stepwise refinement** to break down the steps of our outline solution into smaller steps until it is detailed enough. In Section 12.02 we looked at a recipe for a cake. The step of mixing together all the ingredients was broken down into more detailed steps.

WORKED EXAMPLE 12.10

Drawing a pyramid using stepwise refinement

The problem to be solved: Take as input a chosen symbol and an odd number. Output a pyramid shape made up entirely of the chosen symbol, with the number of symbols in the final row matching the number input.

For example the two input values A and 9 result in the following output:

```
    A
   AAA
  AAAAA
 AAAAAAA
AAAAAAAAA
```

This problem is similar to Worked Example 12.09, but the number of symbols in each row starts with one and increases by two with each row. Each row starts with a decreasing number of spaces, to create the slope effect.

Our first attempt at solving this problem using structured English is:

```
01  Set up initial values
02  REPEAT
03     Output number of spaces
04     Output number of symbols
05     Adjust number of spaces and number of symbols to be output in next row
06  UNTIL the required number of symbols have been output in one row
```

The steps are numbered to make it easier to refer to them later.

This is not enough detail to write a program in a high-level programming language. Exactly what values do we need to set?

We need as input:

- the symbol character from which the pyramid is to be formed
- the number of symbols in the final row (for the pyramid to look symmetrical, this needs to be an odd number).

We need to calculate how many spaces we need in the first row. So that the slope of the pyramid is symmetrical, this number should be half of the final row's symbols. We need to set the number of symbols to be output in the first row to 1. We therefore need the identifiers listed in Table 12.12.

Identifier	Explanation
Symbol	The character symbol to form the pyramid
MaxNumberOfSymbols	The number of symbols in the final row

NumberOfSpaces	The number of spaces to be output in the current row
NumberOfSymbols	The number of symbols to be output in the current row

Table 12.12 Identifier table for pyramid example

Using pseudocode, we now refine the steps of our first attempt. To show which step we are refining, a numbering system is used as shown.

Step 01 can be broken down as follows:

```
01   // Set up initial values expands into:
01.1 INPUT Symbol
01.2 INPUT MaxNumberOfSymbols
01.3 NumberOfSpaces ← (MaxNumberOfSymbols – 1) / 2
01.4 NumberOfSymbols ← 1
```

Remember we need an odd number for `MaxNumberOfSymbols`. We need to make sure the input is an odd number. So we further refine Step 01.2:

```
01.2   // INPUT MaxNumberOfSymbols expands into:
01.2.1 REPEAT
01.2.2    INPUT MaxNumberOfSymbols
01.2.3 UNTIL MaxNumberOfSymbols MOD 2 = 1
01.2.4 // MOD 2 gives the remainder after integer division by 2
```

We can now look to refine Steps 03 and 04:

```
03 // Output number of spaces expands into:
03.1  FOR i ← 1 TO NumberOfSpaces
03.2    OUTPUT Space // without moving to next line
03.3  NEXT i
04    // Output number of symbols expands into:
04.1  FOR i ← 1 TO NumberOfSymbols
04.2    OUTPUT Symbol // without moving to next line
04.3  NEXT i
04.4  OUTPUT Newline // move to the next line
```

In Step 05 we need to decrease the number of spaces by 1 and increase the number of symbols by 2:

```
05   // Adjust values for next row expands into:
05.1 NumberOfSpaces ← NumberOfSpaces – 1
05.2 NumberOfSymbols ← NumberOfSymbols + 2
```

Step 06 essentially checks whether the number of symbols for the next row is now greater than the value input at the beginning.

```
06     UNTIL NumberOfSymbols > MaxNumberOfSymbols
```

We can put together all the steps and end up with a solution.

```
01     // Set Values
01.1   INPUT Symbol
01.2   // Input max number of symbols (an odd number)
01.2.1 REPEAT
01.2.2     INPUT MaxNumberOfSymbols
01.2.3 UNTIL MaxNumberOfSymbols MOD 2 = 1
01.3   NumberOfSpaces ← (MaxNumberOfSymbols – 1) / 2
01.4   NumberOfSymbols ← 1
02     REPEAT
03         // Output number of spaces
03.1       FOR i ← 1 TO NumberOfSpaces
```

```
03.2            OUTPUT Space // without moving to next line
03.3        NEXT i
04          // Output number of symbols
04.1        FOR i ← 1 TO NumberOfSymbols
04.2            OUTPUT Symbol // without moving to next line
04.3        NEXT i
04.4        OUTPUT Newline // move to the next line
05          // Adjust Values For Next Row
05.1        NumberOfSpaces ← NumberOfSpaces – 1
05.2        NumberOfSymbols ← NumberOfSymbols + 2
06      UNTIL NumberOfSymbols > MaxNumberOfSymbols
```

TASK 12.06

Use stepwise refinement to output a hollow triangle. For example the two input values A and 9 result in the following output:

```
    A
   A A
  A   A
 A     A
AAAAAAAAA
```

A first attempt at solving this problem using structured English is:

```
01   Set up initial values
02   REPEAT
03       Output leading number of spaces
04       Output symbol, middle spaces, symbol
05       Adjust number of spaces and number of symbols to be output in next row
06   UNTIL the required number of symbols have been output in one row
```

12.09 Modules

Another method of developing a solution is to decompose the problem into sub-tasks. Each sub-task can be considered as a ‘module’ that is refined separately. Modules are procedures and functions.

A **procedure** groups together a number of steps and gives them a name (an identifier). We can use this identifier when we want to refer to this group of steps. When we want to perform the steps in a procedure we call the procedure by its name.

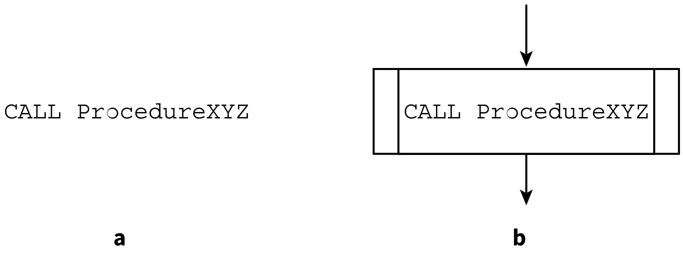


Figure 12.11 Representation of a procedure in (a) pseudocode and (b) a flowchart

A **function** groups together a number of steps and gives them a name (an identifier). These steps produce and return a value that is used in an expression. Worked Example 12.12 uses functions.

Note: Because a function returns a value, the function definition states the data type of this value. See more about data types in Chapter 13.

The rules for module identifiers are the same as for variable identifiers (see Section 12.04)

WORKED EXAMPLE 12.11

Drawing a pyramid using modules

The problem is the same as in Worked Example 12.10.

When we want to set up the initial values, we call a procedure, using the following statement:

```
CALL SetValues
```

We can rewrite the top-level solution to our pyramid problem using a procedure for each step, as:

```
CALL SetValues
REPEAT
      CALL OutputSpaces
      CALL OutputSymbols
      CALL AdjustValuesForNextRow
UNTIL NumberOfSymbols > MaxNumberOfSymbols
```

This top-level solution calls four procedures. This means each procedure has to be defined. The procedure definitions are:

```
PROCEDURE SetValues
      INPUT Symbol
      CALL InputMaxNumberOfSymbols // need to ensure it is an odd number
      NumberOfSpaces ← (MaxNumberOfSymbols - 1) / 2
      NumberOfSymbols ← 1
ENDPROCEDURE
PROCEDURE InputMaxNumberOfSymbols
      REPEAT
          INPUT MaxNumberOfSymbols
      UNTIL MaxNumberOfSymbols MOD 2 = 1
ENDPROCEDURE
PROCEDURE OutputSpaces
      FOR Count1 ← 1 TO NumberOfSpaces
```

```
            OUTPUT Space // without moving to next line
        NEXT Count1
ENDPROCEDURE
PROCEDURE OutputSymbols
        FOR Count2 ← 1 TO NumberOfSymbols
            OUTPUT Symbol // without moving to next line
        NEXT Count2
        OUTPUT Newline // move to the next line
ENDPROCEDURE
PROCEDURE AdjustValuesForNextRow
        NumberOfSpaces ← NumberOfSpaces – 1
        NumberOfSymbols ← NumberOfSymbols + 2
ENDPROCEDURE
```

TASK 12.07

Amend your algorithm for Task 12.05 to use modules.

WORKED EXAMPLE 12.12

Drawing a pyramid using modules

The problem is the same as in Worked Example 12.11.

We can rewrite the top-level solution to our pyramid problem using procedures and functions.

```
01    CALL SetValues
02    REPEAT
03         CALL OutputSpaces
04         CALL OutputSymbols
05.1       NumberOfSpaces ← AdjustedNumberOfSpaces
05.2       NumberOfSymbols ← AdjustedNumbeOfSymbols
06    UNTIL NumberOfSymbols > MaxNumberOfSymbols
```

This top-level solution calls three procedures. It also makes use of two functions in lines 05.1 and 05.2.

The procedures and functions have to be defined.

```
PROCEDURE SetValues
    INPUT Symbol
    MaxNumberOfSymbols ← ValidatedMaxNumberOfSymbols
    NumberOfSpaces ← (MaxNumberOfSymbols - 1) / 2
    NumberOfSymbols ← 1
ENDPROCEDURE
FUNCTION ValidatedMaxNumberOfSymbols RETURNS INTEGER
    REPEAT
        INPUT MaxNumberOfSymbols
    UNTIL MaxNumberOfSymbols MOD 2 = 1
    RETURN MaxNumberOfSymbols
ENDFUNCTION
PROCEDURE OutputSpaces
    FOR Count1 ← 1 TO NumberOfSpaces
        OUTPUT Space // without moving to next line
    NEXT Count1
ENDPROCEDURE
```

```
PROCEDURE OutputSymbols
    FOR Count2 ← 1 TO NumberOfSymbols
       OUTPUT Symbol // without moving to next line
    NEXT Count2
    OUTPUT Newline // move to the next line
ENDPROCEDURE
FUNCTION AdjustedNumberOfSpaces RETURNS INTEGER
    NumberOfSpaces ← NumberOfSpaces – 1
    RETURN NumberOfSpaces
ENDFUNCTION
FUNCTION AdjustedNumberOfSymbols RETURNS INTEGER
    NumberOfSymbols ← NumberOfSymbols + 2
    RETURN NumberOfSymbols
ENDFUNCTION
```

Note that procedure `SetValues` uses a function `ValidatedMaxNumberOfSymbols`.

One benefit of using modules is that individual modules can be reused in other solutions. Therefore, modules should be designed to be self-contained. That means they should not rely on external variables. All variables that are required by a module should be passed to it using parameters. To illustrate this, look at Worked Example 12.13

WORKED EXAMPLE 12.13

Drawing a pyramid using modules and parameters

The problem is the same as in Worked Example 12.12.

```
01   CALL SetValues(Symbol, MaxNumberOfSymbols, NumberOfSpaces, NumberOfSymbols)
02   REPEAT
03        CALL OutputSpaces(NumberOfSpaces)
04        CALL OutputSymbols(NumberOfSymbols, Symbol)
05.1      NumberOfSpaces ← AdjustedNumberOfSpaces(NumberOfSpaces)
05.2      NumberOfSymbols ← AdjustedNumbeOfSymbols(NumberOfSymbols)
06   UNTIL NumberOfSymbols > MaxNumberOfSymbols
```

Module definitions:

```
PROCEDURE SetValues(Symbol, MaxNumberOfSymbols, NumberOfSpaces, NumberOfSymbols)
     INPUT Symbol
     MaxNumberOfSymbols ← ValidatedMaxNumberOfSymbols
     NumberOfSpaces ← (MaxNumberOfSymbols - 1) / 2
     NumberOfSymbols ← 1
ENDPROCEDURE
FUNCTION ValidatedMaxNumberOfSymbols RETURNS INTEGER
     REPEAT
         INPUT MaxNumberOfSymbols
     UNTIL MaxNumberOfSymbols MOD 2 = 1
     RETURN MaxNumberOfSymbols
ENDFUNCTION
PROCEDURE OutputSpaces(NumberOfSpaces)
     FOR Count1 ← 1 TO NumberOfSpaces
         OUTPUT Space // without moving to next line
     NEXT Count1
ENDPROCEDURE
PROCEDURE OutputSymbols(NumberOfSymbols, Symbol)
```

```
    FOR Count2 ← 1 TO NumberOfSymbols
        OUTPUT Symbol // without moving to next line
    NEXT Count2
    OUTPUT Newline // move to the next line
ENDPROCEDURE
FUNCTION AdjustedNumberOfSpaces(NumberOfSpaces) RETURNS INTEGER
    NumberOfSpaces ← NumberOfSpaces – 1
    RETURN NumberOfSpaces
ENDFUNCTION
FUNCTION AdjustedNumbeOfSymbols(NumberOfSymbols) RETURNS INTEGER
    NumberOfSymbols ← NumberOfSymbols + 2
    RETURN NumberOfSymbols
ENDFUNCTION
```

Note that the procedure `OutputSpaces` uses a variable, `Count1`, which is used only within the module. Similarly, `OutputSymbols` uses variable `Count2` only within the module. We call such a variable a **local variable** (see Chapter 14, Section 14.09). A variable available to all modules is known as a **global variable** (see Chapter 14, Section 14.09).

TIP

Good design uses local variables as it makes modules independent and re-usable.

Reflection Point:

Can you think of other problems and use decomposition to break them down into basic constructs, input and output statements?

Summary

- Abstraction involves filtering out information that is not needed to solve the problem.
- Decomposition is breaking down problems into sub-problems, leading to the concept of a program module.
- An algorithm is a sequence of steps that can be carried out to solve a problem.
- Algorithms are expressed using the four basic constructs of assignment, sequence, selection and repetition.
- Algorithms can be documented using pseudocode.
- Stepwise refinement: breaking down the steps of an outline solution into smaller and smaller steps.
- Logic statements use the relational operators =, <, >, <>, <= and >= and the logic operators AND, OR and NOT.
- Selection constructs and conditional loops use conditions to determine the steps to be followed.

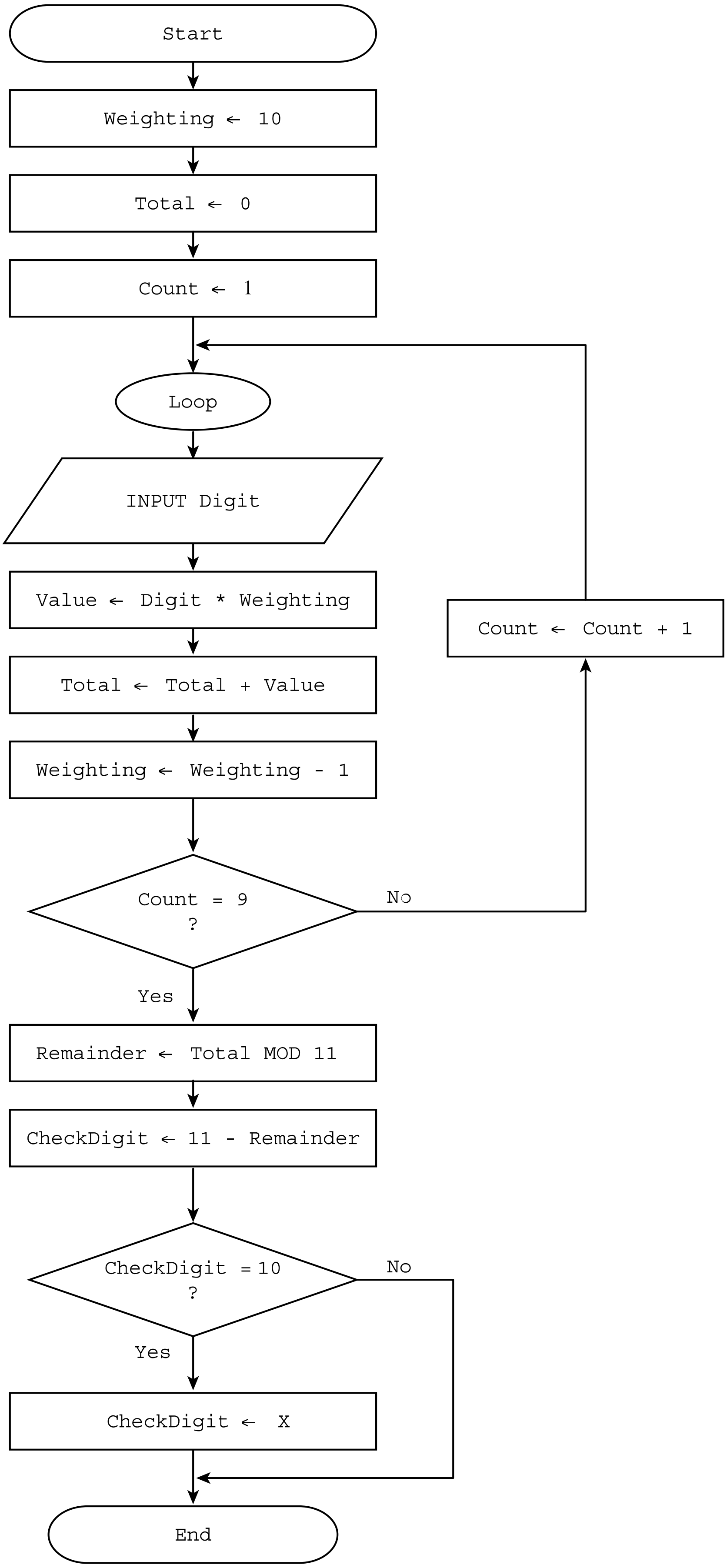
Exam-style Questions

1 The Modulo-11 method of calculating a check digit for a sequence of nine digits is as follows:

Each digit in the sequence is given a weight depending on its position in the sequence. The leftmost digit has a weight of 10. The next digit to the right has a weight of 9, the next one 8 and so on. Values are calculated by multiplying each digit by its weight. These values are added together and the sum is divided by 11. The remainder from this division is subtracted from 11 and this value is the check digit. If this value is 10, then the check digit is X. Note that `x MOD y` gives the remainder from the division of `x` by `y`.

The flowchart shows the algorithm for calculating the Modulo-11 check digit.

Write pseudocode from the flowchart.



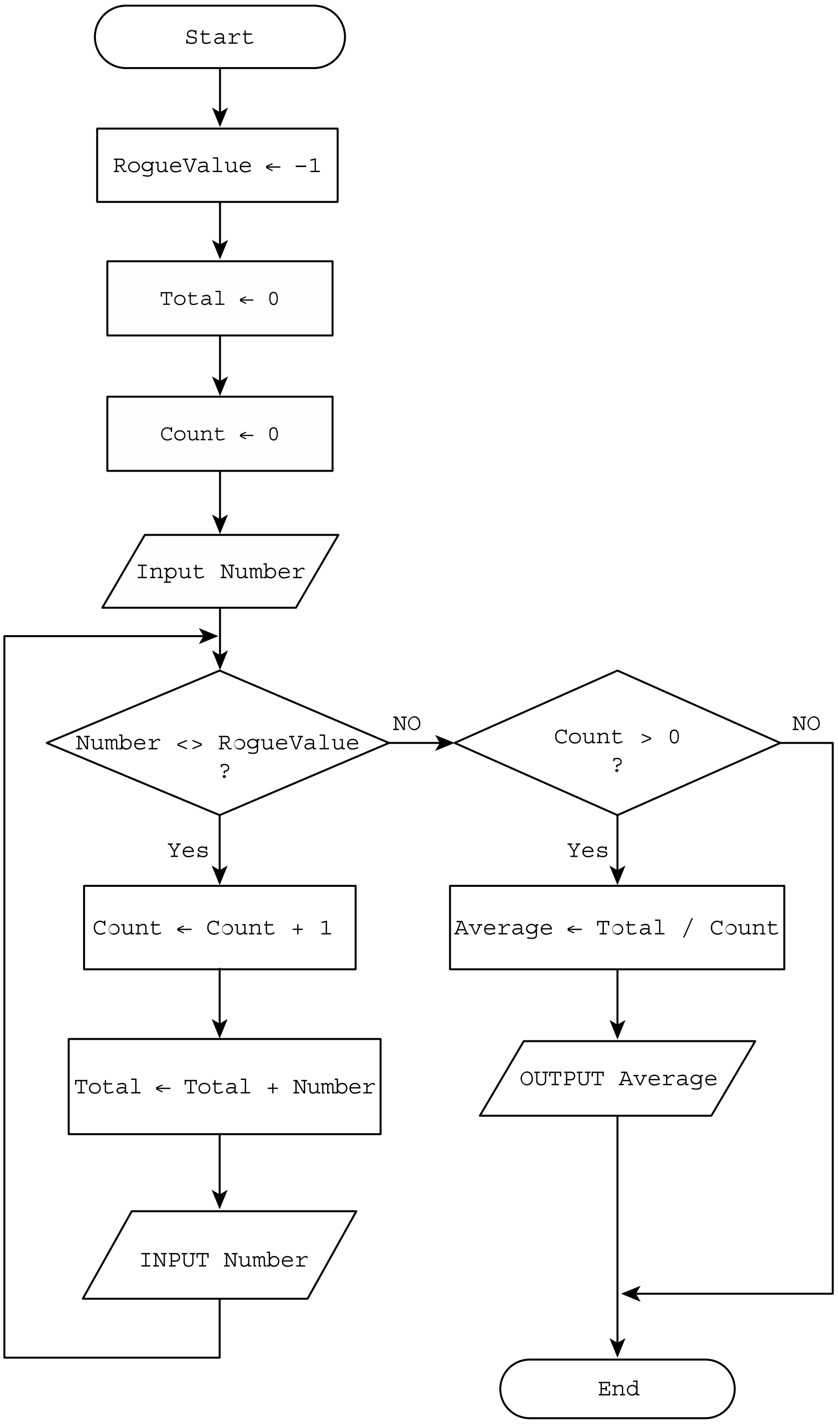
[9]

2 Write pseudocode for the following problem given in structured English.

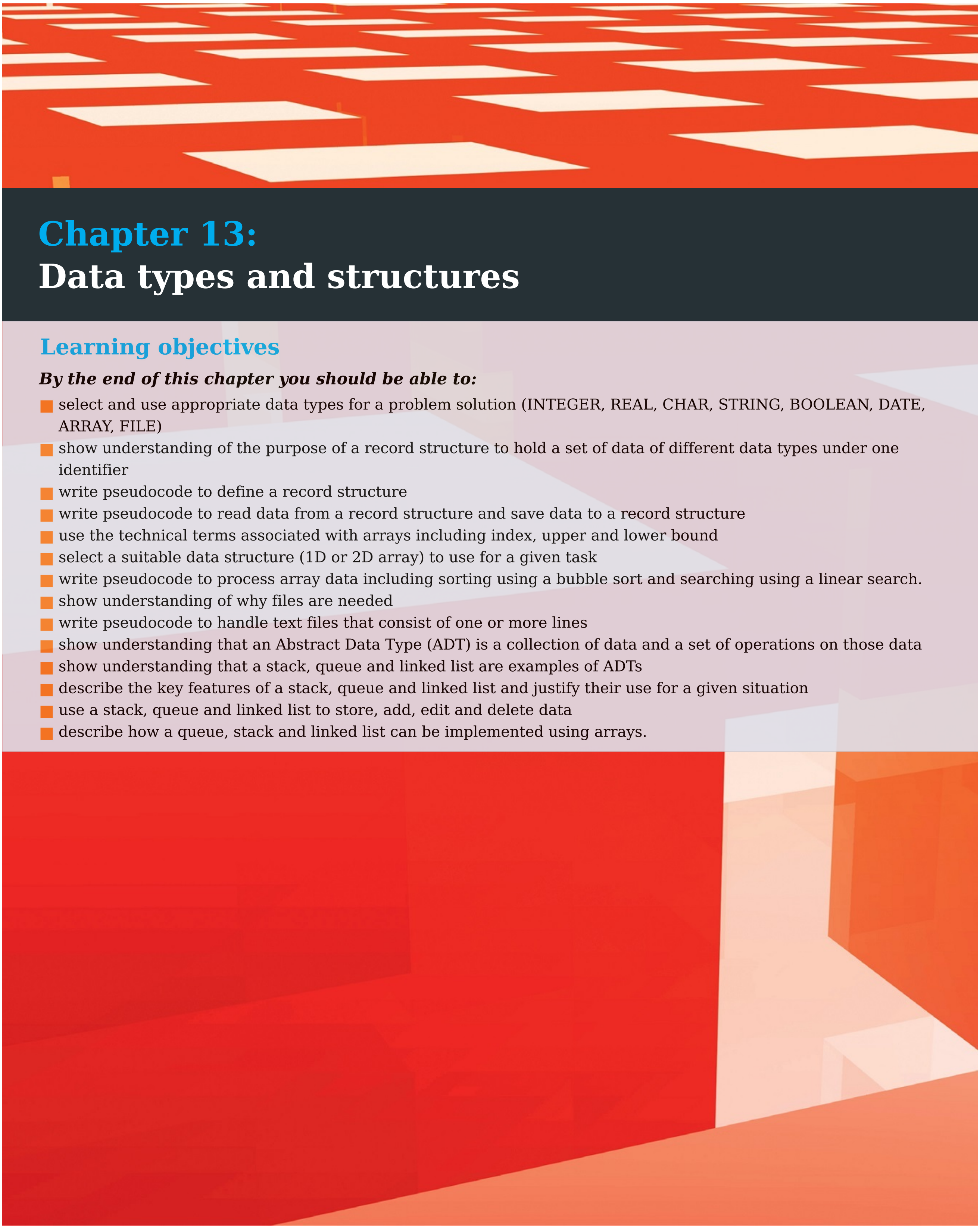
```
REPEAT the following UNTIL the number input is zero
INPUT a number
Check whether number is positive or negative
Increment positive number count if the number is positive
Increment negative number count if the number is negative
```

[7]

3 Write pseudocode from the given flowchart. Use a `WHILE` loop.



[8]



Chapter 13:
Data types and structures

Learning objectives

By the end of this chapter you should be able to:

- select and use appropriate data types for a problem solution (INTEGER, REAL, CHAR, STRING, BOOLEAN, DATE, ARRAY, FILE)
- show understanding of the purpose of a record structure to hold a set of data of different data types under one identifier
- write pseudocode to define a record structure
- write pseudocode to read data from a record structure and save data to a record structure
- use the technical terms associated with arrays including index, upper and lower bound
- select a suitable data structure (1D or 2D array) to use for a given task
- write pseudocode to process array data including sorting using a bubble sort and searching using a linear search.
- show understanding of why files are needed
- write pseudocode to handle text files that consist of one or more lines
- show understanding that an Abstract Data Type (ADT) is a collection of data and a set of operations on those data
- show understanding that a stack, queue and linked list are examples of ADTs
- describe the key features of a stack, queue and linked list and justify their use for a given situation
- use a stack, queue and linked list to store, add, edit and delete data
- describe how a queue, stack and linked list can be implemented using arrays.

13.01 Data types

Primitive data types

In Chapter 12 we used variables to store values required by our algorithm. Look at Worked Example 12.01. The Identifier Table 12.02 lists two variable identifiers: `Miles` and `Km`. An identifier table should also show what sort of data (or data type) is going to be stored in each variable. The explanation shows that Miles will be a whole number, but that Km will be calculated using the formula Miles * 1.61. This will result in a number that may not be a whole number.

Primitive data types are those variables that can be defined simply by commands built into the programming language. Primitive data types are also known as atomic data types. In Computer Science a whole number is referred to as an INTEGER and a number with a decimal point is referred to as a REAL. Conditions are either TRUE or FALSE. These are logical values known as BOOLEAN. Sometimes we may want to store a single character; this is referred to as a CHAR.

A value that will always be a whole number should be defined to be of type INTEGER, such as when counting the iterations of a loop.

TIP

See Table 13.01 for a list of data types you should be familiar with.

See Chapter 1 (Sections 1.02 and 1.03) on how integers and characters are represented inside the computer. Chapter 16 (Section 16.03) covers the internal representation of real (single, double, float) numbers.

Further data types

If we want to store several characters; this is known as a string.

Note that there is a difference between the number 12 and the string “12”.

The string data type is known as a structured type because it is essentially a sequence of characters. A special case is the empty string: a value of data type string, but with no characters stored in it.

When we write a date, such as 3 February 2018, we can also write this as a set of three numbers: 3/2/2018. Sometimes we might wish to calculate with dates, such as taking one date away from another to find out how many days, months and years are between these dates. To make it easier to do this, DATE has been designed as a data type. To see how different programming languages implement this data type, see Chapter 14 Section 14.03.

INTEGER	A signed whole number
REAL	A signed number with a decimal point
CHAR	A single character
STRING	A sequence of zero or more characters
BOOLEAN	The logical values TRUE and FALSE
DATE	A date consisting of day, month and year, sometimes including a time in hours, minutes and seconds

Table 13.01 Summary of pseudocode data types

TASK 13.01

Look at the identifier tables in Chapter 12 (Tables 12.06 and 12.09 to 12.12). Give the data type that is appropriate for each variable listed.

13.02 The record type

Sometimes variables of different data types are a logical group, such as data about a person (name, date of birth, height, number of siblings, whether they are a full-time student).

Name is a STRING; date of birth is a DATE; height is a REAL; number of siblings is an INTEGER; whether they are a full-time student is a BOOLEAN.

We can declare a record type to suit our purposes. The record type is known as a user-defined type, because the programmer can decide which variables (fields) to include as a record.

TIP

A record type is also known as a composite type.

In pseudocode a record type is declared as:

```
TYPE <TypeIdentifier>
    DECLARE <field identifier> : <data type>
    .
    .
ENDTYPE
```

We can now declare a variable of this record type:

```
DECLARE <variable identifier> : <record type>
```

And then access an individual field using the dot notation:

```
<variable identifier>.<field identifier>
```

Using the example above we can declare a Person record type:

```
TYPE PersonType
    Name : STRING
    DateOfBirth : DATE
    Height : REAL
    NumberOfSiblings : INTEGER
    IsFullTimeStudent : BOOLEAN
ENDTYPE
```

To declare a variable of this type we write:

```
DECLARE Person : PersonType
```

And now we can assign a value to a field of this Person record:

```
Person.Name ← "Fred"
Person.NumberOfSiblings ← 3
Person.IsFullTimeStudent ← TRUE
```

To output a field of a record:

```
OUTPUT Person.Name
```

TASK 13.02

Write the declaration of a record type to store the details of a book: Title, Year of publication, Price, ISBN.

Write the statements required to assign the values “Computer Science”, 2019, £44.95, “9781108733755” to the fields respectively.

13.03 Arrays

Sometimes we want to organise data values into a list or a table / matrix. In most programming languages these structures are known as arrays. An array is an ordered set of data items, usually of the same type, grouped together using a single identifier. Individual array elements are addressed using an **array index** for each array dimension.

A list is a one-dimensional (1D) array and a table or matrix is a two-dimensional (2D) array.

TIP

When writing pseudocode, arrays need to be declared before they are used. This means choosing an identifier, the data type of the values to be stored in the array and **upper bound** and **lower bound** for each dimension.

13.04 One-dimensional arrays

When we write a list on a piece of paper and number the individual items, we would normally start the numbering with 1. You can view a 1D array like a numbered list of items. Many programming languages number array elements from 0 (the lower bound), including VB.NET, Python and Java. Depending on the problem to be solved, it might make sense to ignore element 0. The upper bound is the largest number used for numbering the elements of an array.

In pseudocode, a 1D array declaration is written as:

```
DECLARE <arrayIdentifier> : ARRAY[<lowerBound>:<upperBound>] OF <dataType>
```

Here is a pseudocode example:

```
DECLARE List1 : ARRAY[1:3] OF STRING  // 3 elements in this list
DECLARE List2 : ARRAY[0:5] OF INTEGER // 6 elements in this list
DECLARE List3 : ARRAY[1:100] OF INTEGER // 100 elements in this list
DECLARE List4 : ARRAY[0:25] OF CHAR  // 26 elements in this list
```

Accessing 1D arrays

A specific element in an array is accessed using an index value. In pseudocode, this is written as:

```
<arrayIdentifier>[x]
```

The *n*th element within the array `MyList` is referred to as `MyList[n]`.

Here is a pseudocode example:

```
NList[25] ← 0  // set 25th element to zero
AList[3] ← 'D' // set 3rd element to letter D
```

WORKED EXAMPLE 13.01

Working with a one-dimensional array

The problem to be solved: Take seven numbers as input and store them for later use.

We could use seven separate variables. However, if we wanted our algorithm to work with 70 numbers, for example, then setting up 70 variables would be complicated and waste time. Instead, we can make use of a data structure known as a 'linear list' or a 1D array.

This array is given an identifier, for example `MyList`, and each element within the array is referred to using this identifier and its position (index) within the array. For example, `MyList[4]` refers to the element at position 4 in the `MyList` array. If we are counting the element at position 0 as the first element, `MyList[4]` refers to the fifth element.

We can use a loop to access each array element in turn. If the numbers input to the pseudocode algorithm below are 25, 34, 98, 7, 41, 19 and 5 then the algorithm will produce the result in Figure 13.01.

```
FOR Index ← 0 TO 6
    INPUT MyList[Index]
NEXT Index
```

Index	**MYList**
[0]	25
[1]	34
[2]	98
[3]	7
[4]	41
[5]	19
[6]	5

Figure 13.01 `Mylist` array populated by a loop

TASK 13.03

Define two arrays, one for your friends' names and one for their ages as shown in Figure 13.02.

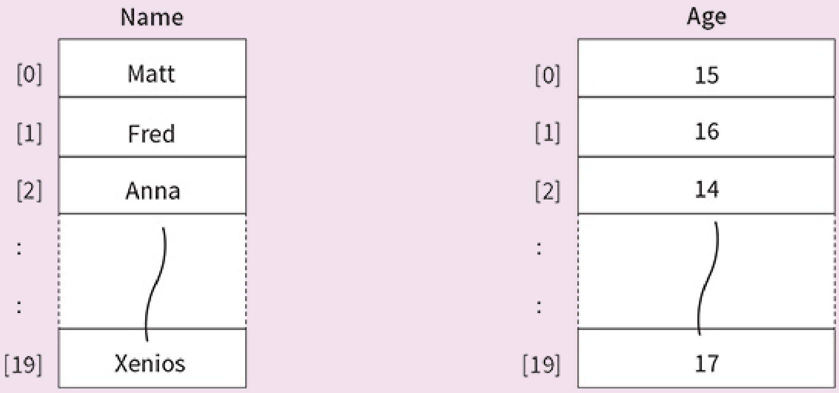


Figure 13.02 Arrays for names and ages

WORKED EXAMPLE 13.02

Searching a 1D array

The problem to be solved: Take a number as input. Search for this number in an existing 1D array of seven numbers (see Worked Example 13.01).

Start at the first element of the array and check each element in turn until the search value is found or the end of the array is reached. This method is called a **linear search**.

Identifier	Data type	Explanation
`MyList`	`ARRAY[0:6] OF INTEGER`	Data structure (1D array) to store seven numbers
`MaxIndex`	`INTEGER`	The number of elements in the array
`SearchValue`	`INTEGER`	The value to be searched for
`Found`	`BOOLEAN`	`TRUE` if the value has been found `FALSE` if the value has not been found
`Index`	`INTEGER`	Index of the array element currently being processed

Table 13.02 Identifier table for linear search algorithm

```
MaxIndex ← 6
INPUT SearchValue
Found ← FALSE
Index ← –1
REPEAT
    Index ← Index + 1
    IF MyList[Index] = SearchValue
      THEN
        Found ← TRUE
    ENDIF
UNTIL FOUND = TRUE OR Index >= MaxIndex
IF Found = TRUE
```

```
  THEN
    OUTPUT "Value found at location: " Index
  ELSE
    OUTPUT "Value not found"
ENDIF
```

The complex condition to the `REPEAT...UNTIL` loop allows us to exit the loop when the search value is found. Using the variable `Found` makes the algorithm easier to understand. `Found` is initialised (first set) to `FALSE` before entering the loop and set to `TRUE` if the value is found.

If the value is not in the array, the loop terminates when `Index` is greater than or equal to `MaxIndex`. That means we have come to the end of the array. Note that using `MaxIndex` in the logic statement to terminate the loop makes it much easier to adapt the algorithm when the array consists of a different number of elements. The algorithm only needs to be changed in the first line, where `MaxIndex` is given a value.

TASK 13.04

Use the algorithm in Worked Example 13.02 as a design pattern. Write an algorithm using the arrays from Task 13.03 to search for a friend's name and output their age.

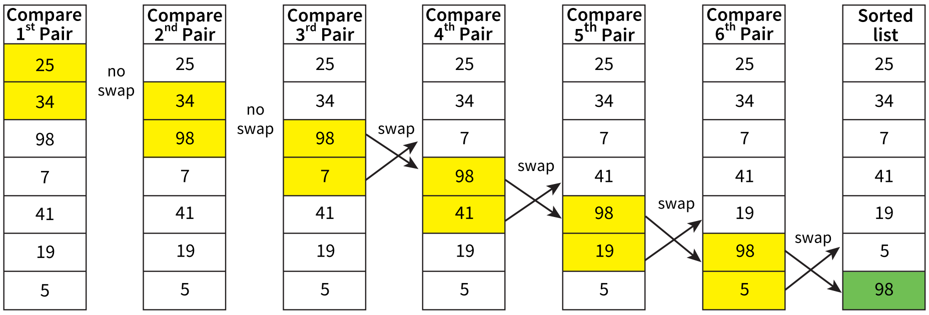
WORKED EXAMPLE 13.03

Sorting elements in a 1D array

The simplest way to sort an unordered list of values is the following method.

1. Compare the first and second values. If the first value is larger than the second value, swap them.
2. Compare the second and third values. If the second value is larger than the third value, swap them.
3. Compare the third and fourth values. If the third value is larger than the fourth value, swap them.
4. Keep on comparing adjacent values, swapping them if necessary, until the last two values in the list have been processed.

Figure 13.03 shows what happens to the values as we work down the array, following this algorithm.



Compare 1st Pair	Compare 2nd Pair	Compare 3rd Pair	Compare 4th Pair	Compare 5th Pair	Compare 6th Pair	Sorted list
25	25	25	25	25	25	25
34	34	34	34	34	34	34
98	98	98	7	7	7	7
7	7	7	98	41	41	41
41	41	41	41	98	19	19
19	19	19	19	19	98	5
5	5	5	5	5	5	98

Figure 13.03 Swapping values working down the array

When we have completed the first pass through the entire array, the largest value is in the correct position at the end of the array. The other values may or may not be in the correct order.

We need to work through the array again and again. After each pass through the array the next largest value will be in its correct position, as shown in Figure 13.04.

Original list	After pass 1	After pass 2	After pass 3	After pass 4	After pass 5	After pass 6
25	25	25	7	7	7	5
34	34	7	25	19	5	7
98	7	34	19	5	19	19
7	41	19	5	25	25	25
41	19	5	34	34	34	34
19	5	41	41	41	41	41
5	98	98	98	98	98	98

Figure 13.04 States of the array after each pass

In effect we perform a loop within a loop, a nested loop. This method is known as a **bubble sort**. The name comes from the fact that smaller values slowly rise to the top, like bubbles in a liquid.

The identifiers needed for the algorithm are listed in Table 13.03.

Identifier	Data type	Explanation
`MyList`	`ARRAY[0:6] OF INTEGER`	Data structure (1D array) to store seven numbers
`MaxIndex`	`INTEGER`	The upper bound of the array
`n`	`INTEGER`	The number of pairs of elements to compare in each pass
`i`	`INTEGER`	Counter for outer loop
`j`	`INTEGER`	Counter for inner loop
`Temp`	`INTEGER`	Variable for temporary storage while swapping values

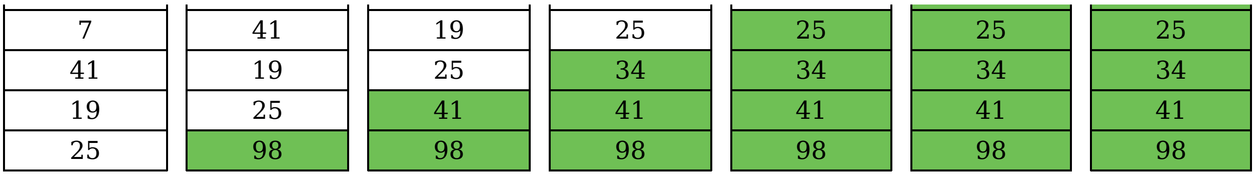
Table 13.03 Identifier table for bubble sort algorithm

The algorithm in pseudocode is:

```
n ← MaxIndex – 1
FOR i ← 0 TO MaxIndex – 1
    FOR j ← 0 TO n
        IF MyList[j] > MyList[j + 1]
          THEN
            Temp ← MyList[j]
            MyList[j] ← MyList[j + 1]
            MyList[j + 1] ← Temp
        ENDIF
    NEXT j
    n ← n – 1 // this means the next time round the inner loop, we don't
              // look at the values already in the correct positions.
NEXT i
```

The values to be sorted may already be in the correct order before the outer loop has been through all its iterations. Look at the list of values in Figure 13.05. It is only slightly different from the first list we sorted.

Original list	After pass 1	After pass 2	After pass 3	After pass 4	After pass 5	After pass 6
5	5	5	5	5	5	5
34	34	7	7	7	7	7
98	7	34	19	19	19	19



7	41	19	25	25	25	25
41	19	25	34	34	34	34
19	25	41	41	41	41	41
25	98	98	98	98	98	98

Figure 13.05 States of the list after each pass

After the third pass the values are all in the correct order but our algorithm will carry on with three further passes through the array. This means we are making comparisons when no further comparisons need to be made.

If we have gone through the whole of the inner loop (one pass) without swapping any values, we know that the array elements must be in the correct order. We can therefore replace the outer loop with a conditional loop.

We can use a variable `NoMoreSwaps` to store whether or not a swap has taken place during the current pass. We initialise the variable `NoMoreSwaps` to `TRUE`. When we swap a pair of values we set `NoMoreSwaps` to `FALSE`. At the end of the pass through the array we can check whether a swap has taken place.

The identifier table for this improved algorithm is shown in Table 13.04.

Identifier	Data type	Explanation
`MyList`	`ARRAY[0:6] OF INTEGER`	Data structure (1D array) to store seven numbers
`MaxIndex`	`INTEGER`	The upper bound of the array
`n`	`INTEGER`	The number of pairs of elements to compare in each pass
`NoMoreSwaps`	`BOOLEAN`	`TRUE` when no swaps have occurred in current pass `FALSE` when a swap has occurred
`j`	`INTEGER`	Counter for inner loop
`Temp`	`INTEGER`	Variable for temporary storage while swapping values

Table 13.04 Identifier table for improved bubble sort algorithm

This improved algorithm in pseudocode is:

```
n ← MaxIndex – 1
REPEAT
    NoMoreSwaps ← TRUE
    FOR j ← 0 TO n
        IF MyList[j] > MyList[j + 1]
          THEN
            Temp ← MyList[j]
            MyList[j] ← MyList[j + 1]
            MyList[j + 1] ← Temp
            NoMoreSwaps ← FALSE
        ENDIF
    NEXT j
    n ← n – 1
UNTIL NoMoreSwaps = TRUE
```

Discussion Point:

What happens if the array elements are already in the correct order?

TASK 13.05

Rewrite the algorithm in Worked Example 13.03 to sort the array elements into descending order.

13.05 Two-dimensional arrays

When we write a table of data (a matrix) on a piece of paper and want to refer to individual elements of the table, the convention is to give the row number first and then the column number. When declaring a 2D array, the number of rows is given first, then the number of columns. Again we have lower and upper bounds for each dimension.

In pseudocode, a 2D array declaration is written as:

```
DECLARE <identifier> : ARRAY[<lBound1>:<uBound1>,
<lBound2>:<uBound2>] OF <dataType>
```

The array elements in a 2D array can be numbered from 0. Sometimes it is more intuitive to use rows from row 1 and columns from column 1, as shown with the board game in Worked Example 13.05.

To declare a 2D array to represent a game board of six rows and seven columns, the pseudocode statement is:

```
Board : ARRAY[1:6,1:7] OF INTEGER
```

Accessing 2D arrays

A specific element in a table is accessed using an index pair. In pseudocode this is written as:

```
<arrayIdentifier>[x,y]
```

Pseudocode example:

```
Board[3,4] ← 0 // sets the element in row 3 and column 4 to zero
```

When we want to access each element of a 1D array, we use a loop to access each element in turn. When working with a 2D array, we need a loop to access each row. Within each row we need to access each column. This means we use a loop within a loop (nested loops).

In structured English our algorithm is:

```
For each row
    For each column
        Assign the initial value to the element at the current position
```

WORKED EXAMPLE 13.04

Working with two-dimensional arrays and nested loops

Using pseudocode, the algorithm to set each element of array `ThisTable` to zero is:

```
FOR Row ← 0 TO MaxRowIndex
    FOR Column ← 0 TO MaxColumnIndex
        ThisTable[Row, Column] ← 0
    NEXT Column
NEXT Row
```

We need the identifiers shown in Table 13.05.

Identifier	Data type	Explanation
ThisTable	ARRAY[0:3, 0:5] OF INTEGER	Table data structure (2D array) to store values
MaxRowIndex	INTEGER	The upper bound of the row index
MaxColumnIndex	INTEGER	The upper bound of the column index
Row	INTEGER	Counter for the row index
Column	INTEGER	Counter for the column index

Table 13.05 Identifier table for working with a table

When we want to output the contents of a 2D array, we again need nested loops. We want to output all the values in one row of the array on the same line. At the end of the row, we want to output a new line.

```
FOR Row ← 0 TO MaxRowIndex
    FOR Column ← 0 TO MaxColumnIndex
        OUTPUT ThisTable[Row, Column] // stay on same line
    NEXT Column
    OUTPUT Newline      // move to next line for next row
NEXT Row
```

TASK 13.06

1. Declare a 2D array to store the board data for the game Noughts and Crosses. The empty squares of the board are to be represented by a space. Player A's counters are to be represented by “O”. Player B’s counters are to be represented by “X”.
2. Initialise the array to start with each square being empty.
3. Write a statement to represent player A placing their counter in the top left square.
4. Write a statement to represent player B placing their counter in the middle square.

WORKED EXAMPLE 13.05

Creating a program to play Connect 4

Connect 4 is a game played by two players. In the commercial version shown in Figure 13.06, one player uses red tokens and the other uses black. Each player has 21 tokens. The game board is a vertical grid of six rows and seven columns.

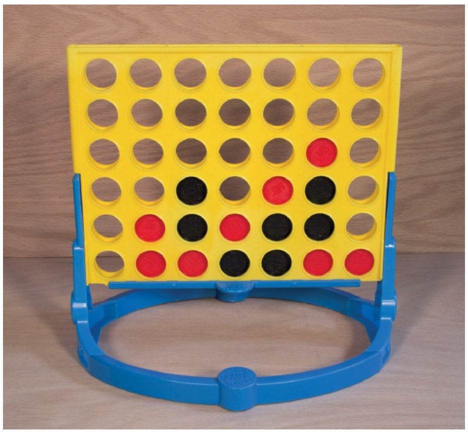


Figure 13.06 A Connect 4 board

Columns get filled with tokens from the bottom. The players take it in turns to choose a column that is not full and drop a token into this column. The token will occupy the lowest empty position in the chosen column. The winner is the player who is the first to connect four of their own tokens in a horizontal, vertical or diagonal line. If all tokens have been used and neither player has connected four tokens, the game ends in a draw.

If we want to write a program to play this game on a computer, we need to work out the steps required to ‘solve the problem’, that means to let players take their turn in placing tokens and checking for a winner. We will designate our players (and their tokens) by ‘O’ and ‘X’. The game board will be represented by a 2D array. To simplify the problem, the winner is the player who is the first to connect four of their tokens horizontally or vertically.

Our first attempt in structured English is:

```
Initialise board
Set up game
Display board
While game not finished
   Player makes a move
   Display board
   Check if game finished
   If game not finished, swap player
```

The top-level pseudocode version using modules is:

```
01 CALL InitialiseBoard
02 CALL SetUpGame
03 CALL OutputBoard
04 WHILE GameFinished = FALSE DO
05     CALL PlayerMakesMove
06     CALL OutputBoard
07     CALL CheckGameFinished
08     IF GameFinished = FALSE
09       THEN
10         CALL SwapThisPlayer
11     ENDIF
12 ENDWHILE
```

Note that Steps 03 and 06 are the same. This means that we can save ourselves some effort. We only need to define this module once, but can call it from more than one place. This is one of the advantages of using modules.

The identifier table for the program is shown in Table 13.06.

Identifier	Data type	Explanation
`Board`	`ARRAY[1:6,1:7] OF CHAR`	2D array to represent the board
`InitialiseBoard`		Procedure to initialise the board to all blanks
`SetUpGame`		Procedure to set initial values for `GameFinished` and `ThisPlayer`
`GameFinished`	`BOOLEAN`	FALSE if the game is not finished TRUE if the board is full or a player has won
`ThisPlayer`	`CHAR`	`'O'` when it is Player O's turn `'X'` when it is Player X's turn
`OutputBoard`		Procedure to output the current contents of the board
`PlayerMakesMove`		Procedure to place the current player's token into the chosen board location
`CheckGameFinished`		Procedure to check if the token just placed makes the current player a winner or board is full
`SwapThisPlayer`		Procedure to change player's turn

Table 13.06 Initial identifier table for Connect 4 game

Now we can refine each procedure (module). This is likely to add some more identifiers to our identifier table. The additional entries required are shown after each procedure.

```
PROCEDURE InitialiseBoard
```

```
    FOR Row ← 1 TO 6
        FOR Column ← 1 TO 7
            Board[Row, Column] ← BLANK // use a suitable value for blank
        NEXT Column
    NEXT Row
ENDPROCEDURE
```

Identifier	Data type	Explanation
`Row`	`INTEGER`	Loop counter for the rows
`Column`	`INTEGER`	Loop counter for the columns
`BLANK`	`CHAR`	A value that represents an empty board location

Table 13.07 Additional identifiers for the `InitialiseBoard` procedure

```
PROCEDURE SetUpGame
    ThisPlayer ← 'O' // Player O always starts
    GameFinished ← FALSE
ENDPROCEDURE
PROCEDURE OutputBoard
    FOR Row ← 6 DOWNTO 1
        FOR Column ← 1 TO 7
            OUTPUT Board[Row, Column] // don't move to next line
        NEXT Column
        OUTPUT Newline // move to next line
    NEXT Row
ENDPROCEDURE
PROCEDURE PlayerMakesMove
    ValidColumn ← PlayerChoosesColumn // a module returns column number
    ValidRow ← FindFreeRow // a module returns row number
    Board[ValidRow, ValidColumn] ← ThisPlayer
ENDPROCEDURE
```

Identifier	Data type	Explanation
`ValidColumn`	`INTEGER`	The column number the player has chosen
`PlayerChoosesColumn`	`INTEGER`	Function to get the current player’s valid choice of column
`ValidRow`	`INTEGER`	The row number that represents the first free location in the chosen column
`FindFreeRow`	`INTEGER`	Function to find the next free location in the chosen column

Table 13.08 Additional identifiers for the `PlayerMakesMove` procedure

```
FUNCTION PlayerChoosesColumn RETURNS INTEGER// returns a valid column number
    OUTPUT "Player ", ThisPlayer, "'s turn."
    REPEAT
        OUTPUT "Enter a valid column number: "
        INPUT ColumnNumber
    UNTIL ColumnNumberValid = TRUE // check whether the column number is valid
    RETURN ColumnNumber
ENDFUNCTION
```

Identifier	Data type	Explanation
ColumnNumber	INTEGER	The column number chosen by the current player
ColumnNumberValid	BOOLEAN	Function to check whether the chosen column is valid

Table 13.09 Additional identifiers for PlayerChoosesColumn function

Note that we need to define the function ColumnNumberValid. A column is valid if it is within the range 1 to 7 inclusive and there is still at least one empty location in that column.

```
FUNCTION ColumnNumberValid RETURNS BOOLEAN
  // returns whether or not the column number is valid
    Valid ← FALSE
    IF ColumnNumber >= 1 AND ColumnNumber <= 7
      THEN
        IF Board[6, ColumnNumber] = BLANK // at least 1 empty space in column
          THEN
            Valid ← TRUE
        ENDIF
    ENDIF
    RETURN Valid
ENDFUNCTION
```

Identifier	Data type	Explanation
Valid	BOOLEAN	FALSE if column number is not valid TRUE if column number is valid

Table 13.10 Additional identifier for the ColumnNumberValid function

```
FUNCTION FindFreeRow RETURNS INTEGER
  // returns the next free position
    ThisRow ← 1
    WHILE Board[ThisRow, ValidColumn] <> BLANK DO // find first empty cell
        ThisRow ← ThisRow + 1
    ENDWHILE
    RETURN ThisRow
ENDFUNCTION
```

Identifier	Data type	Explanation
ThisRow	INTEGER	Points to the next row to be checked

Table 13.11 Additional identifier for the FindFreeRow function

```
PROCEDURE CheckGameFinished
    WinnerFound ← FALSE
    CALL CheckIfPlayerHasWon
    IF WinnerFound = TRUE
      THEN
        GameFinished ← TRUE
        OUTPUT ThisPlayer " is the winner"
      ELSE
        CALL CheckForFullBoard
    ENDIF
ENDPROCEDURE
```

Note that the CheckGameFinished procedure uses two further procedures that we need to define.

Identifier	Data type	Explanation
WinnerFound	BOOLEAN	FALSE if no winning line TRUE if a winning line is found
CheckIfPlayerHasWon		Procedure to check if there is a winning line
CheckVerticalLineInValidColumn		Procedure to check if there is a winning vertical line in the column the last token was placed in
CheckForFullBoard		Procedure to check whether the board is full

Table 13.12 Additional identifiers for the CheckGameFinished procedure

```
PROCEDURE CheckIfPlayerHasWon
    WinnerFound ← False
    CALL CheckHorizontalLine
    IF WinnerFound = FALSE
      THEN
        CALL CheckVerticalLine
ENDPROCEDURE
PROCEDURE CheckHorizontalLine
    FOR i ← 1 TO 4
       IF Board[ValidRow, i] = ThisPlayer AND
         Board[ValidRow, i + 1] = ThisPlayer AND
         Board[ValidRow, i + 2] = ThisPlayer AND
         Board[ValidRow, i + 3] = ThisPlayer
         THEN
           WinnerFound ← TRUE
       ENDIF
    NEXT i
ENDPROCEDURE
PROCEDURE CheckVerticalLine
    IF ValidRow = 4 OR ValidRow = 5 OR ValidRow = 6
      THEN
        IF Board[ValidRow, ValidColumn] = ThisPlayer AND
          Board[ValidRow – 1, ValidColumn] = ThisPlayer AND
          Board[ValidRow – 2, ValidColumn] = ThisPlayer AND
          Board[ValidRow – 3, ValidColumn] = ThisPlayer
          THEN
            WinnerFound ← TRUE
        ENDIF
    ENDIF
ENDPROCEDURE
PROCEDURE CheckForFullBoard
    BlankFound ← FALSE
    ThisRow ← 0
    REPEAT
        ThisColumn ← 0
        ThisRow ← ThisRow + 1
        REPEAT
            ThisColumn ← ThisColumn + 1
            IF Board[ThisRow, ThisColumn] = BLANK
              THEN
                BlankFound ← TRUE
            ENDIF
        UNTIL ThisColumn = 7 OR BlankFound = TRUE
    UNTIL ThisRow = 6 OR BlankFound = TRUE
    IF BlankFound = FALSE
      THEN
        OUTPUT "It is a draw"
        GameFinished ← TRUE
    ENDIF
ENDPROCEDURE
```

Identifier	Data type	Explanation
BlankFound	BOOLEAN	FALSE if no blank location found on the board TRUE if a blank location found on the board
ThisRow	INTEGER	Loop counter for rows
ThisColumn	INTEGER	Loop counter for columns

Table 13.13 Additional identifiers for the `CheckForFullBoard` procedure

```
PROCEDURE SwapThisPlayer
    IF ThisPlayer = 'O'
      THEN
        ThisPlayer ← 'X'
      ELSE
        ThisPlayer ← 'O'
    ENDIF
ENDPROCEDURE
```

We can also use arrays of records. Using the Person record type from Section 13.02, we can declare an array of that type for 100 person records:

```
DECLARE Person : ARRAY[1:100] OF PersonType
```

We can then access an individual's data. For example the first person's name in the array is set as follows:

```
Person[1].Name ← "Fred"
OUTPUT Person[1].Name
```

This is particularly useful when we have several people's data to work with and do not want to use a separate 1D array for each field.

TASK 13.07

1 Declare an array of BookType (see Task 13.02) for 200 books.

2 Set the first book's details to the values given in Task 13.02.

13.06 Text files

Data need to be stored permanently. One approach is to use a file. For example, any data held in an array while your program is executing will be lost when the program stops. You can save the data out to a file and read it back in when your program requires it on subsequent executions.

A text file consists of a sequence of characters formatted into lines. Each line is terminated by an end-of-line marker. The text file is terminated by an end-of-file marker.

> **TIP**
>
> You can check the contents of a text file (or even create a text file required by a program) by using a text editor such as NotePad.

Writing to a text file

Writing to a text file usually means creating a text file.

The following pseudocode statements provide facilities for writing to a file:

```
OPENFILE <filename> FOR WRITE       // open the file for writing
WRITEFILE <filename>, <stringValue>  // write a line of text to the file
CLOSEFILE <filename>               // close file
```

Reading from a text file

An existing file can be read by a program. The following pseudocode statements provide facilities for reading from a file:

```
OPENFILE <filename> FOR READ          // open file for reading
READFILE <filename>, <stringVariable>   // read a line of text from the file
CLOSEFILE <filename>                 // close file
```

Appending to a text file

Sometimes we may wish to add data to an existing file rather than create a new file. This can be done in Append mode. It adds the new data to the end of the existing file.

The following pseudocode statements provide facilities for appending to a file:

```
OPENFILE <filename> FOR APPEND     // open file for append
WRITEFILE <filename>, <stringValue> // write a line of text to the file
CLOSEFILE <filename>              // close file
```

The end-of-file (EOF) marker

If we want to read a file from beginning to end, we can use a conditional loop. Text files contain a special marker at the end of the file that we can test for. Testing for this special end-of-file marker is a standard function in many programming languages. Every time this function is called it will test for this marker. The function will return FALSE if the end of the file is not yet reached and will return TRUE if the end-of-file marker has been reached.

In pseudocode we call this function `EOF()`. We can use the construct `REPEAT...UNTIL EOF()`. If it is possible that the file contains no data, it is better to use the construct `WHILE NOT EOF()`.

For example, the following pseudocode statements read a text file and output its contents:

```
OPENFILE "Test.txt" FOR READ
WHILE NOT EOF("Test.txt") DO
    READFILE "Test.txt", TextString
    OUTPUT TextString
ENDWHILE
CLOSEFILE "Test.txt"
```

TASK 13.08

1 Write pseudocode to save the array data from Task 13.06 to a text file.

2 Write pseudocode to read the values stored in the text file back into the board array.

13.07 Abstract Data Types (ADTs)

An **Abstract Data Type** is a collection of data and a set of associated operations:

- create a new instance of the data structure
- find an element in the data structure
- insert a new element into the data structure
- delete an element from the data structure
- access all elements stored in the data structure in a systematic manner.

The remainder of this chapter describes the following ADTs: stack, queue and linked list. It also demonstrates how they can be implemented from arrays.

In the following ADTs data items are represented as a single character, but this would normally be a set of data, possibly stored as fields in a record.

13.08 Stacks

What are the features of a stack in the real world? To make a stack, we pile items on top of each other. The item that is accessible is the one on top of the stack. If we try to find an item in the stack and take it out, we are likely to cause the pile of items to collapse.

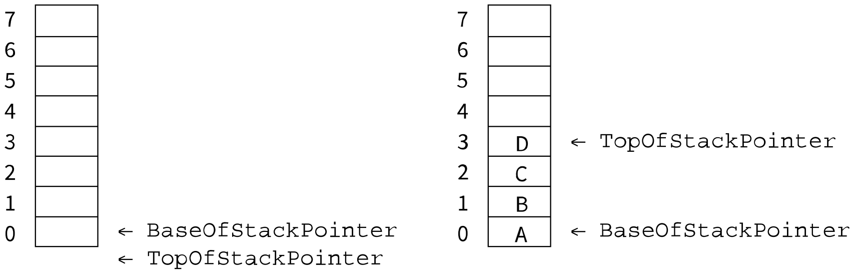


Figure 13.07 An empty stack (left) and a stack with four items pushed (right)

Figure 13.07 shows how we can represent a stack when we have added four items in this order: A, B, C, D. Note that the slots are shown numbered from the bottom as this feels more natural.

The `BaseOfStackPointer` will always point to the first slot in the stack. The `TopOfStackPointer` will point to the last element pushed (added) onto the stack. When an element is popped (removed) from the stack, the `TopOfStackPointer` will decrease to point to the element now at the top of the stack. When the stack is empty, `TopOfStackPointer` will have the value –1.

To implement this stack using a 1D array, we write:

```
DECLARE Stack:ARRAY[0 : 7] OF CHAR
```

TASK 13.09

1 Draw a diagram to show the contents of the stack shown in Figure 13.07 after “E” has been pushed onto the stack.

2 Draw a diagram to show the contents of the stack shown in Figure 13.07 after one item has been popped off the stack.

13.09 Queues

What are the features of a queue in the real world? When people form a queue, they join the queue at the end. People leave the queue from the front of the queue. If it is an orderly queue, no-one pushes in between and people don't leave the queue from any other position.

Figure 13.08 shows how we can represent a queue when five items have joined the queue in this order: A, B, C, D, E.

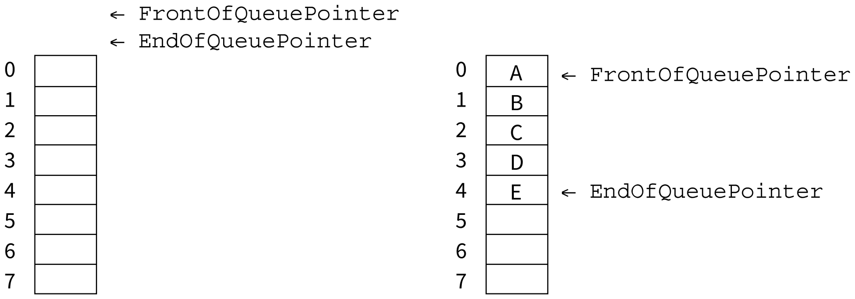


Figure 13.08 An empty queue (left) and a queue after 5 items have joined (right)

To implement a queue using an array, we can assume that the front of the queue is at position 0. When the queue is empty, the `EndOfQueuePointer` will have the value −1. When one value joins the queue, the `EndOfQueuePointer` will be incremented before adding the value to the array element where the pointer is pointing to. When the item at the front of the queue leaves, we need to move all the other items one slot forward and adjust `EndOfQueuePointer`.

TASK 13.10

1 Draw a diagram to show the contents of the queue after "F" has joined the non-empty queue shown in Figure 13.08.

2 Draw a diagram to show the contents of the queue after one item has left the non-empty queue shown in Figure 13.08.

This method involves a lot of moving of data. A more efficient way to make use of the slots is the concept of a 'circular' queue. Pointers show where the front and end of the queue are. Eventually the queue will 'wrap around' to the beginning. Figure 13.09 shows a circular queue after 11 items have joined and five items have left the queue.

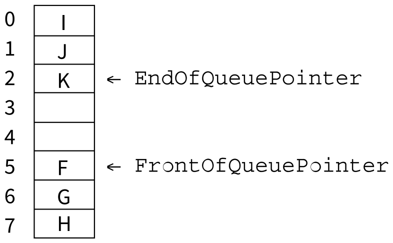


Figure 13.09 A circular queue

13.10 Linked lists

In Section 13.03 we used an array as a linear list. In a linear list, the list items are stored in consecutive locations. This is not always appropriate. Another method is to store an individual list item in whatever location is available and link the individual item into an ordered sequence using pointers.

An element of a linked list is called a node. A **node** can consist of several data items and a **pointer**, which is a variable that stores the address of the node it points to.

A pointer that does not point at anything is called a **null pointer.** It is usually represented by ∅. A variable that stores the address of the first element is called a **start pointer.**

In Figure 13.10, the data value in the node box represents the key field of that node. There are likely to be many data items associated with each node. The arrows represent the pointers. It does not show at which address a node is stored, so the diagram does not give the value of the pointer, only where it conceptually links to.

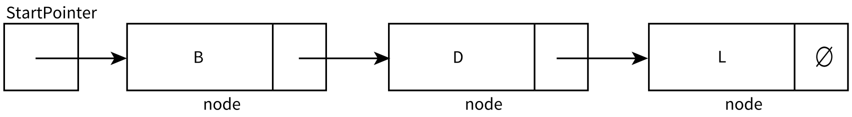


Figure 13.10 Conceptual diagram of a linked list

Figure 13.11 shows how a new node, A, is inserted at the beginning of the list. The content of `StartPointer` is copied into the new node's pointer field and `StartPointer` is set to point to the new node, A.

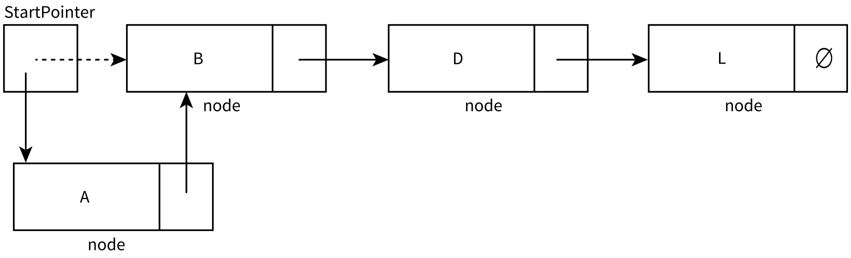


Figure 13.11 Conceptual diagram of adding a new node to the beginning of a linked list

In Figure 13.12, a new node, P, is inserted at the end of the list. The pointer field of node L points to the new node, P. The pointer field of the new node, P, contains the null pointer.

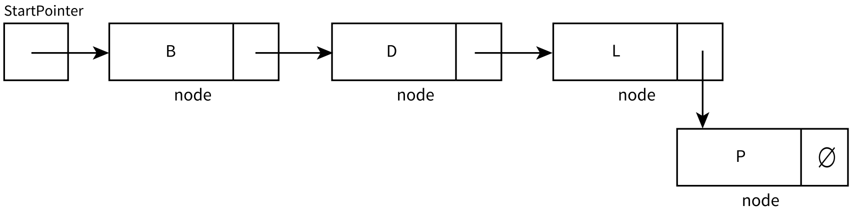


Figure 13.12 Conceptual diagram of adding a new node to the end of a linked list

To delete the first node in the list (see Figure 13.13), we copy the pointer field of the node to be deleted into `StartPointer`.

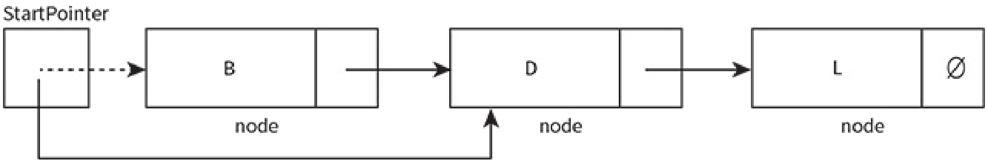


Figure 13.13 Deleting the first node in a linked list

To delete the last node in the list (see Figure 13.14), we set the pointer field for the previous node to the null pointer.

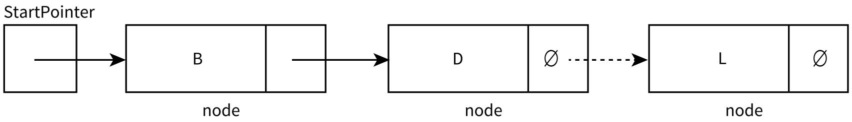


Figure 13.14 Conceptual diagram of deleting the last node of a linked list

Sometimes the nodes are linked together in order of key field value to produce an ordered linked list. This means a new node may need to be inserted or deleted from between two existing nodes.

To insert a new node, C, between existing nodes, B and D (see Figure 13.15), we copy the pointer field of node B into the pointer field of the new node, C. We change the pointer field of node B to point to the new node, C.

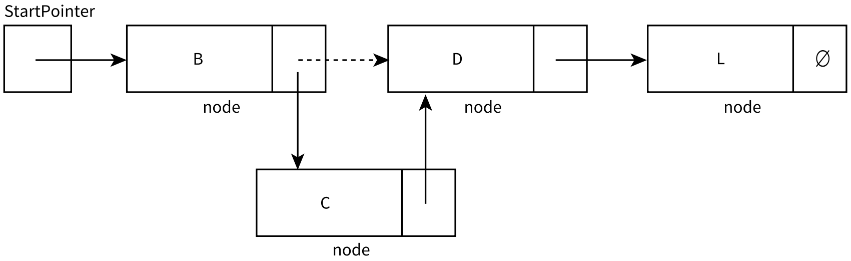


Figure 13.15 Conceptual diagram of adding a new node into a linked list

To delete a node, D, within the list (see Figure 13.16), we copy the pointer field of the node to be deleted, D, into the pointer field of node B.

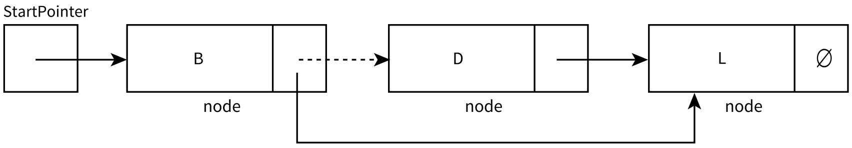


Figure 13.16 Conceptual diagram of deleting a node within a linked list

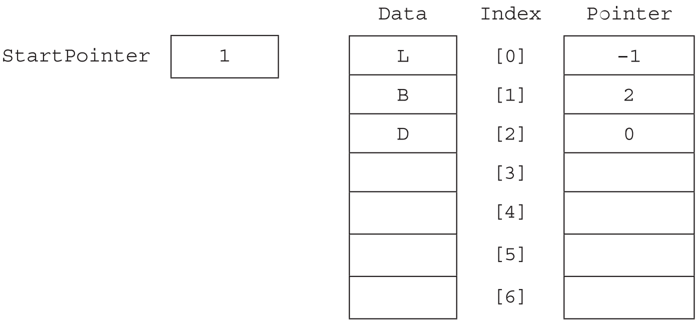
Remember that, in real applications, the data would consist of much more than a key field and one data item. This is why linked lists are preferable to linear lists. When list elements need reordering, only pointers need changing in a linked list. In a linear list, all data items would need to be moved.

Using linked lists saves time, however we need more storage space for the pointer fields.

To implement a linked list using arrays, we can use a 1D array to store the data and a 1D array to store the pointer. Reading the array values across at the same index, one row represents a node.

A value is added to the next free element of the `Data` array and pointers are adjusted to incorporate the node in the correct position within the linked list.

Figure 13.17 shows how two arrays can be used to implement the linked list from Figure 13.10.

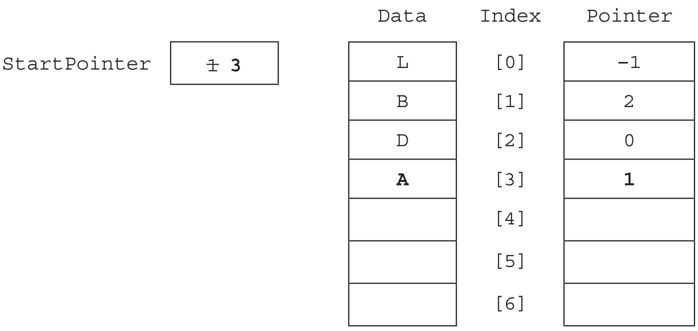


StartPointer 1

Data	Index	Pointer
L	[0]	-1
B	[1]	2
D	[2]	0
	[3]	
	[4]	
	[5]	
	[6]	

Figure 13.17 Using arrays to implement the linked list shown in Figure 13.10

Figure 13.18 shows how a new node is added to the beginning of a linked list implemented using arrays. Note that the value "A" is added at index 3 but the start pointer is adjusted to make this the new first element of the list.

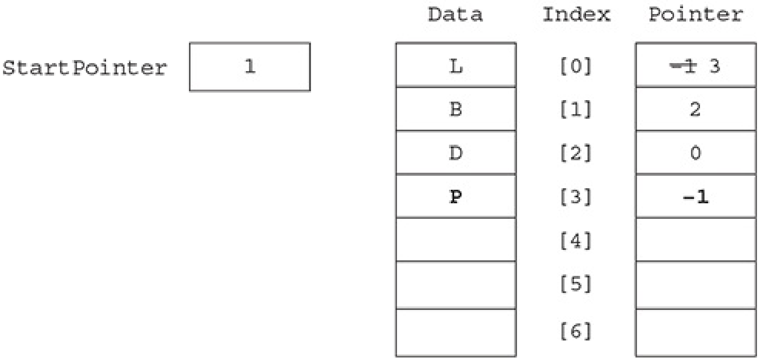


StartPointer ~~1~~ 3

Data	Index	Pointer
L	[0]	-1
B	[1]	2
D	[2]	0
A	[3]	**1**
	[4]	
	[5]	
	[6]	

Figure 13.18 Adding a new node to the beginning of a linked list

Figure 13.19 shows how a new node is added to the end of a linked list implemented using arrays. Note that the value “P” is added at index 3. The node that previously contained the null pointer (at index 0) has its pointer adjusted to point to the new node.



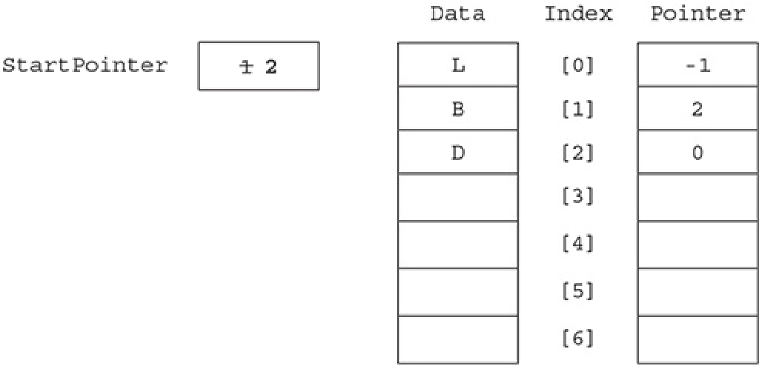
StartPointer 1

Data	Index	Pointer
L	[0]	~~-1~~ 3
B	[1]	2
D	[2]	0
P	[3]	**-1**
	[4]	
	[5]	
	[6]	

Figure 13.19 Adding a new node to the end of a linked list implemented using arrays

When deleting a node, only pointers need to be adjusted. The old data can remain in the array, but it will no longer be accessible as no pointer will point to it.

Figure 13.20 shows how the start pointer is adjusted to effectively delete the first element of the linked list. Note that the start pointer now contains the pointer value of the deleted node.

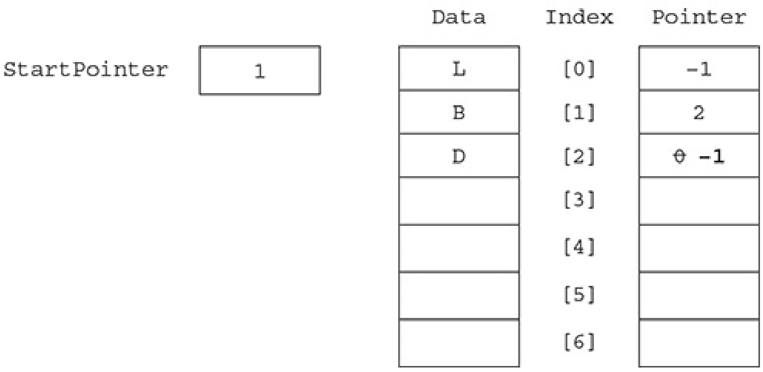


StartPointer ~~1~~ 2

Data	Index	Pointer
L	[0]	-1
B	[1]	2
D	[2]	0
	[3]	
	[4]	
	[5]	
	[6]	

Figure 13.20 Deleting the first node in a linked list implemented using arrays

Figure 13.21 shows how the pointer value of the penultimate node of the linked list is changed to the null pointer.

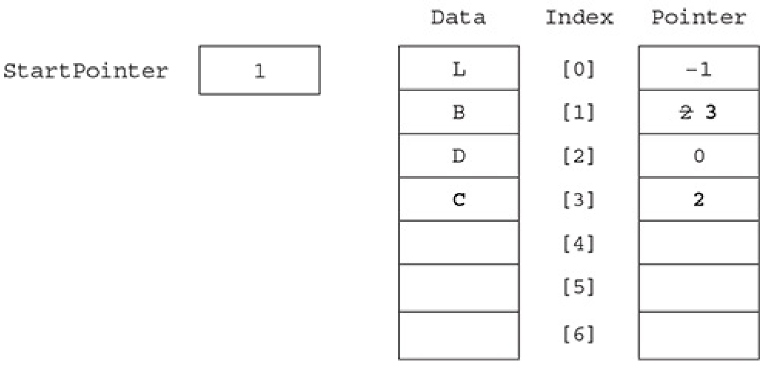


StartPointer 1

Data	Index	Pointer
L	[0]	-1
B	[1]	2
D	[2]	~~0~~ -1
	[3]	
	[4]	
	[5]	
	[6]	

Figure 13.21 Deleting the last node of a linked list implemented using arrays

When adding a node that needs to be inserted into the list, the data is added to any free element of the Data array. The pointer of the new node is set to point to the index of the node that comes after the insertion point. Note that this is the value of the pointer of the node preceding the insertion point. The pointer of the node preceding the insertion point is set to point to the new node.

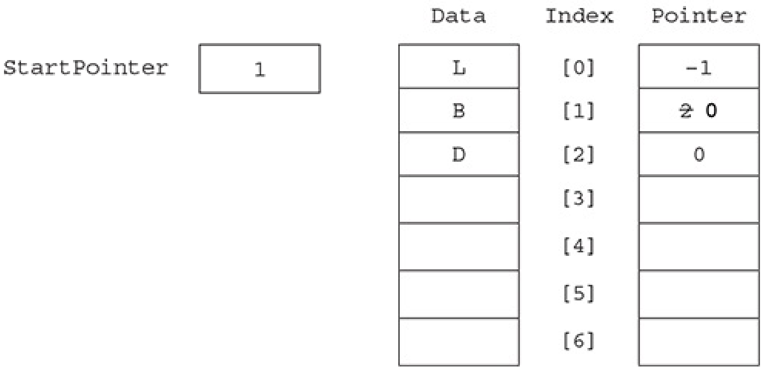


StartPointer 1

Data	Index	Pointer
L	[0]	-1
B	[1]	~~2~~ 3
D	[2]	0
C	[3]	2
	[4]	
	[5]	
	[6]	

Figure 13.22 Adding a new node into a linked list implemented using arrays

Again, when deleting a node, only pointers need to be adjusted. Figure 13.23 shows how the pointer of the node to be deleted is copied into the pointer of the preceding node.



StartPointer 1

Data	Index	Pointer
L	[0]	-1
B	[1]	~~2~~ 0
D	[2]	0
	[3]	
	[4]	
	[5]	
	[6]	

Figure 13.23 Deleting a node within a linked list implemented using arrays

Unused nodes need to be easy to find. A suitable technique is to link the unused nodes to form another linked list: the free list. Figure 13.24 shows our linked list and its free list.

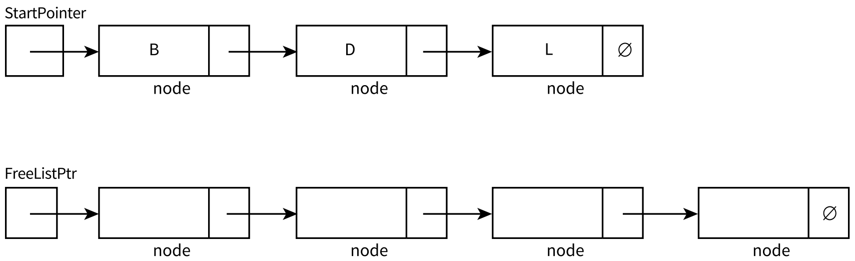


Figure 13.24 Conceptual diagram of a linked list and a free list

When an array of nodes is first initialised to work as a linked list, the linked list will be empty. So the start pointer will be the null pointer. All nodes need to be linked to form the free list. Figure 13.25 shows an example of an implementation of a linked list before any data is inserted into it.

		Data	Index	Pointer
StartPointer	-1		[0]	1
			[1]	2
FreeListPtr	0		[2]	3
			[3]	4
			[4]	5
			[5]	6
			[6]	-1

Figure 13.25 A linked list and a free list implemented using arrays

Assume “L”, “B” and “D” were added to the linked list and to be kept in alphabetical order.

Figure 13.26 shows how the values are stored in the `Data` array and the pointers of the linked list and free list adjusted.

		Data	Index	Pointer
StartPointer	1	L	[0]	-1
		B	[1]	2
FreeListPtr	3	D	[2]	0
			[3]	4
			[4]	5
			[5]	6
			[6]	-1

Figure 13.26 Linked list and free list implemented using arrays

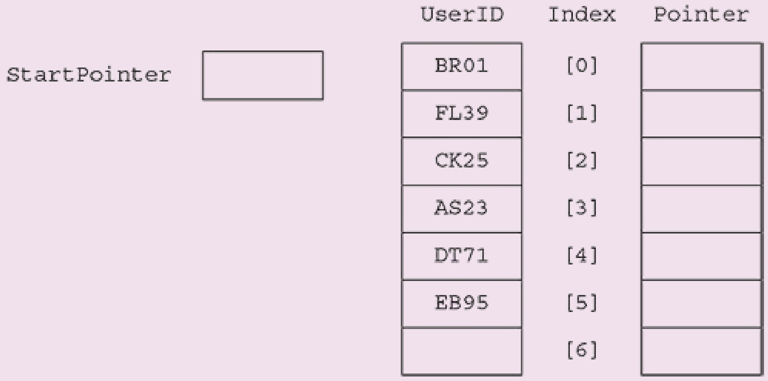
If the node containing “B” is to be deleted, the array element of that node needs to be linked back into the free list. Figure 13.27 shows how this is done by adding the node to the front of the free list.

		Data	Index	Pointer
StartPointer	1	L	[0]	-1
		B	[1]	~~2~~ **0**
FreeListPtr	~~3~~ **2**	D	[2]	3
			[3]	4
			[4]	5
			[5]	6
			[6]	-1

Figure 13.27 Linked list and free list implemented using arrays

TASK 13.11

A linked list is to be set up using the values in the `UserID` array shown below.



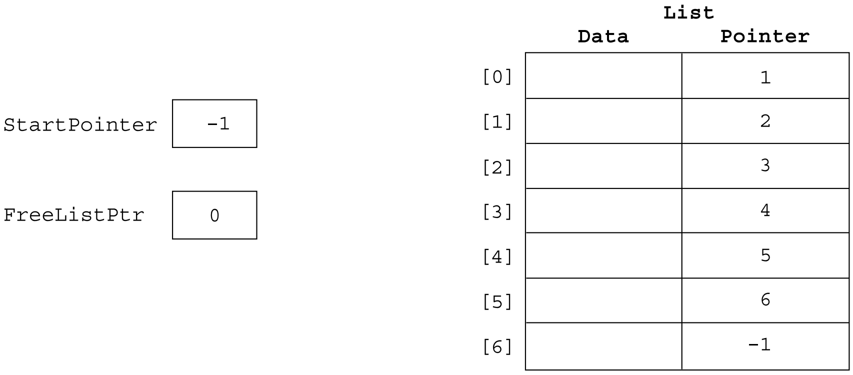
StartPointer:

UserID	Index	Pointer
BR01	[0]	
FL39	[1]	
CK25	[2]	
AS23	[3]	
DT71	[4]	
EB95	[5]	
	[6]	

Without moving any of the contents of the `UserID` array, insert the pointer values so that the linked list is in alphabetical order.

In Section 13.02 we looked at the user-defined record type. We grouped together related data items into record data structures. To use a record variable, we first define a record type. Then we declare variables of that record type.

We can store the linked list in an array of records. One record represents a node and consists of the data and a pointer (see Figure 13.28).



StartPointer: -1

FreeListPtr: 0

List

	Data	Pointer
[0]		1
[1]		2
[2]		3
[3]		4
[4]		5
[5]		6
[6]		-1

Figure 13.28 A linked list before any nodes are used

TIP

A stack can be implemented from a linked list. The start pointer is seen as the top of stack pointer and a data item is only added to the start of the linked list and a node is only removed from the start of the linked list.

TIP

A queue can be implemented from a linked list. The start pointer is seen as the front of the queue. Data items are always added to the end of the linked list and items are always removed from the start of the linked list.

Reflection Point:

What is the difference between standard data types, ADTs and user-defined data types?

Summary

- Standard data types are INTEGER, REAL, CHAR, STRING, BOOLEAN, DATE.
- A record structure holds a set of data of different data types under one identifier.
- Use the dot notation to address fields of a record.

- Arrays have dimensions with upper and lower bounds.
- Individual array elements are accessed using an index (1D arrays) or two indexes (2D arrays).
- A bubble sort algorithm compares pairs of values in a linear list and swaps them if required.
- A linear search checks each value in turn for a required value.
- Text files can be written to and read from and store data between program runs.
- Stacks, queues and linked lists are examples of Abstract Data Types (ADTs) and can be implemented using arrays.

Exam-style Questions

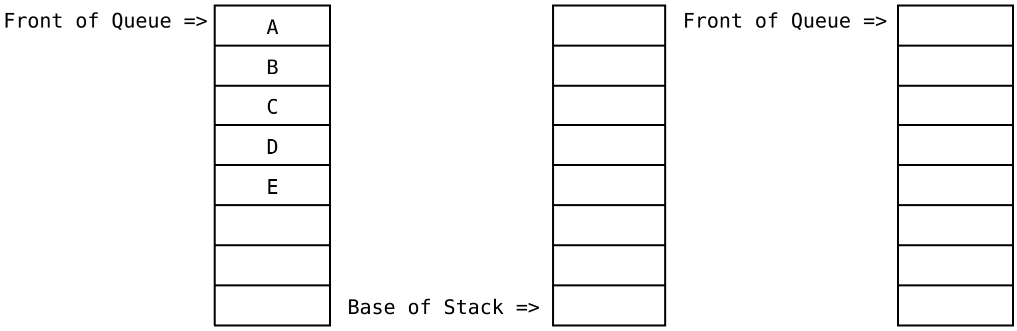
1 Complete the following variable identifier table:

Variable	**Example value**	**Data type**
`ColourCode`	***"034AB45"***	
`ProductionDate`	***2018/03/31***	
`Weight`	***67.45***	
`NumberInStock`	***98***	
`SizeCode`	***'X'***	
`Completed`	***FALSE***	

[3]

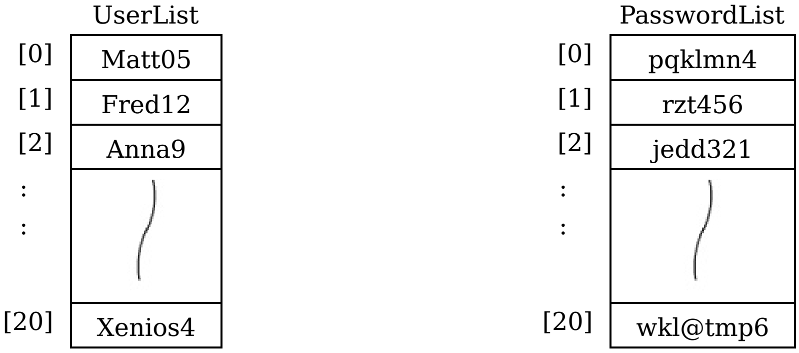
2 A stack and a queue are used to reverse the order of a set of values.

Complete the diagram:



[2]

3 Alicia uses two 1D arrays, `UserList` and `PasswordList`. For twenty users, she stores each user ID in `UserList` and the corresponding password in `PasswordList`. For example, the person with user ID `Fred12` has password `rzt456`.



Alicia wants to write an algorithm to check whether a user ID and password, entered by a user, are correct. She designs the algorithm to search `UserList` for the user ID. If the user ID is found, the password stored in `PasswordList` is to be compared to the entered password. If the passwords match, the login is successful. In all other cases, login is unsuccessful.

a Complete the identifier table. [8]

Identifier	**Data type**	**Explanation**
`UserList`		1D array to store user IDs
....		1D array to store passwords
`MaxIndex`		Upper bound of the array
`MyUserID`		User ID entered to login

MyPassword		....
UserIdFound		FALSE if user ID not found in UserList TRUE if ....
LoginOK		FALSE if .... TRUE if ....
Index		Pointer to current array element

b Complete the pseudocode for Alicia’s algorithm:

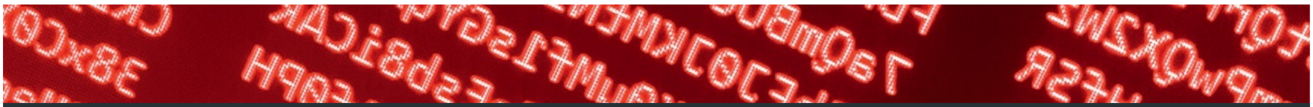
```
MaxIndex ← 20
INPUT MyUserID
INPUT MyPassword
UserIdFound ← FALSE
LoginOK ← ...............
Index ← −1
REPEAT
    INDEX ← ...............
    IF UserList[...............] = ...............
      THEN
        UserIdFound ← TRUE
    ENDIF
UNTIL ............... OR ...............
IF UserIdFound = TRUE
  THEN
    IF PasswordList[...............] = ...............
      THEN
        LoginOK ← TRUE
    ...............
ENDIF
IF ...............
  THEN
    OUTPUT "Login successful"
  ELSE
    OUTPUT "User ID and/or password incorrect"
ENDIF
```

[10]

c **i** Instead of using two 1D arrays, Alicia could have used an array of records.

Write pseudocode to declare the record structure `UserRecord`. [2]

ii Write pseudocode to declare the `User` array. [2]

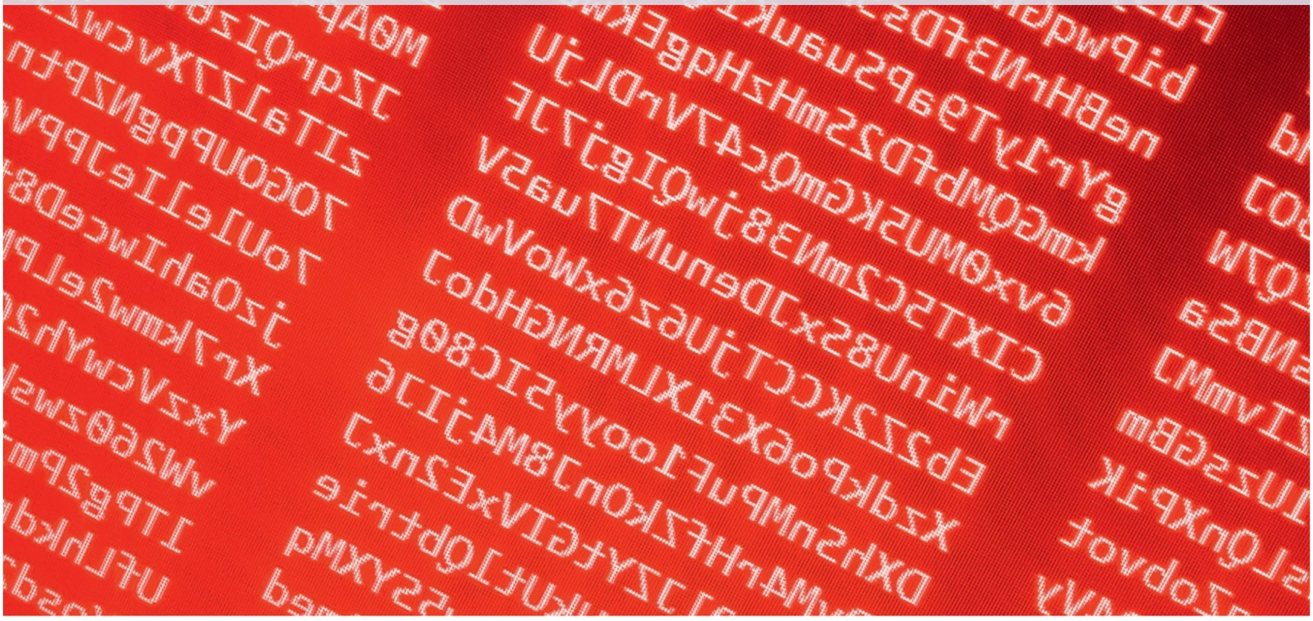


Chapter 14:
Programming and data representation

Learning objectives

By the end of this chapter you should be able to:

- write a program in a high-level language (Python, Visual Basic console mode, Java)
- implement and write pseudocode from a given design presented as either a program flowchart or structured English
- write pseudocode and program statements for:
  - the declaration of variables and constants
  - the assignment of values to variables and constants
  - expressions involving any of the arithmetic or logical operators
  - input from the keyboard and output to the console
- use a subset of the built-in functions and library routines supported by the chosen programming language, including those used for string/character manipulation and random number generator
- use an ‘IF’ structure including the ‘ELSE’ clause and nested IF statements
- use a ‘CASE’ structure
- use a loop (‘count controlled’, ‘post-condition’, ‘pre-condition’)
- justify why one loop structure may be better suited to a problem than the others
- define and use a procedure and explain where in the construction of an algorithm it is appropriate to use a procedure
- use parameters
- show understanding of passing parameters by reference and by value
- define and use a function and explain where in the construction of an algorithm it is appropriate to use a function
- show understanding that a function is used in an expression (the return value replaces the call)
- use the terminology associated with procedures and functions: procedure/function header, procedure/function interface, parameter, argument, return value
- write efficient code.



14.01 Programming languages

Chapters 12 and 13 introduced the concept of solving a problem and representing a solution using a flowchart or pseudocode. We expressed our solutions using the basic constructs: assignment, sequence, selection, iteration, input and output.

To write a computer program, we need to know the syntax (the correct structure of statements) of these basic constructs in our chosen programming language. This chapter introduces syntax for Python, Visual Basic console mode and Java.

Note that for convenience and easy reference, definitive pseudocode syntax is repeated in this chapter at the appropriate points.

You only need learn to program in one of the three languages covered in this book. Programming language is only examined at A Level but it is important to start learning it from AS Level to ensure you are well-prepared. For the AS Level exams you should use pseudocode rather than programming language.

TIP

The only way of knowing whether the algorithm you have designed is a suitable solution to the problem you are trying to solve, is to implement your pseudocode in your chosen programming language and test the program by running it.

Python

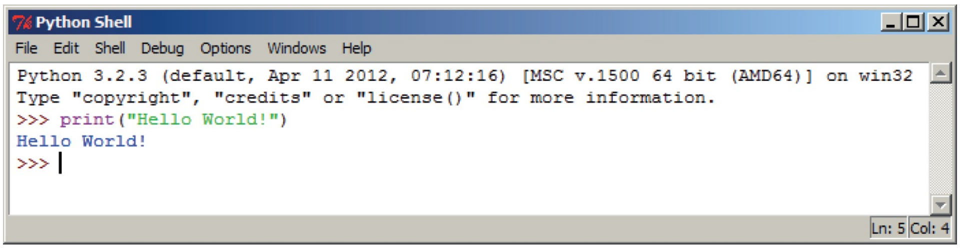
Python was conceived by Guido van Rossum in the late 1980s. Python 2.0 was released in 2000 and Python 3.0 in 2008. Python is a multi-paradigm programming language, meaning that it fully supports both object-oriented programming and structured programming. Many other paradigms, including logic programming, are supported using extensions. These paradigms are covered in Chapters 26, 27 and 29.

The Python programs in this book have been prepared using Python 3 (see Python for a free download) and Python's Integrated DeveLopment Environment (IDLE).

Key characteristics of Python include the following.

- Every statement must be on a separate line.
- Indentation is significant. This is known as the 'off-side rule'.
- Keywords are written in lower case.
- Python is case sensitive: the identifier `Number1` is seen as different from `number1` or `NUMBER1`.
- Everything in Python is an object (see Chapter 27).
- Code makes extensive use of a concept called 'slicing' (see Section 14.07).
- Programs are interpreted (see Chapter 8, Section 8.05 for information on interpreted and compiled programs).

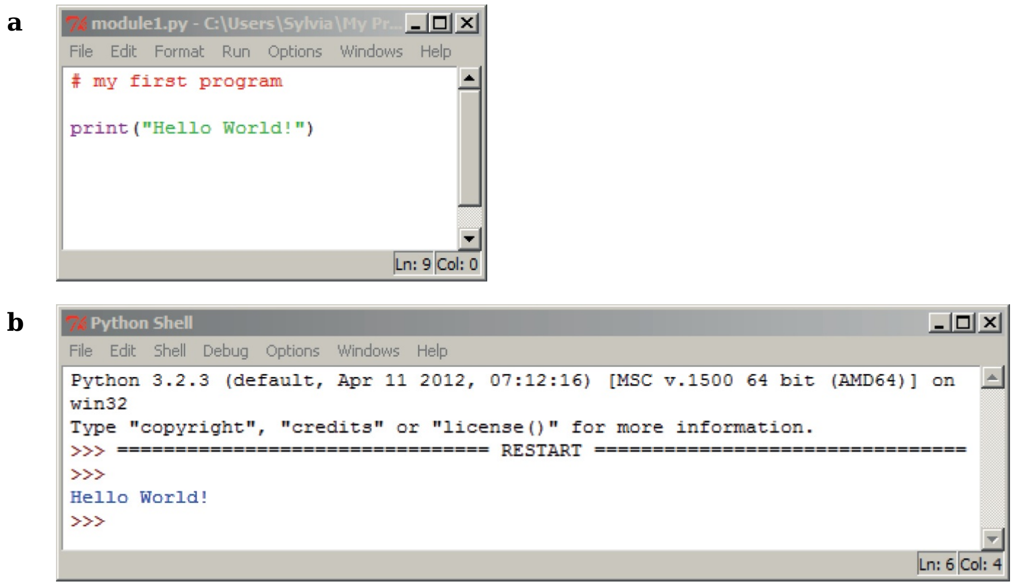
You can type a statement into the Python Shell and the Python interpreter will run it immediately (see Figure 14.01).



```
Python 3.2.3 (default, Apr 11 2012, 07:12:16) [MSC v.1500 64 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> print("Hello World!")
Hello World!
>>> 
```

Figure 14.01 Running a statement in the Python shell

You can also type program code into a Python editor (such as IDLE), save it with a .py extension and then run the program code from the Run menu in the editor window (see Figure 14.02).


a
module1.py - C:\Users\Sylvia\My Pr...
File Edit Format Run Options Windows Help
Ln: 9 Col: 0
b
Python Shell
File Edit Shell Debug Options Windows Help
Ln: 6 Col: 4

```
# my first program

print("Hello World!")
```

```
Python 3.2.3 (default, Apr 11 2012, 07:12:16) [MSC v.1500 64 bit (AMD64)] on
win32
Type "copyright", "credits" or "license()" for more information.
>>> ================================ RESTART ================================
>>>
Hello World!
>>>
```

Figure 14.02 (a) A saved program in the Python editor window and (b) running in the Python shell

Visual Basic Console Mode (VB.NET)

VB.NET is a multi-paradigm, high-level programming language, implemented on the .NET Framework. Microsoft launched VB.NET in 2002 as the successor to its original Visual Basic language. Microsoft’s integrated development environment (IDE) for developing in VB.NET is Visual Studio. Visual Studio Express and Visual Studio Community are freeware.

The Visual Basic programs in this book have been prepared using Microsoft Visual Basic 2010 Express Console Application. (Free download available from Visual Studio Express)

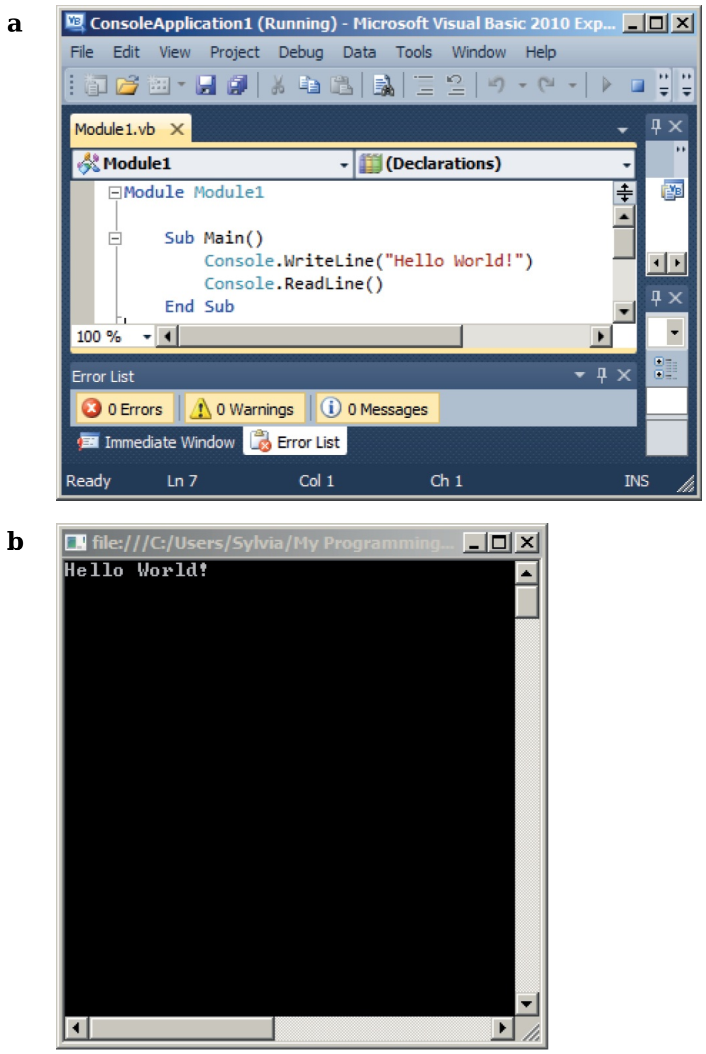
Key characteristics of VB.NET include the following.

- Every statement should be on a separate line. Statements can be typed on the same line with a colon (:) as a separator. However, this is not recommended.
- Indentation is good practice.
- VB.NET is not case sensitive. Modern VB.NET editors will automatically copy the case from the first definition of an identifier.
- The convention is to use CamelCaps (also known as PascalCaps) for identifiers and keywords.
- Programs need to be compiled (see Chapter 8, Section 8.05 for information on interpreted and compiled programs).

You type your program code into the Integrated Development Environment (IDE) as shown in Figure 14.03 (a), save the program code and then click on the Run button ▶. This starts the compiler. If there are no syntax errors the compiled program will then run. Output will be shown in a separate console window (see Figure 14.03 (b)).

Note that the console window shuts when the program has finished execution. To keep the console window open so you can see the output (see Figure 14.03), the last statement of your program should be

```
Console.ReadLine()
```



a
ConsoleApplication1 (Running) - Microsoft Visual Basic 2010 Exp...
File Edit View Project Debug Data Tools Window Help
Module1.vb
Module1
(Declarations)
100 %
Error List
0 Errors
0 Warnings
0 Messages
Immediate Window
Error List
Ready Ln 7 Col 1 Ch 1 INS
b
file:///C:/Users/Sylvia/My Programming...

```
Module Module1

    Sub Main()
        Console.WriteLine("Hello World!")
        Console.ReadLine()
    End Sub
```

```
Hello World!
```

Figure 14.03 (a) A saved program in the VB.NET editor and (b) running in the program execution (console) window

Java

Java was originally developed by James Gosling at Sun Microsystems (now owned by Oracle) and released in 1995. The Java Runtime Environment (JRE) is intended for end users, and the Java Development Kit (JDK) is intended for software developers and includes development tools such as the Java compiler and a debugger.

Java was intended to be platform independent. The Java programs in this book have been prepared using NetBeans 8.2. However, any text editor can be used to write Java source code.

Key characteristics of Java include the following.

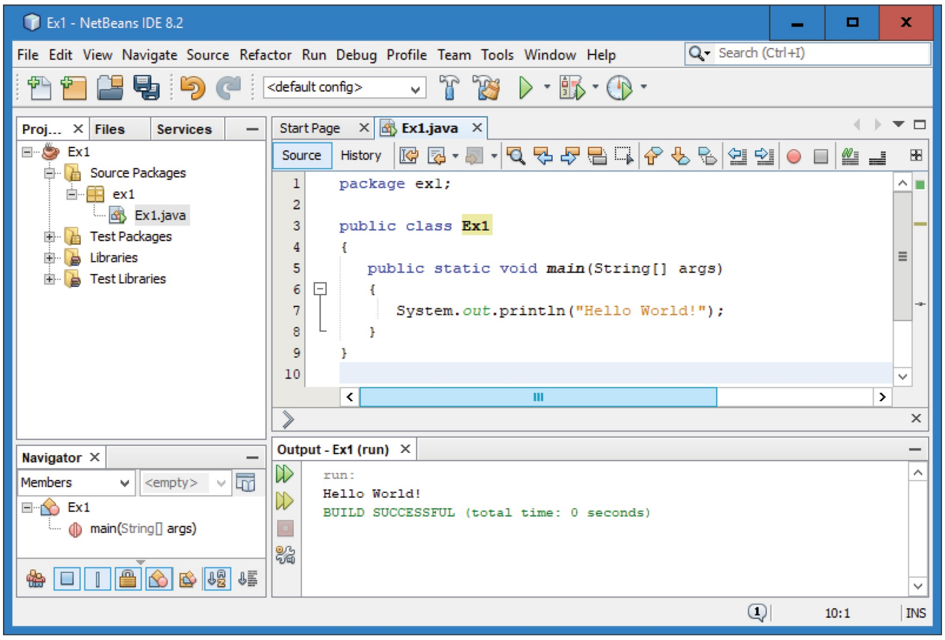
- Every statement ends with a semicolon (;). More than one statement can go on a single line, but this is not recommended.
- Indentation is good practice.
- Java is case sensitive.
- The convention is to use camelCaps for identifiers, lower case for keywords and capitalised identifiers for classes.
- A compound statement consists of a sequence of statements enclosed between braces { }.
- Whenever Java syntax requires a statement, a compound statement can be used.
- Programs need to be compiled (see Chapter 8, Section 8.05 for information on interpreted and compiled programs) into bytecode and then run using the Java Virtual Machine.

Java was designed almost exclusively as an object-oriented language. All code is written inside classes. Only the simple data types (such as integer, real) are not objects. Strings are objects.

Source files must be named after the public class they contain, appending the suffix .java, for example, Ex1.java. It must first be compiled into bytecode, using a Java compiler, producing a file named Ex1.class. Only then can it be executed.

The method name “main” is not a keyword in the Java language. It is simply the name of the method the Java launcher calls to pass control to the program.

You type your program statements into the Integrated Development Environment (IDE) as shown in Figure 14.04, save the program code and then click on the Run button (▶). This starts the compiler. If there are no syntax errors the compiled program code will then run. Output will be shown in the Output window (see Figure 14.04).



```
package ex1;

public class Ex1
{
    public static void main(String[] args)
    {
        System.out.println("Hello World!");
    }
}
```

```
run:
Hello World!
BUILD SUCCESSFUL (total time: 0 seconds)
```

Figure 14.04 A Java program in the NetBeans editor and running in the Output window

14.02 Programming basics

Declaration of variables

Most programming languages require you to declare the type of data to be stored in a variable, so the correct amount of memory space can be reserved by the compiler. A variable declared to store a whole number (integer) cannot then be used to store alphanumeric characters (strings), or the other way around. VB.NET and Java require variables to be declared before they are used.

Python handles variables differently to most programming languages. It tags values. This is why Python does not have variable declarations.

> **TIP**
>
> It is good programming practice to include a comment about the variables you are planning to use and the type of data you will store in them.

In pseudocode, variable declarations are written as:

```
DECLARE <identifier> : <dataType>
```

For example, you may declare the following variables:

```
DECLARE Number1 : INTEGER // this declares Number1 to store a whole number
DECLARE YourName : STRING // this declares YourName to store a
                          // sequence of characters
DECLARE N1, N2, N3 : INTEGER  // declares 3 integer variables
DECLARE Name1, Name2 : STRING // declares 2 string variables
```

Syntax definitions

The syntax of variable declarations in language code is as follows:

Python	Python does not have variable declarations
VB.NET	`Dim` <identifier>[, <identifier>] `As <dataType>` Each line of declarations must start with the keyword `Dim`.
Java	`<datatype> <identifier>[, <identifier>];`

Code examples

Python	`# Number1 of type Integer`
	`# YourName of type String`
	`# N1, N2, N3 of type integer;`
	`# Name1, Name2 of type string;`
		There are no declarations, but comments should be made at the beginning of a module (see the section about comments at the end of Section 14.02).

VB.NET	`Dim Number1 As Integer`
	`Dim YourName As Integer`
	`Dim N1, N2, N3 As Integer`
	`Dim Name1, Name2 As String`
		You can group more than one variable of the same type on the same line.

Java	`int number1;`
	`String yourName;`
	`int n1, n2, n3;`
	`String name1, name2;`
		You can group more than one variable of the same type on the same line.

Declaration and assignment of constants

Sometimes we use a value in a solution that never changes, for example, the value of the mathematical constant pi (π). Instead of using the actual value in program statements, it is good practice and helps readability, if we give a constant value a name and declare it at the beginning of the program.

In pseudocode, constant declarations are written as:

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

For example:

```
CONSTANT Pi = 3.14
```

Syntax definitions

Python	**`<identifier> = <value>`**
VB.NET	`Const` **`<identifier> = <value>`** Each line of declarations must start with the keyword `Const`.

Java	`static final <datatype> <identifier> = <value>;`
	Each line of constant declarations must start with the keywords
	`static final`

Code examples

Python	`PI = 3.14`	Python convention is to write constant identifiers using uppercase only. The values can be changed, although you should treat constants as not changeable.
VB.NET	`Const Pi = 3.14`	The value of a constant in VB.NET cannot be altered within the program.
Java	`static final double PI = 3.14;`	The value of a constant in Java cannot be altered within the program.

Assignment of variables

Once we have declared a variable, we can assign a value to it (See Chapter 12, Section 12.05).

In pseudocode, assignment statements are written as:

```
<identifier> ← <expression>
```

For example:

```
A ← 34
B ← B + 1
```

Syntax definitions and code examples

Python	`<identifier> = <expression>`	`A = 34`
		`B = B + 1`
			The assignment operator is =

VB.NET	`<identifier> = <expression>`	`A = 34`
		`B = B + 1`
			The assignment operator is =

Java	`<identifier> = <expression>;`	`A = 34;`
		`B = B + 1;`
			The assignment operator is =

VB.NET allows you to initialise a variable as part of the declaration statement, for example:

```
Dim Number1 As Integer = 0
```

Java allows you to initialise a variable as part of the declaration statement, for example:

```
int number1 = 0;
```

VB.NET and Python allow increment statements such as B = B + 1 to be written as B += 1.

Java allows increment statements such as `b = b + 1` to be written as `b++;`

Arithmetic operators

Assignments don't just give initial values to variables. We also use an assignment when we need to store the result of a calculation. The arithmetic operators used for calculations are shown in Table 14.01.

Operation	Pseudocode	Python	VB.NET	Java
Addition	+	+	+	+
Subtraction	-	-	-	-
Multiplication	*	*	*	*
Division	/	/	/	/ (when dividing float or double types)
Exponent	^	**	^	No operator available, only method: `Math.pow(n,e)`
Integer division	`DIV`	`//`	`\`	/ (when dividing integer types)
Modulus	`MOD`	`%`	`Mod`	`%`

Table 14.01 Arithmetic operators

TIP

The result of integer division is the whole number part of the division. For example, 7 DIV 2 gives 3.

TIP

The result of the modulus operation is the remainder of a division. For example, 7 MOD 2 gives 1.

When more than one operator appears in an expression, the order of evaluation depends on the mathematical **rules of precedence**: parentheses, exponentiation, multiplication, division, addition, subtraction.

Question 14.01

Evaluate each of the following expressions:

4 * 3 – 3 ^ 2

(4 * 3 – 3) ^ 2

4 * (3 – 3) ^ 2

4 * (3 – 3 ^ 2)

Outputting information to the screen

In pseudocode, output statements are written as:

```
OUTPUT <string>
OUTPUT <identifier(s)>
```

When outputting text and data to the console screen, we can list a mixture of output strings and variable values in the print list.

Syntax definitions

Python	`print(<printlist>)`
	`print(<printlist>, end ='')`
		Print list items are separated by commas (,). To avoid moving onto the next line after the output, use `end ='')`

VB.NET	`Console.WriteLine(<printlist>)`
	`Console.Write(<printlist>)`
		Print list items are joined using &.
		`Console.WriteLine` will move onto the next line after the output; `Console.Write` will remain on the same line.

Java	`System.out.print(<printlist>);`
	`System.out.println(<printlist>);`
		Print list items are joined using +.
		`System.out.println` will move onto the next line after the output; `System.out.print` will remain on the same line.

In the examples below, the print list consists of four separate items:

“Hello ” and “. Your number is ” are strings and

YourName and Number1 are variables, for which we print the value.

In pseudocode, we can indicate whether a new line should be output at the end by a comment at the end of the statement.

```
OUTPUT "Hello ", YourName, ". Your number is ", Number1 // newline
OUTPUT "Hello " // no new line
```

Code examples

Python	`print("Hello ", YourName,`
	`   ". Your number is ", Number1)`
	`print("Hello ", end= '')`

VB.NET	`Console.WriteLine("Hello " & YourName &`
	`   ". Your number is " & Number1)`
	`Console.Write("Hello")`

Java	`System.out.println("Hello " + yourName +`
	`   ". Your number is " + number1);`
	`System.out.print("Hello");`

In the code examples above you can see how output statements can be spread over more than one line when they are very long. You must break the line between two print list items. You cannot break in the middle of a string, unless you make the string into two separate strings.

In Python and VB.NET you can also use the placeholder method for output: the variables to be printed are represented by sequential numbers in { } in the message string and the variables are listed in the correct order after the string, separated by commas:

Python	`print ("Hello {0}. Your number is {1}".format(YourName, Number1))`
VB.NET	Console.WriteLine("Hello {0}. Your number is {1}", YourName, Number1)

Getting input from the user

When coding an input statement, it is good practice to prompt the user as to what they are meant to enter. For example, consider the pseudocode statement:

```
  INPUT "Enter a number: " A
```

Note the space between the colon and the closing quote. This is significant. It gives a space before the user types their input.

Code examples

Python	`A = input("Enter a number: ")`	The prompt is provided as a parameter to the input function. Single quotes are also accepted. All input is taken to be a string; if you want to use the input as a number the input string has to be converted using a function (see Section 14.07).

VB.NET	`Console.Write("Enter a number: ")`
	`A = Console.ReadLine()`
		The prompt has to be supplied as an output statement separately.

Java	`import java.util.Scanner;`
	`Scanner console = new Scanner(System.in);`
	`System.out.print("Enter a number: ");`
	`a = console.next();`
		The Scanner class has to be imported from the Java library first and a scanner object has to be created before it can be used to read an input string.
		The prompt has to be supplied as an output statement separately.

Comments

It is good programming practice to add comments to explain code where necessary.

Code examples

Python	`# this is a comment`
	`# this is another comment`

VB.NET	`' this is a comment`
	`' this is another comment`

Java	`// this is a comment`
	`// this is another comment`
	`/* this is a multi-line`
	`   comment`
	`*/`

TASK 14.01

Use the IDE of your chosen programming language (in future just referred to as 'your language'). Type the program statements equivalent to the following pseudocode (you may need to declare the variable `YourName` first):

```
INPUT "What is your name? " YourName
OUTPUT "Have a nice day ", YourName
```

Save your program as Example1 and then run it. Is the output as you expected?

14.03 Data types

Every programming language has built-in data types. Table 14.02 gives a subset of those available. The number of bytes of memory allocated to a variable of the given type is given in brackets for VB.NET and Java.

Description of data	Pseudocode	Python	VB.NET	Java
Whole signed numbers	`INTEGER`	`int`	`Integer` (4 bytes)	`int` (4 bytes)
Signed numbers with a decimal point	`REAL`	`float`	`Single` (4 bytes) `Double` (8 bytes)	`float` (4 bytes) `double` (8 bytes)
A single character	`CHAR` Use single (`'`) quotation marks to delimit a character	Not available	`Char` (2 bytes – Unicode)	`char` (2 bytes – Unicode)
A sequence of characters (a string)	`STRING` Use double (`"`) quotation marks to delimit a string.	`str` (stored as ASCII but Unicode strings are also available) Use single (`'`), double (`"`) or triple (`'''` or `"""`) quotation marks to delimit a string.	`String` (2 bytes per character) Use double (`"`) quotation marks to delimit a string.	`String` (2 bytes per character) Use double (`"`) quotation marks to delimit a string.

Logical values: True (represented as 1) and False (represented as 0)	`BOOLEAN`	`bool`
		possible values:
		`True`
		`False`
			`Boolean` (2 bytes)
			possible values:
			`True`
			`False`
				`Boolean`
				possible values:
				`true`
				`false`

Table 14.02 Simple data types

In Python, a single character is represented as a string of length 1.

In VB.NET, each character in a string requires two bytes of memory and each character is represented in memory as Unicode (in which, the values from 1 to 127 correspond to ASCII).

Date has various internal representations but is output in conventional format.

Description of data	Pseudocode	Python	VB.NET	Java
Date value	`DATE`	Use the datetime class	`Date` (8 bytes)	Date is a class in Java. To make use of it use: `import java.util.Date;`

Table 14.03 The Date data types

In Python and Java, date is not available as a built-in data type. Date is provided as a class (see Table 14.03).

VB.NET stores dates and times from 1.1.0001 (0 hours) to 31.12.9999 (23:59:59 hours) with a resolution of 100 nanoseconds (this unit is called a ‘tick’). Floating-point (decimal) numbers are stored in binary-coded decimal format (see Section 1.02).

There are many more data types. Programmers can also design and declare their own data types (see Chapter 16 (Section 16.01) and Chapter 26 (Section 26.01).

TASK 14.02

1. Look at the identifier tables in Chapter 12 (Tables 12.02 and 12.04 to 12.12). Decide which data type from your language is appropriate for each variable listed.
2. Write program code to implement the pseudocode from Worked Example 12.01 in Chapter 12.

14.04 Boolean expressions

In Chapter 12 (Section 12.06), we covered logic statements. These were statements that included a condition. Conditions are also known as Boolean expressions and evaluate to either True or False. True and False are known as Boolean values.

Simple Boolean expressions involve comparison operators (see Table 14.04). Complex Boolean expressions also involve Boolean operators (see Table 14.05).

Operation	Pseudocode	Python	VB.NET	Java
equal	`=`	`==`	`=`	`==`
not equal	`<>`	`!=`	`<>`	`!=`
greater than	`>`	`>`	`>`	`>`
less than	`<`	`<`	`<`	`<`
greater than or equal to	`>=`	`>=`	`>=`	`>=`
less than or equal to	`<=`	`<=`	`<=`	`<=`

Table 14.04 Comparison operators

Operation	Pseudocode	Python	VB.NET	Java
AND (logical conjunction)	`AND`	`and`	`And`	`&&`
OR (logical inclusion)	`OR`	`or`	`Or`	`||`
NOT (logical negation)	`NOT`	`not`	`Not`	`!`

Table 14.05 Boolean operators

14.05 Selection

IF...THEN statements

In pseudocode the IF...THEN construct is written as:

```
IF <Boolean expression>
  THEN
    <statement(s)>
ENDIF
```

Syntax definitions

Python	`if <Boolean expression>:`
	`  <statement(s)>`
		Note that the `THEN` keyword is replaced by a colon (:). Indentation is used to show which statements form part of the conditional statement.

VB.NET	`If <Boolean expression> Then`
	`  <statement(s)>`
	`End If`
		Note the position of `Then` on the same line as the Boolean expression. The `End If` keywords should line up with the `If` keyword.

Java	`if (<Boolean expression>)`
	`  <statement>;`
		Note that the Boolean expression is enclosed in brackets.
		If more than one statement is required as part of the conditional statement, the statements must be enclosed in braces `{ }`.

Pseudocode example:

```
IF x < 0
  THEN
    OUTPUT "Negative"
ENDIF
```

Code examples

Python	`if x < 0:`
	`  print("Negative")`

VB.NET	`If x < 0 Then`
	`  Console.WriteLine("Negative")`
	`End If`

Java	`if (x < 0)`
	`  System.out.println("Negative");`

TASK 14.03

Write program code to implement the pseudocode from Worked Example 12.03 in Chapter 12.

IF...THEN...ELSE statements

In pseudocode, the IF...THEN...ELSE construct is written as:

```
IF <Boolean expression>
  THEN
    <statement(s)>
  ELSE
    <statement(s)>
ENDIF
```

Syntax definitions

Python	`if <Boolean expression>:`
	`  <statement(s)>`
	`else:`
	`  <statement(s)>`
		Indentation is used to show which statements form part of the conditional statement; the `else` keyword must line up with the corresponding `if` keyword.

VB.NET	`If <Boolean expression> Then`
	`  <statement(s)>`
	`Else`
	`  <statement(s)>`
	`End If`
		The `Else` keyword is on its own on a separate line. It is good programming practice to line it up with the corresponding `If` keyword and indent the statements within the conditional statement.

Java	`if (<Boolean expression>)`
	`  <statement>;`
	`else`
	`  <statement>;`
		If more than one statement is required in the `else` part of the statement, the statements must be enclosed in braces `{ }`.

Pseudocode example:

```
IF x < 0
  THEN
    OUTPUT "Negative"
  ELSE
    OUTPUT "Positive"
ENDIF
```

Code examples

Python	`if x < 0:`
	`  print("Negative")`
	`else:`
	`  print("Positive")`

VB.NET	`If x < 0 Then`
	`  Console.WriteLine("Negative")`
	`Else`
	`  Console.WriteLine("Positive")`
	`End If`

Java	`if (x < 0)`
	`  System.out.println("Negative");`
	`else`
	`  System.out.println("Positive");`

Nested IF statements

In pseudocode, the nested IF statement is written as:

```
IF <Boolean expression>
  THEN
    <statement(s)>
  ELSE
    IF <Boolean expression>
      THEN
        <statement(s)>
      ELSE
        <statement(s)>
    ENDIF
ENDIF
```

Syntax definitions

Python	`if <Boolean expression>:`
	`  <statement(s)>`
	`elif <Boolean expression>:`
		Note the keyword `elif` (an abbreviation of `else if`). This keyword must line up

	`  <statement(s)>`
	`else:`
	`  <statement(s)>`
		with the corresponding `if`.
		There can be as many `elif` parts to this construct as required.

VB.NET	`If <Boolean expression> Then`
	`  <statement(s)>`
	`ElseIf`
	`  <statement(s)>`
	`Else`
	`  <statement(s)>`
	`End If`
		If `ElseIf` is used as one word, only one `End If` is required at the end of this construct.
		There can be as many `ElseIf` parts as required.

Java	`if (<Boolean expression>)`
	`  <statement>;`
	`else if (<Boolean expression>)`
	`  <statement>;`
	`else`
	`  <statement>;`	

Pseudocode example:

```
IF x < 0
  THEN
    OUTPUT "Negative"
  ELSE
    IF x = 0
      THEN
        OUTPUT "Zero"
      ELSE
        OUTPUT "Positive"
    ENDIF
ENDIF
```

Code examples

Python	`if x < 0:`
	`  print("Negative")`
	`elif x == 0:`
	`   print("Zero")`
	`else:`
	`   print("Positive")`

VB.NET	`If x < 0 Then`
	`   Console.WriteLine("Negative")`
	`ElseIf x = 0 Then`
	`   Console.WriteLine("Zero")`
	`Else`
	`   Console.WriteLine("Positive")`
	`End If`

Java	`if (x < 0)`
	`{`
	`   System.out.println('Negative');`
	`}`
	`else if (x == 0)`
	`{`
	`   System.out.println('Zero');`
	`}`
	`else`
	`{`
	`   System.out.println('Positive');`
	`}`

TASK 14.04

Write program code to implement the pseudocode from Worked Example 12.02 in Chapter 12.

CASE statements

An alternative selection construct is the CASE statement. Each considered CASE condition can be:

- a single value
- single values separated by commas
- a range.

In pseudocode, the CASE statement is written as:

```
CASE OF <expression>
  <value1>             : <statement(s)>
  <value2>,<value3>    : <statement(s)>
  <value4> TO <value5> : <statement(s)>
  .
  .
  OTHERWISE <statement(s)>
ENDCASE
```

The value of `<expression>` determines which statements are executed. There can be as many separate cases as required. The `OTHERWISE` clause is optional and useful for error trapping.

Syntax definitions

Python	Python does not have a CASE statement. You need to use nested If statements instead.

VB.NET	`Select Case <expression>`
	`  Case value1`
	`    <statement(s)>`
	`  Case value2,value3`
	`    <statement(s)>`
	`  Case value4 To value5`
	`    <statement(s)>`
	`  .`
	`  .`
	`   .`
	`  Case Else`
	`    <statement(s)>`
	`End Select`

Java	`switch (<expression>)`
	`{`
	`  case value1:`
	`    <statement(s)>;`
	`    break;`
	`  case value2: case value3:`
	`    <statement(s)>;`
	`    break;`
	`  .`
	`  .`
	`  .`
	`  default: <statement(s)>;`
	`}`

In pseudocode, an example CASE statement is:

```
CASE OF Grade
  "A"      : OUTPUT "Top grade"
  "F", "U" : OUTPUT "Fail"
  "B".."E" : OUTPUT "Pass"
OTHERWISE OUTPUT "Invalid grade"
ENDCASE
```

Code examples

Python	`if Grade == "A":`
	`  print("Top grade")`
	`elif Grade == "F" or Grade == "U":`

```
  print("Fail")
elif Grade in ("B", "C", "D", "E"):
  print("Pass")
else:
  print("Invalid grade")
```

VB.NET

```
Select Case Grade
  Case "A"
    Console.WriteLine("Top grade")
  Case "F","U"
    Console.WriteLine("Fail")
  Case "B" To "E"
    Console.WriteLine("Pass")
  Case Else
    Console.WriteLine("Invalid grade")
End Select
```

Java

```
switch (grade)
{
  case 'A':
    System.out.println("Top Grade");
    break;
  case 'F': case 'U':
    System.out.println("Fail");
    break;
  case 'B': case 'C': case 'D': case 'E':
    System.out.println("Pass");
    break;
  default:
    System.out.println("Invalid grade");
}
```

TASK 14.05

The problem to be solved: the user enters the number of the month and year. The output is the number of days in that month. The program has to check if the year is a leap year for February.

The pseudocode solution is:

```
INPUT MonthNumber
INPUT Year
Days ← 0
CASE OF MonthNumber
  CASE 1,3,5,7,8,10,12 : Days ← 31
  CASE 4,6,9,11 : Days ← 30
  CASE 2 : Days ← 28
    If Year MOD 400 = 0
      THEN // it is a leap year
        Days ← 29
    ENDIF
    IF (Year MOD 4 = 0) AND (Year MOD 100 > 0)
      THEN // it is a leap year
        Days ← 29
    ENDIF
  OTHERWISE OUTPUT "Invalid month number"
ENDCASE
OUTPUT Days
```

Write program code to implement the pseudocode above.

14.06 Iteration

Count-controlled (FOR) loops

In pseudocode, a count-controlled loop is written as:

```
FOR <control variable> ← s TO e STEP i // STEP is optional
    <statement(s)>
NEXT <control variable>
```

The control variable starts with value s, increments by value i each time round the loop and finishes when the control variable reaches the value e.

Syntax definitions

Python	`for <control variable> in range(s, e, i):`
	`    <statement(s)>`
		The values s, e and i must be of type integer.
		The loop finishes when the control variable is just below e.
		The values for s and i can be omitted and they default to 0 and 1, respectively.

VB.NET	`For <control variable> = s To e Step i`
	`    <statement(s)>`
	`Next`
		The values s, e and i can be of type integer or float.

Java	`for (int i = s; i < e; i++)`
	`    <statement>;`
		Where i is the control variable.

In pseudcode, examples are:

```
FOR x ← 1 TO 5
    OUTPUT x
NEXT x
FOR x = 2 TO 14 STEP 3
    OUTPUT x
NEXT x
FOR x = 5 TO 1 STEP -1
    OUTPUT x
NEXT x
```

Code examples

Python	`for x in range(5):`
	`    print(x, end=' ')`
		The start value of x is 0 and it increases by 1 on each iteration.
		Output: `0 1 2 3 4`

	`for x in range(2, 14, 3):`
	`    print(x, end=' ')`
		Output: `2 5 8 11`

	`for x in range(5, 1, -1):`
	`    print(x, end=' ')`
		The start value of x is 5 and it decreases by 1 on each iteration.
		Output: `5 4 3 2`

	`for x in ["a", "b", "c"]:`
	`    print(x, end='')`
		The control variable takes the value of each of the group elements in turn.
		Output: `abc`

VB.NET	`For x = 1 To 5`
	`    Console.Write(x)`
	`Next`
		Output: `1 2 3 4 5`

	`For x = 2 To 14 Step 3`	Output: `2 5 8 11 14`

	`    Console.Write(x)`
	`Next`	

	`For x = 5 To 1 Step -1`
	`    Console.Write(x)`
	`Next`
		Output: `5 4 3 2 1`

	`For x = 1 To 2.5 Step 0.5`
	`    Console.WriteLine(x)`
	`Next`
		`Output:`
		`1`
		`1.5`
		`2`
		`2.5`

	`For Each x In {"a", "b", "c"}`
	`    Console.Write(x)`
	`Next`
		The control variable takes the value of each of the group elements in turn.
		Output: `abc`

Java	`for (int x = 1; x < 6; x++)`
	`{`
	`    System.out.print(x);`
	`}`
		Output: `12345`

	`for (int x = 2; x < 15; x = x + 3)`
	`{`
	`    System.out.print(x + "  ");`
	`}`
		Output: `2  5  8  11  14`

	`for (int x = 5; x > 0; x--)`
	`{`
	`    System.out.print(x + "  ");`
	`}`
		Output: `5  4  3  2  1`

	`for (double x = 1; x < 3; x = x + 0.5)`
	`{`
	`    System.out.print(x + "  ");`
	`}`
		Output: `1.0  1.5  2.0  2.5`

	`char[] letter = {'a', 'b', 'c'};`
	`for (char x : letter )`
	`{`
	`   System.out.print(x);`
	`}`
		The control variable takes the value of each of the group elements in turn.
		Output: `abc`

TASK 14.06

1 Write program code to implement the pseudocode from Worked Example 12.05 in Chapter 12.

2 Write program code to implement the pseudocode from Worked Example 12.08 in Chapter 12.

3 Write program code to implement the pseudocode from Worked Example 12.09 in Chapter 12.

Post-condition loops

A post-condition loop, as the name suggests, executes the statements within the loop at least once. When the condition is encountered, it is evaluated. As long as the condition evaluates to False, the statements within the loop are executed again. When the condition evaluates to True, execution will go to the next statement after the loop.

When coding a post-condition loop, you must ensure that there is a statement within the loop that will at some point change the end condition to True. Otherwise the loop will execute forever.

In pseudocode, the post-condition loop is written as:

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

Syntax definitions

<table>
<tr><td><b>Python</b></td><td>Post-condition loops are not available in Python. Use a pre-condition loop instead.</td></tr>
<tr><td><b>VB.NET</b></td><td><pre>
Do
    <statement(s)>
Loop Until <condition>
</pre></td></tr>
<tr><td><b>Java</b></td><td><pre>
do
{
    <statement(s)>
} while <condition>;
</pre></td></tr>
</table>

Pseudocode example:

```
REPEAT
    INPUT "Enter Y or N: " Answer
UNTIL Answer = "Y"
```

Code examples

<table>
<tr><td><b>VB.NET</b></td><td><pre>
Do
   Console.Write("Enter Y or N: ")
   Answer = Console.ReadLine()
Loop Until Answer = "Y"
</pre></td></tr>
<tr><td><b>Java</b></td><td><pre>
do
{
   System.out.print("Enter Y or N: ");
   answer = console.next();
} while (!(answer.equals("Y")));
</pre></td></tr>
</table>

TASK 14.07

1. Write program code to implement the pseudocode from Worked Example 12.04 in Chapter 12.
2. Write program code to implement the first algorithm from Worked Example 12.06 in Chapter 12.

Pre-condition loops

Pre-condition loops, as the name suggests, evaluate the condition before the statements within the loop are executed. Pre-condition loops will execute the statements within the loop as long as the condition evaluates to True. When the condition evaluates to False, execution will go to the next statement after the loop. Note that any variable used in the condition must not be undefined when the loop structure is first encountered.

When coding a pre-condition loop, you must ensure that there is a statement within the loop that will at some point change the value of the controlling condition. Otherwise the loop will execute forever.

In pseudocode the pre-condition loop is written as:

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

Syntax definitions

<table>
<tr><td><b>Python</b></td><td><pre>
while <condition>:
    <statement(s)>
</pre></td><td>Note that statements within the loop must be indented by a set number of spaces. The first statement after the</td></tr>
</table>

		loop must be indented less.

VB.NET	`Do While <condition>`
	`    <statement(s)>`
	`Loop`
	`Do Until <condition>`
	`    <statement(s)>`
	`Loop`
		Note the keyword `Loop` indicates the end of the loop.
		VB.NET also has a pre-condition `Until` loop. This will execute the statements within the loop as long as the condition evaluates to False. If the condition evaluates to True when the loop is first encountered, the statements within the loop are not executed at all.

Java	`while (<condition>)`
	`{`
	`    <statement(s)>;`
	`}`	

Pseudocode example,

```
Answer ← ""
WHILE Answer <> "Y" DO
    INPUT "Enter Y or N: " Answer
ENDWHILE
```

Code examples

Python	`Answer = ''`
	`while Answer != 'Y':`
	`   Answer = input("Enter Y or N: ")`

VB.NET	`Dim Answer As String = ""`
	`Do While Answer <> "Y"`
	`   Console.Write("Enter Y or N: ")`
	`   Answer = Console.ReadLine()`
	`Loop`
	`Answer = ""`
	`Do Until Answer = "Y"`
	`   Console.Write("Enter Y or N: ")`
	`   Answer = Console.ReadLine()`
	`Loop`

Java	`String answer = "";`
	`while(answer.equals("Y") == false)`
	`{`
	`   System.out.print("Enter Y or N: ");`
	`   answer = console.next();`
	`}`

TASK 14.08

Write program code to implement the second algorithm from Worked Example 12.06 in Chapter 12.

Which loop structure to use?

If you know how many times around the loop you need to go when the program execution gets to the loop statements, use a count-controlled loop. If the termination of the loop depends on some condition determined by what happens within the loop, then use a conditional loop. A pre-condition loop has the added benefit that the loop may not be entered at all, if the condition does not require it.

TIP

Computer Scientists like efficient code. Choosing the most suitable types of selection statements and loop structures is an important step along the way to design efficient code.

14.07 Built-in functions

Programming environments provide many built-in functions. Some of them are always available to use; some need to be imported from specialist module libraries.

Discussion Point:

Investigate your own programming environment and research other library routines.

String manipulation functions

Table 14.06 contains some useful functions for manipulating strings.

Description	Pseudocode	Python	VB.NET	Java
Access a single character using its position P in a string `ThisString`	`ThisString[P]` Counts from 1	`ThisString[P]` Counts from 0	`ThisString(P)` Counts from 0	`ThisString.charAt(P)` Counts from 0
Returns the character whose ASCII value is i	`CHR(i : INTEGER) RETURNS CHAR`	`chr(i)`	`Chr(i)`	`(char) i;`
Returns the ASCII value of character ch	`ASC(ch) RETURNS INTEGER`	`ord(ch)`	`Asc(ch)`	`(int) ch;`
Returns the integer value representing the length of string S	`LENGTH(S : STRING) RETURNS INTEGER`	`len(S)`	`len(S)`	`S.length();`
Returns leftmost L characters from S	`LEFT(S : STRING, L : INTEGER) RETURNS STRING`	`S[0:L]` See the next section, on slicing	`Left(S, L)`	`S.subString (0, L)`
Returns rightmost L characters from S	`RIGHT(S: STRING, L : INTEGER) RETURNS STRING`	`S[-L:]` See the next section, on slicing	`Right(S, L)`	`S.subString (S.length() – L)`
Returns a string of length L starting at position P from S	`MID(S : STRING, P : INTEGER, L : INTEGER) RETURNS STRING`	`S[P : P + L]` See the next section, on slicing	`mid(S, P, L)`	`S.subString(P, P + L)`
Returns the character value representing the lower case equivalent of Ch	`LCASE(Ch : CHAR) RETURNS CHAR`	`Ch.lower()`	`LCase(Ch)`	`Character.toLowerCase(ch)`
Returns the character value representing the upper case equivalent of Ch	`UCASE(Ch : CHAR) RETURNS CHAR`	`Ch.upper()`	`UCase(Ch)`	`Character.toUpperCase(ch)`
Returns a string formed by converting all alphabetic characters of S to upper case	`TO_UPPER(S : STRING) RETURNS STRING`	`S.upper()`	`S.ToUpper`	`S.toUpperCase()`
Returns a string formed by converting all alphabetic characters of S to lower	`TO_LOWER(S : STRING) RETURNS STRING`	`S.lower()`	`S.ToLower`	`S.toLowerCase()`

case				

Concatenate (join) two strings	`S1 & S2`	`s = S1 + S2`	`s = S1 + S2`
			`s = S1 & S2`
				`s = S1 + S2;`

Table 14.06 Some useful string manipulation functions

Slicing in Python

In Python a subsequence of any sequence type (e.g. lists and strings) can be created using ‘slicing’.

For example, to get a substring of length L from position P in string S we write S[P : P + L].

Figure 13.05 shows a representation of `ThisString`. If we want to return a slice of length 3 starting at position 3, we use `ThisString[3 : 6]` to give ‘DEF’. Position is counted from 0 and the position at the upper bound of the slice is not included in the substring.

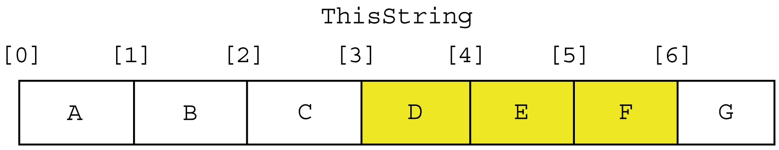


Figure 14.05 A representation of `ThisString`

If you imagine the numbering of each element to start at the left-hand end (as shown in Figure 14.05), then it is easier to see how the left element (the lower bound) is included, but the right element (the upper bound) is excluded. Table 14.07 shows some other useful slices in Python.

Expression	Result	Explanation
`ThisString[2:]`	`CDEFG`	If you do not state the upper bound, the slice includes all characters to the end of the string.
`ThisString[:2]`	`AB`	If you do not state the lower bound, the slice includes all characters from the beginning of the string.
`ThisString[-2:]`	`FG`	A negative lower bound means that it takes the slice starting from the end of the string.
`ThisString[:-2]`	`ABCDE`	A negative upper bound means that it terminates the string at that position.

Table 14.07 Some useful slices in Python

Truncating numbers

Instead of rounding, sometimes we just want the whole number part of a real number.

This is known as ‘truncation’.

Pseudocode	`INT(x : REAL) RETURNS INTEGER`	Returns the integer part of `x`.
Python	`int(x)`	If `x` is a floating-point number, the conversion truncates towards zero.
VB.NET	`Math.Truncate(x)`	The whole number part of the real number `x` is returned.
Java	`(int) x;`	Casts the number as an integer.

Converting a string to a number

Sometimes a whole number may be held as a string. To use such a number in a calculation, we first need to convert it to an integer. For example, these functions return the integer value 5 from the string `"5"`:

Python	`int(S)`
VB.NET	`CInt(S)`
Java	`Integer.valueOf(S)`

Sometimes a number with a decimal point may be held as a string. To use such a number in a calculation, we first need to convert it to a real (float). For example, these functions return the real number 75.43 from the string `"75.43"`:

Pseudocode	`STRING_TO_NUM(x : STRING) RETURNS REAL`	Returns a numeric representation of a string.
Python	`float(x)`	The returned value is a floating-point number.
VB.NET	`CDbl(x)`	The returned value is of type double.
Java	`Float.valueOf(x)`	The returned value is a floating-point number.

Random number generator

Random numbers are often required for simulations. Most programming languages have various random number generators available. As the random numbers are generated through a program, they are referred to as 'pseudo-random' numbers. A selection of the most useful random number generators are shown in the following code examples.

Code examples

Python	`# in the random library:`
	`randint(1, 6)`
		This code produces a random number between 1 and 6 inclusive.

VB.NET	`Dim RandomNumber As New Random`
	`Dim x As Integer`
	`x = RandomNumber.Next(1, 6)`
		You have to set up a RandomNumber object (see Chapter 27). This code generates an integer between 1 (inclusive) and 6 (exclusive).

Java	`import java.util.Random;`
	`Random randomNumber = new Random();`
	`int x = randomNumber.nextInt(6) + 1;`
		You have to set up a RandomNumber object (see Chapter 27). This code generates an integer between 1 (inclusive) and 6 (inclusive).

TASK 14.09

1 Write program code to generate 20 random numbers in the range 1 to 10 inclusive.

2 Write program code to implement the pseudocode using a pre-condition loop from Worked Example 12.07 in Chapter 12.

Discussion Point:

What other useful functions can you find? Which module libraries have you searched?

14.08 Procedures

In Chapter 12 (Section 12.09), we used procedures as a means of giving a group of statements a name. When we want to program a procedure we need to define it before the main program. We call it in the main program when we want the statements in the procedure body to be executed.

In pseudocode, a procedure definition is written as:

```
PROCEDURE <procedureIdentifier> // this is the procedure header
    <statement(s)>   // these statements are the procedure body
ENDPROCEDURE
```

This procedure is called using the pseudocode statement:

```
   CALL <procedureIdentifier>
```

Syntax definitions

Python	`def <identifier>():`
	`    <statement(s)>`

VB.NET	`Sub <identifier>()`
	`    <statement(s)>`
	`End Sub`

Java	`void <identifier>()`
	`{`
	`    <statement(s)>;`
	`}`

When programming a procedure, note where the definition is written and how the procedure is called from the main program.

Here is an example pseudocode procedure definition:

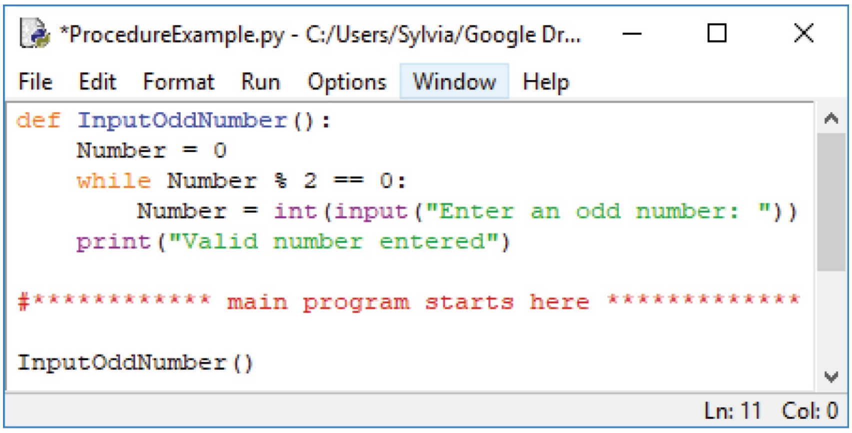
```
PROCEDURE InputOddNumber
    REPEAT
        INPUT "Enter an odd number: " Number
    UNTIL Number MOD 2 = 1
    OUTPUT "Valid number entered"
ENDPROCEDURE
```

This procedure is called using the CALL statement:

```
CALL InputOddNumber
```

Code examples

Python


*ProcedureExample.py - C:/Users/Sylvia/Google Dr...
File Edit Format Run Options Window Help
Ln: 11 Col: 0

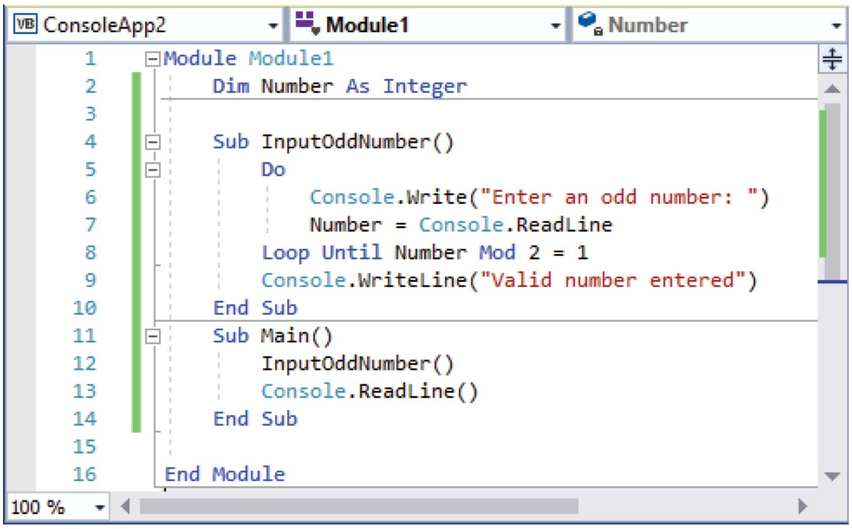
```
def InputOddNumber():
    Number = 0
    while Number % 2 == 0:
        Number = int(input("Enter an odd number: "))
    print("Valid number entered")

#************* main program starts here *************

InputOddNumber()
```

Figure 14.06 The Python editor with a procedure

The Python editor colour-codes the different parts of a statement. This helps when you are typing your own code. The indentation shows which statements are part of the loop.

The built-in function `input` returns a string, which must be converted to an integer before it can be used as a number.

VB.NET



```
Module Module1
    Dim Number As Integer

    Sub InputOddNumber()
        Do
            Console.Write("Enter an odd number: ")
            Number = Console.ReadLine
        Loop Until Number Mod 2 = 1
        Console.WriteLine("Valid number entered")
    End Sub
    Sub Main()
        InputOddNumber()
        Console.ReadLine()
    End Sub

End Module
```

Figure 14.07 The Visual Basic Express editor with a procedure

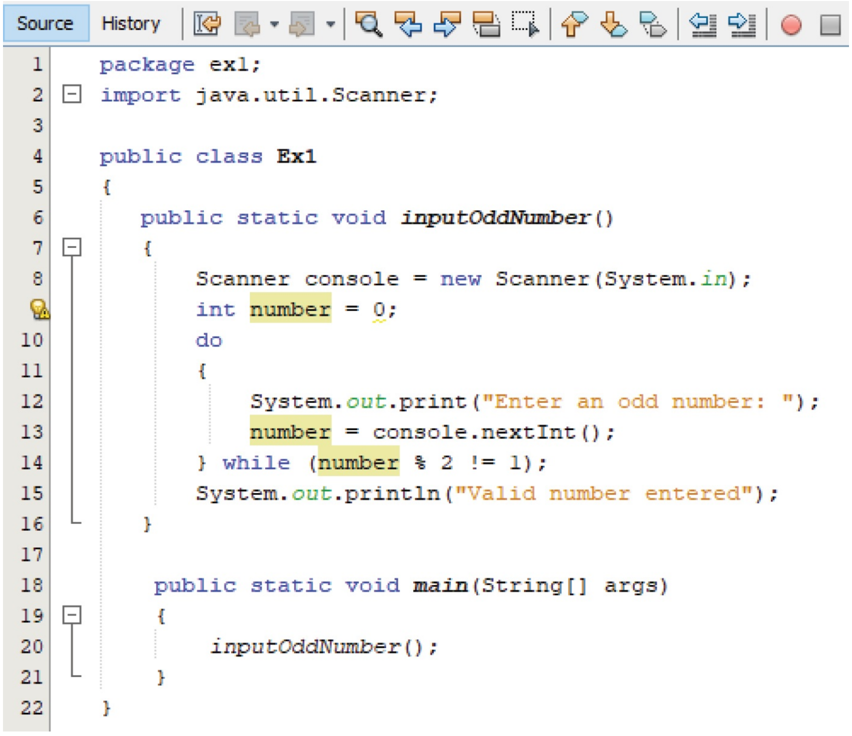
The Visual Basic Express editor colour-codes different parts of the statement, so it is easy to see if syntax errors are made. The editor also auto-indents and capitalises keywords.

Variables need to be declared before they are used. The editor will follow the capitalisation of the variable declaration when you type an identifier without following your original capitalisation.

The editor is predictive: pop-up lists will show when you type the first part of a statement.

When you execute the Main program, `Console.ReadLine()` keeps the run-time window open.

Java



```
package ex1;
import java.util.Scanner;

public class Ex1
{
    public static void inputOddNumber()
    {
        Scanner console = new Scanner(System.in);
        int number = 0;
        do
        {
            System.out.print("Enter an odd number: ");
            number = console.nextInt();
        } while (number % 2 != 1);
        System.out.println("Valid number entered");
    }

     public static void main(String[] args)
     {
         inputOddNumber();
     }
}
```

Figure 14.08 The NetBeans editor with a void method

The editor automatically colour codes keyword and strings.

The procedure body is enclosed within braces { and }.

The editor is predictive: pop-up lists will show when you type the first part of a statement.

Variables need to be declared before they are used.

TASK 14.10

Write program code to implement the pseudocode from Worked Example 12.11 in Chapter 12.

14.09 Functions

In Section 14.07 we used built-in functions. These are useful subroutines written by other programmers and made available in module libraries. The most-used ones are usually in the system library, so are available without you having to import them.

You can write your own functions. Any function you have written can be used in another program if you build up your own module library.

A function is used as part of an expression. When program execution gets to the statement that includes a function call as part of the expression, the function is executed. The **return value** from this function call is then used in the expression.

When writing your own function, ensure you always return a value as part of the statements that make up the function (the function body). You can have more than one RETURN statement if there are different paths through the function body.

In pseudocode, a function definition is written as:

```
FUNCTION <functionIdentifier> RETURNS <dataType> // function header
    <statement(s)> // function body
    RETURN <value>
ENDFUNCTION
```

Syntax definitions

Python	`def <functionIdentifier>():`
	`    <statement(s)>`
	`    return <value>`

VB.NET	`Function <functionIdentifier>() As <dataType>`
	`    <statement(s)>`
	`    <functionIdentifier> = <value> 'Return <value>`
	`End Function`

Java	`<data type> <functionIdentifier>()`
	`{`
	`    <statement(s)>;`
	`    return <value>;`
	`}`

When programming a function, the definition is written in the same place as a procedure. The function is called from within an expression in the main program, or in a procedure.

Different programming languages use different terminology for their subroutines, as listed in Table 14.08.

Pseudocode	**PROCEDURE**	**FUNCTION**
Python	void function	fruitful function
VB	Subroutine	Function
Java	void method	method

Table 14.08 Programming language terminology for subroutines

Void means ‘nothing’. Both Python and Java use this term to show that their procedure-type subroutine does not return a value. Python refers to both types of subroutines as functions. The fruitful function returns one or more values.

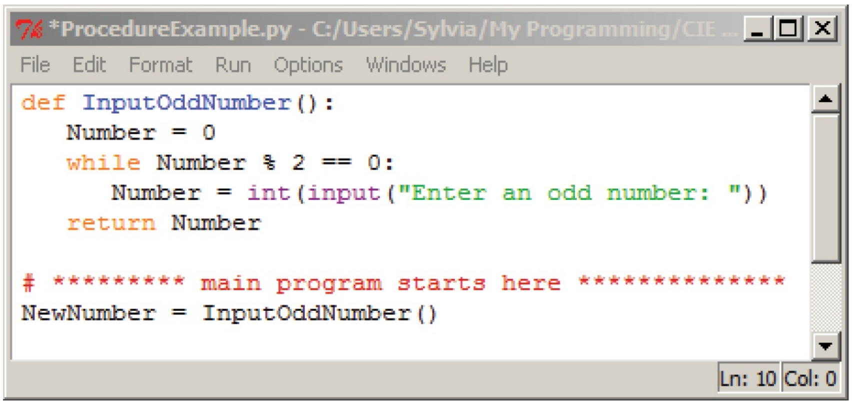
We can write the example procedure from Section 14.09 as a function. In pseudocode, this is:

```
FUNCTION InputOddNumber RETURNS INTEGER
    REPEAT
        INPUT "Enter an odd number: " Number
    UNTIL Number MOD 2 = 1
    OUTPUT "Valid number entered"
    RETURN Number
```

```
ENDFUNCTION
```

Code examples

Python


*ProcedureExample.py - C:/Users/Sylvia/My Programming/CIE ...
File Edit Format Run Options Windows Help
Ln: 10 Col: 0

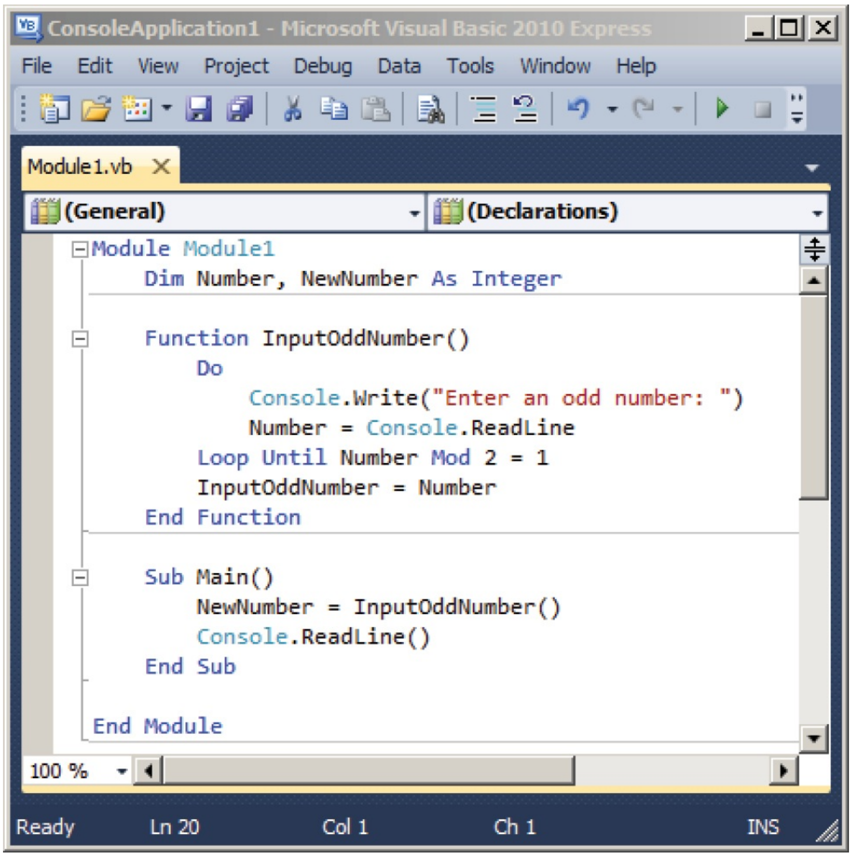
```python
def InputOddNumber():
    Number = 0
    while Number % 2 == 0:
        Number = int(input("Enter an odd number: "))
    return Number

# ********* main program starts here **************
NewNumber = InputOddNumber()
```

Figure 14.09 The Python editor with a function and local variable

The variable `Number` in Figure 14.09 is not accessible in the main program. Python’s variables are local unless declared to be global.

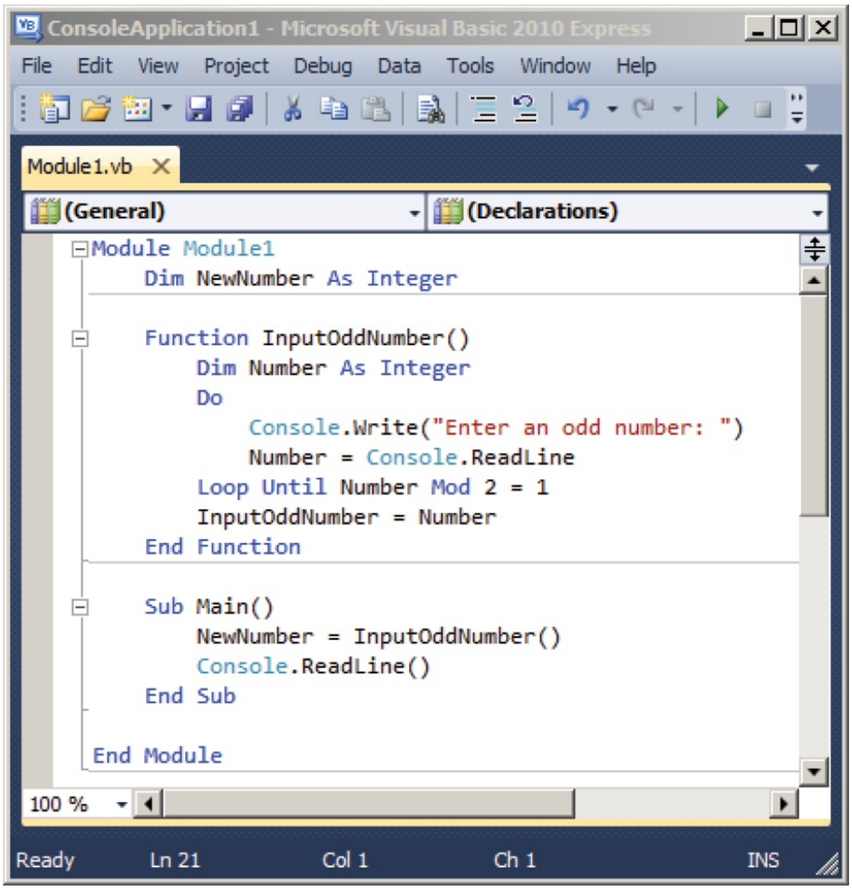
VB.NET

(a)


ConsoleApplication1 - Microsoft Visual Basic 2010 Express
File Edit View Project Debug Data Tools Window Help
Module1.vb
(General)
(Declarations)
100 %
Ready Ln 20 Col 1 Ch 1 INS

```vbnet
Module Module1
    Dim Number, NewNumber As Integer

    Function InputOddNumber()
        Do
            Console.Write("Enter an odd number: ")
            Number = Console.ReadLine
        Loop Until Number Mod 2 = 1
        InputOddNumber = Number
    End Function

    Sub Main()
        NewNumber = InputOddNumber()
        Console.ReadLine()
    End Sub

End Module
```

(b)

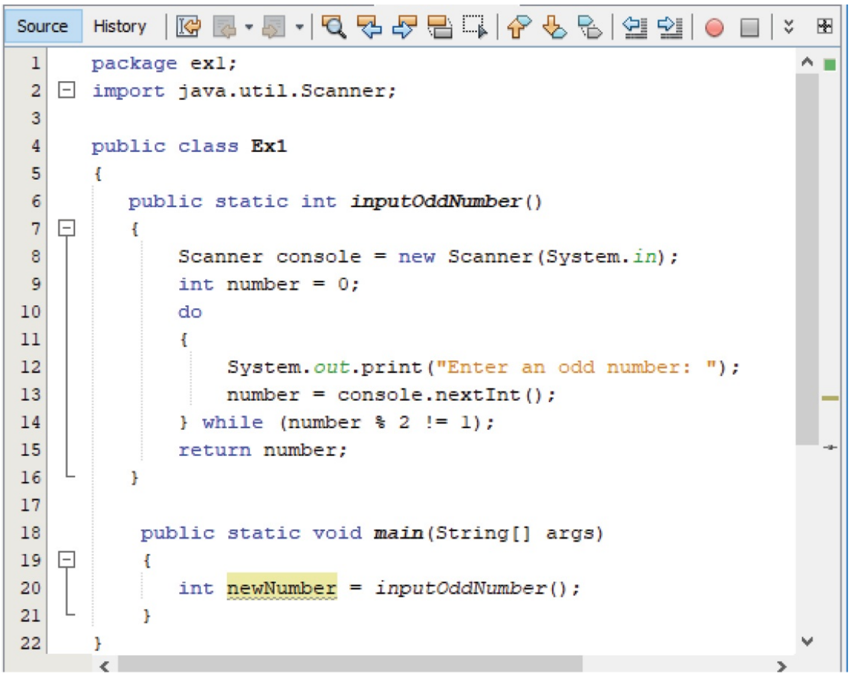

ConsoleApplication1 - Microsoft Visual Basic 2010 Express
File Edit View Project Debug Data Tools Window Help
Module1.vb
(General)
(Declarations)
100 %
Ready Ln 21 Col 1 Ch 1 INS

```
Module Module1
    Dim NewNumber As Integer

    Function InputOddNumber()
        Dim Number As Integer
        Do
            Console.Write("Enter an odd number: ")
            Number = Console.ReadLine
        Loop Until Number Mod 2 = 1
        InputOddNumber = Number
    End Function

    Sub Main()
        NewNumber = InputOddNumber()
        Console.ReadLine()
    End Sub

End Module
```

Figure 14.10 The VB.NET editor with (a) global variables and (b) a local variable

The variable `Number` in Figure 14.10(a) is declared as a global variable at the start of the module. This is not good programming practice.	In Figure 14.10(b), the variable `Number` is declared as a local variable within the function.

Java



```
package ex1;
import java.util.Scanner;

public class Ex1
{
    public static int inputOddNumber()
    {
        Scanner console = new Scanner(System.in);
        int number = 0;
        do
        {
            System.out.print("Enter an odd number: ");
            number = console.nextInt();
        } while (number % 2 != 1);
        return number;
    }

     public static void main(String[] args)
     {
        int newNumber = inputOddNumber();
     }
}
```

Figure 14.11 The NetBeans editor with a function and local variable

The variable `number` in Figure 14.11 is not accessible in the main program.

A global variable is available in any part of the program code. It is good programming practice to declare a variable that is only used within a subroutine as a local variable.

In Python, every variable is local, unless it is overridden with a global declaration. In VB.NET you need to write the declaration statement for a local variable within the subroutine. Java does not support global variables. However, static variables declared in a class are accessible throughout the class.

TASK 14.11

Write program code to implement the pseudocode from Worked Example 13.05 in Chapter 13. Which variables are global and which are local?

14.10 Passing parameters to subroutines

When a subroutine requires one or more values from the main program, we supply these as **arguments** to the subroutine at call time. This is how we use built-in functions. We don’t need to know the identifiers used within the function when we call a built-in function.

When we define a subroutine that requires values to be passed to the subroutine body, we use a parameter list in the subroutine header. When the subroutine is called, we supply the arguments in brackets. The arguments supplied are assigned to the corresponding **parameter** of the subroutine (note the order of the parameters in the parameter list must be the same as the order in the list of arguments). This is known as the **subroutine interface**.

14.11 Passing parameters to functions

The **function header** is written in pseudocode as:

```
FUNCTION <functionIdentifier> (<parameterList>) RETURNS <dataType>
```

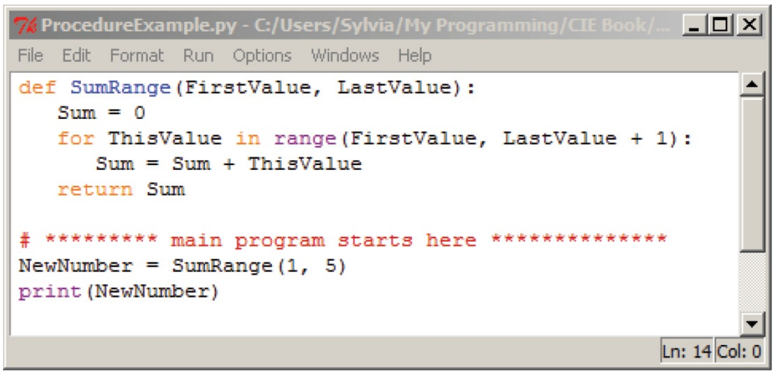
where `<parameterList>` is a list of identifiers and their data types, separated by commas.

Here is an example pseudocode function definition that uses parameters:

```
FUNCTION SumRange(FirstValue : INTEGER, LastValue : INTEGER) RETURNS INTEGER
    DECLARE Sum, ThisValue : INTEGER
    Sum ← 0
    FOR ThisValue ← FirstValue TO LastValue
        Sum ← Sum + ThisValue
    NEXT ThisValue
    RETURN Sum
ENDFUNCTION
```

Code examples

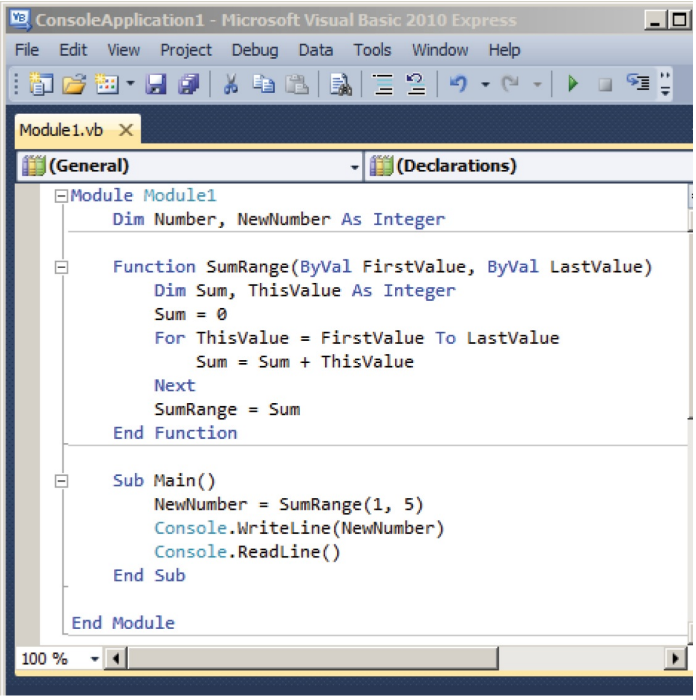
Python


ProcedureExample.py - C:/Users/Sylvia/My Programming/CIE Book/...
File Edit Format Run Options Windows Help
Ln: 14 Col: 0

```python
def SumRange(FirstValue, LastValue):
    Sum = 0
    for ThisValue in range(FirstValue, LastValue + 1):
        Sum = Sum + ThisValue
    return Sum

# ********* main program starts here **************
NewNumber = SumRange(1, 5)
print(NewNumber)
```

Figure 14.12 The `SumRange()` function in Python

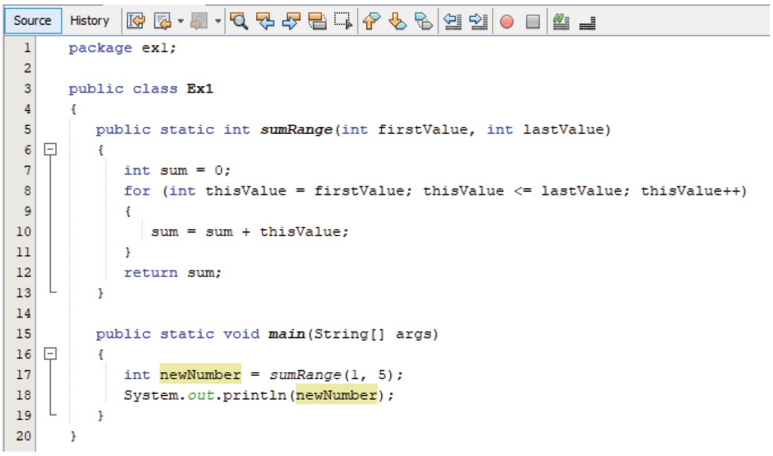
VB.NET


ConsoleApplication1 - Microsoft Visual Basic 2010 Express
File Edit View Project Debug Data Tools Window Help
Module1.vb
(General)
(Declarations)
100 %

```vbnet
Module Module1
    Dim Number, NewNumber As Integer

    Function SumRange(ByVal FirstValue, ByVal LastValue)
        Dim Sum, ThisValue As Integer
        Sum = 0
        For ThisValue = FirstValue To LastValue
            Sum = Sum + ThisValue
        Next
        SumRange = Sum
    End Function

    Sub Main()
        NewNumber = SumRange(1, 5)
        Console.WriteLine(NewNumber)
        Console.ReadLine()
    End Sub

End Module
```

Figure 14.13 The `SumRange()` function in VB.NET

Java



```
package ex1;

public class Ex1
{
    public static int sumRange(int firstValue, int lastValue)
    {
        int sum = 0;
        for (int thisValue = firstValue; thisValue <= lastValue; thisValue++)
        {
            sum = sum + thisValue;
        }
        return sum;
    }

    public static void main(String[] args)
    {
        int newNumber = sumRange(1, 5);
        System.out.println(newNumber);
    }
}
```

Figure 14.14 The `SumRange()` function in Java

TASK 14.12

Write a function to implement the following pseudocode:

```
FUNCTION Factorial (Number : INTEGER) RETURNS INTEGER
    DECLARE Product : INTEGER
    Product ← 1
    FOR n ← 2 TO Number
        Product ← Product * n
    NEXT n
    RETURN Product
ENDFUNCTION
```

14.12 Passing parameters to procedures

If a parameter is passed **by value**, at call time the argument can be an actual value (as we showed in the code examples in Section 14.11). If the argument is a variable, then a copy of the current value of the variable is passed into the subroutine. The value of the variable in the calling program is not affected by what happens in the subroutine.

For procedures, a parameter can be passed **by reference**. At call time, the argument must be a variable. A pointer to the memory location (the address) of that variable is passed into the procedure. Any changes that are applied to the variable's contents will be effective outside the procedure in the calling program/module.

Note that neither of these methods of parameter passing applies to Python. In Python or Java, the method is called pass by object reference. This is basically an object-oriented way of passing parameters and is beyond the scope of this chapter (objects are dealt with in Chapter 27). The important point is to understand how to program in Python and Java to get the desired effect.

The full **procedure header** is written in pseudocode, in a very similar fashion to that for function headers, as:

```
PROCEDURE <ProcedureIdentifier> (<parameterList>)
```

The parameter list needs more information for a procedure definition. In pseudocode, a parameter in the list is represented in one of the following formats:

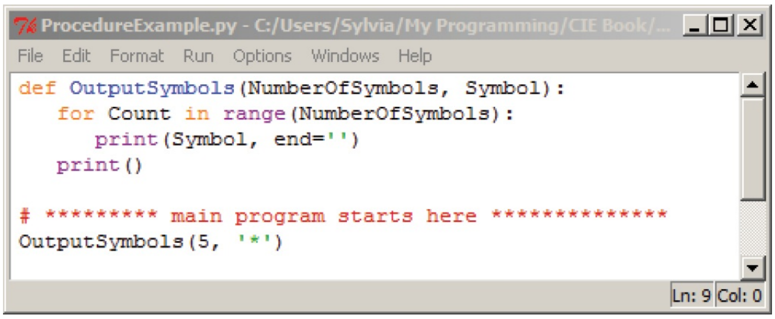
```
BYREF <identifier1> : <dataType>
BYVALUE <identifier2> : <dataType>
```

Passing parameters by value

The pseudocode for the pyramid example in Chapter 12 (Section 12.09) includes a procedure definition that uses two parameters passed by value. We can now make that explicit:

```
PROCEDURE OutputSymbols(BYVALUE NumberOfSymbols : INTEGER, Symbol : CHAR)
    DECLARE Count : INTEGER
    FOR Count ← 1 TO NumberOfSymbols
        OUTPUT Symbol // without moving to next line
    NEXT Count
    OUTPUT NewLine
ENDPROCEDURE
```

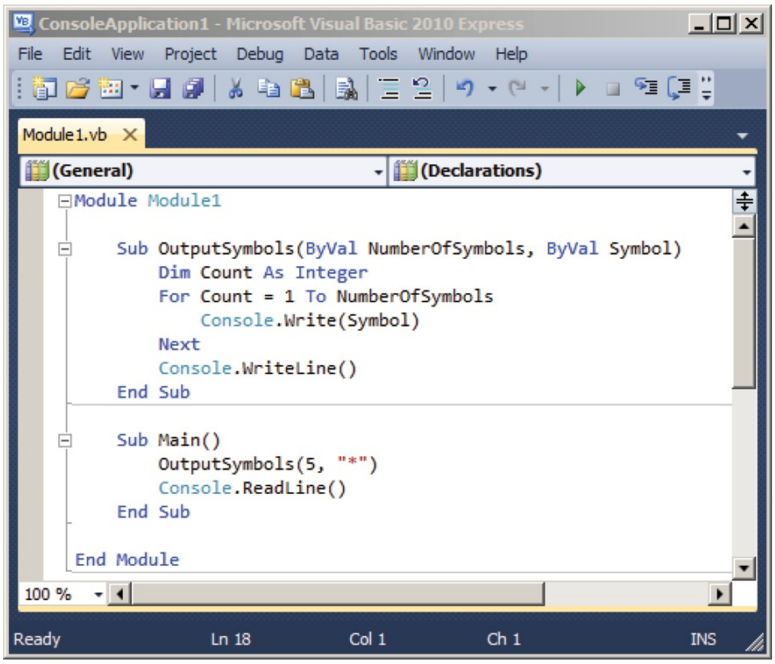
In Python (see Figure 14.15), all parameters behave like local variables and their effect is as though they are passed by value.


ProcedureExample.py - C:/Users/Sylvia/My Programming/CIE Book/...
File Edit Format Run Options Windows Help
Ln: 9 Col: 0

```
def OutputSymbols(NumberOfSymbols, Symbol):
   for Count in range(NumberOfSymbols):
      print(Symbol, end='')
   print()

# ********* main program starts here **************
OutputSymbols(5, '*')
```

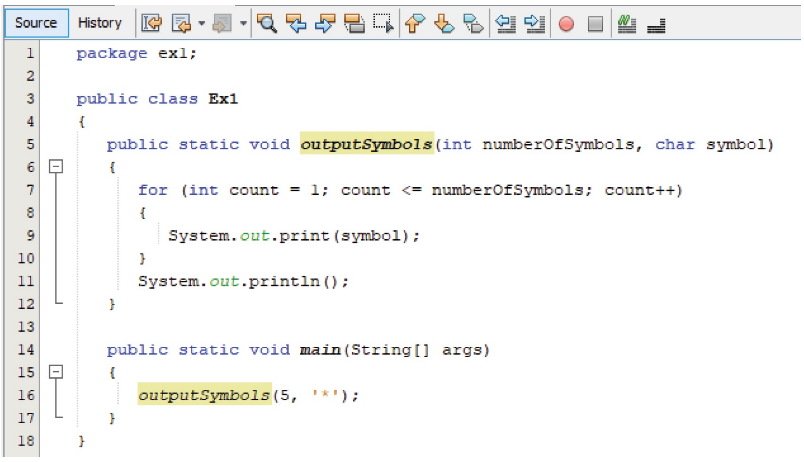
Figure 14.15 Parameters passed to a Python subroutine

In VB.NET (see Figure 14.16), parameters default to passing by value. The keyword ByVal is automatically inserted by the editor.


ConsoleApplication1 - Microsoft Visual Basic 2010 Express
File Edit View Project Debug Data Tools Window Help
Module1.vb
(General)
(Declarations)
100 %
Ready
Ln 18
Col 1
Ch 1
INS

```
Module Module1

    Sub OutputSymbols(ByVal NumberOfSymbols, ByVal Symbol)
        Dim Count As Integer
        For Count = 1 To NumberOfSymbols
            Console.Write(Symbol)
        Next
        Console.WriteLine()
    End Sub

    Sub Main()
        OutputSymbols(5, "*")
        Console.ReadLine()
    End Sub

End Module
```

Figure 14.16 Parameters passed by value to a VB.NET procedure



```
package ex1;

public class Ex1
{
    public static void outputSymbols(int numberOfSymbols, char symbol)
    {
        for (int count = 1; count <= numberOfSymbols; count++)
        {
            System.out.print(symbol);
        }
        System.out.println();
    }

    public static void main(String[] args)
    {
        outputSymbols(5, '*');
    }
}
```

Figure 14.17 Parameters passed by value to a Java procedure

In Java (see Figure 14.17), all parameters behave like local variables and their effect is as though they are passed by value.

Passing parameters by reference

When parameters are passed by reference, when the values inside the subroutine change, this affects the values of the variables in the calling program.

Consider the pseudocode procedure AdjustValuesForNextRow below.

The pseudocode for the pyramid example generated in Chapter 12 (Section 12.09) includes a procedure definition that uses two parameters passed by reference. We can now make that explicit:

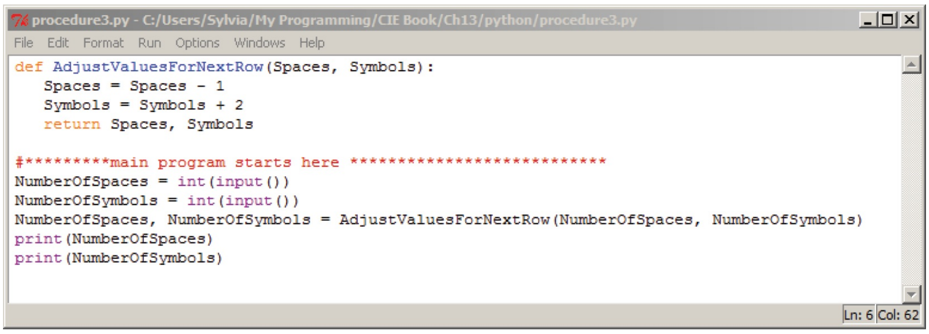
```
PROCEDURE AdjustValuesForNextRow(BYREF Spaces : INTEGER, Symbols : INTEGER)
    Spaces ← Spaces - 1
    Symbols ← Symbols + 2
ENDPROCEDURE
```

The pseudocode statement to call the procedure is:

```
CALL AdjustValuesForNextRow(NumberOfSpaces, NumberOfSymbols)
```

The values of the parameters `Spaces` and `Symbols` are changed within the procedure when this is called. The variables NumberOfSpaces and `NumberOfSymbols` in the program code after the call will store the updated values that were passed back from the procedure.

Python does not have a facility to pass parameters by reference. Instead the subroutine behaves as a function and returns multiple values (see Figure 14.18). Note the order of the variables as they receive these values in the main part of the program.

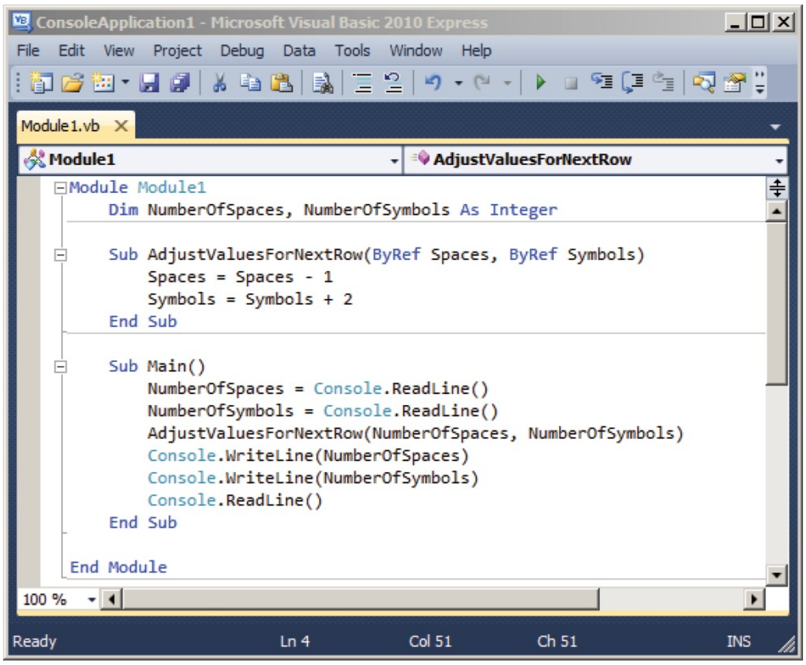

procedure3.py - C:/Users/Sylvia/My Programming/CIE Book/Ch13/python/procedure3.py
File Edit Format Run Options Windows Help
Ln: 6 Col: 62

```
def AdjustValuesForNextRow(Spaces, Symbols):
    Spaces = Spaces - 1
    Symbols = Symbols + 2
    return Spaces, Symbols

#*********main program starts here ***************************
NumberOfSpaces = int(input())
NumberOfSymbols = int(input())
NumberOfSpaces, NumberOfSymbols = AdjustValuesForNextRow(NumberOfSpaces, NumberOfSymbols)
print(NumberOfSpaces)
print(NumberOfSymbols)
```

Figure 14.18 Multiple values returned from a Python subroutine

This way of treating a multiple of values as a unit is called a ‘tuple’ (see Chapter 11, Section 11.02). You can find out more by reading the Python help files.

In VB.NET (see Figure 14.19), the ByRef keyword is placed in front of each parameter to be passed by reference.


ConsoleApplication1 - Microsoft Visual Basic 2010 Express
File Edit View Project Debug Data Tools Window Help
Module1.vb
Module1
AdjustValuesForNextRow
100 %
Ready
Ln 4
Col 51
Ch 51
INS

```
Module Module1
    Dim NumberOfSpaces, NumberOfSymbols As Integer

    Sub AdjustValuesForNextRow(ByRef Spaces, ByRef Symbols)
        Spaces = Spaces - 1
        Symbols = Symbols + 2
    End Sub

    Sub Main()
        NumberOfSpaces = Console.ReadLine()
        NumberOfSymbols = Console.ReadLine()
        AdjustValuesForNextRow(NumberOfSpaces, NumberOfSymbols)
        Console.WriteLine(NumberOfSpaces)
        Console.WriteLine(NumberOfSymbols)
        Console.ReadLine()
    End Sub

End Module
```

Figure 14.19 Parameters passed by reference to a VB.NET procedure

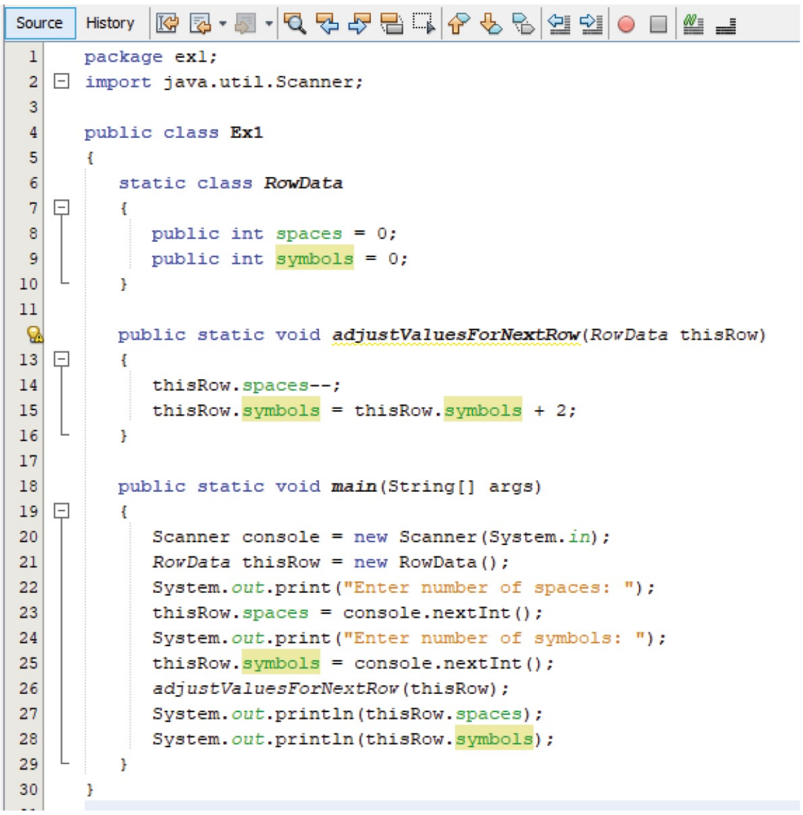
Java does not have a facility to pass simple variable parameters by reference. Only objects can be passed by reference. Arrays are objects in Java, so these are passed by reference.

TIP

If only one value needs to be returned, the subroutine can be written as a function. If more than one value needs to be returned, a work-around is to declare a class and return it as an object (Figure 14.20). If the values to be returned are of the same type, they can be grouped into an array and the array is passed as a reference parameter.

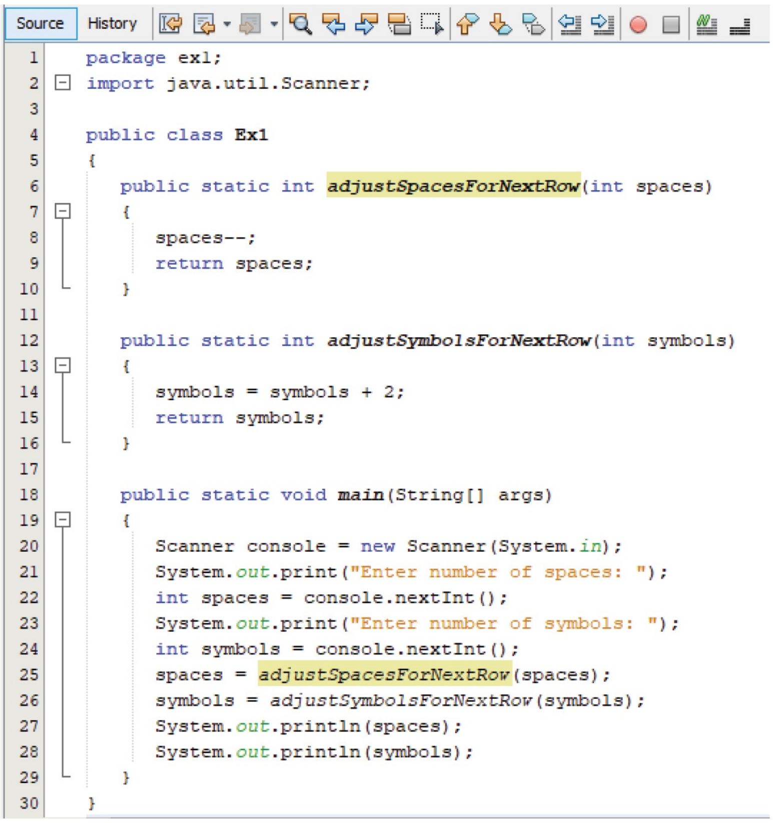
TIP

A preferable solution is to amend the algorithm and write several functions (Figure 14.21).



```java
package ex1;
import java.util.Scanner;

public class Ex1
{
    static class RowData
    {
        public int spaces = 0;
        public int symbols = 0;
    }

    public static void adjustValuesForNextRow(RowData thisRow)
    {
        thisRow.spaces--;
        thisRow.symbols = thisRow.symbols + 2;
    }

    public static void main(String[] args)
    {
        Scanner console = new Scanner(System.in);
        RowData thisRow = new RowData();
        System.out.print("Enter number of spaces: ");
        thisRow.spaces = console.nextInt();
        System.out.print("Enter number of symbols: ");
        thisRow.symbols = console.nextInt();
        adjustValuesForNextRow(thisRow);
        System.out.println(thisRow.spaces);
        System.out.println(thisRow.symbols);
    }
}
```

Figure 14.20 Multiple values returned as an object from a Java subroutine



```java
package ex1;
import java.util.Scanner;

public class Ex1
{
    public static int adjustSpacesForNextRow(int spaces)
    {
        spaces--;
        return spaces;
    }

    public static int adjustSymbolsForNextRow(int symbols)
    {
        symbols = symbols + 2;
        return symbols;
    }

    public static void main(String[] args)
    {
        Scanner console = new Scanner(System.in);
        System.out.print("Enter number of spaces: ");
        int spaces = console.nextInt();
        System.out.print("Enter number of symbols: ");
        int symbols = console.nextInt();
        spaces = adjustSpacesForNextRow(spaces);
        symbols = adjustSymbolsForNextRow(symbols);
        System.out.println(spaces);
        System.out.println(symbols);
    }
}
```

Figure 14.21 The algorithm changed into two subroutines for Java

14.13 Putting it all together

The programs in this section are full solutions to the pyramid-drawing program developed in Section 14.12.

The parameters of the subroutines have different identifiers from the variables in the main program. This is done deliberately, so that it is quite clear that the parameters and local variables within a subroutine are separate from those in the calling program or module. If a parameter is passed by reference to a procedure, the parameter identifier within the procedure references the same memory location as the variable identifier passed to the procedure as argument.

The pyramid-drawing program in Python, VB.NET and Java

Python

```
SPACE = ' ' # constant to give a space a name
def InputMaxNumberOfSymbols():
   Number = 0
   while Number % 2 == 0:
      print("How many symbols make the base? ")
      Number = int(input("Input an odd number: "))
   return Number

def SetValues():
   Symbol = input("What symbol do you want to use? ")
   MaxSymbols = InputMaxNumberOfSymbols()
   Spaces = (MaxSymbols + 1) // 2
   Symbols = 1
   return Symbol, MaxSymbols, Spaces, Symbols

def OutputChars(Number, Symbol):
   for Count in range (Number):
      print(Symbol, end='')

def AdjustValuesForNextRow(Spaces, Symbols):
   Spaces = Spaces – 1
   Symbols = Symbols + 2
   return Spaces, Symbols

def main():
   ThisSymbol, MaxNumberOfSymbols, NumberOfSpaces, NumberOfSymbols = SetValues()
   while NumberOfSymbols <= MaxNumberOfSymbols:
      OutputChars(NumberOfSpaces, SPACE)
      OutputChars(NumberOfSymbols, ThisSymbol)
      print()  # move to new line
      NumberOfSpaces, NumberOfSymbols = AdjustValuesForNextRow(NumberOfSpaces, NumberOfSymbols)
main()
```

VB.NET

```
Module Module1
   Const Space = " " 'constant to give a space a name
   Dim NumberOfSpaces, NumberOfSymbols As Integer
   Dim MaxNumberOfSymbols As Integer
   Dim ThisSymbol As Char

Sub InputMaxNumberOfSymbols(ByRef Number As Integer)
   Do
      Console.WriteLine("How many symbols make the base? ")
      Console.Write("Input an odd number: ")
      Number = Console.ReadLine()
   Loop Until (Number Mod 2 = 1)
End Sub

Sub SetValues(ByRef Symbol, ByRef MaxSymbols, ByRef Spaces, ByRef Symbols)
   Console.Write("What symbol do you want to use? ")
   Symbol = Console.ReadLine()
   InputMaxNumberOfSymbols(MaxSymbols)
   Spaces = (MaxSymbols + 1) \ 2
   Symbols = 1
End Sub
```

```
Sub OutputChars(ByVal Number, ByVal Symbol)
   Dim Count As Integer
   For Count = 1 To Number
      Console.Write(Symbol)
   Next
End Sub

Sub AdjustValuesForNextRow(ByRef Spaces, ByRef Symbols)
   Spaces = Spaces - 1
   Symbols = Symbols + 2
End Sub

Sub Main()
   SetValues(ThisSymbol, MaxNumberOfSymbols, NumberOfSpaces, NumberOfSymbols)
   Do
      OutputChars(NumberOfSpaces, Space)
      OutputChars(NumberOfSymbols, ThisSymbol)
      Console.WriteLine()'move to new line
      AdjustValuesForNextRow(NumberOfSpaces, NumberOfSymbols)
   Loop Until NumberOfSymbols > MaxNumberOfSymbols
   Console.ReadLine()
End Sub

End Module
```

Java

```
package ex1;
import java.util.Scanner;

public class Ex1
{
   static final char SPACE = ' ';

   public static char getSymbol()
   {
      Scanner console = new Scanner(System.in);
      System.out.print("What symbol do you want to use? ");
      String response = console.next();
      return response.charAt(0);
   }
   public static int inputMaxNumberOfSymbols()
   {
      Scanner console = new Scanner(System.in);
      int number = 0;
      while ((number % 2) == 0)
      {
         System.out.print("How many symbols make the base? ");
         number = console.nextInt();
      }
      return number;
   }

   public static void outputChars(int number, char symbol)
   {
      for (int count = 0; count < number; count++)
      {
         System.out.print(symbol);
      }
   }
   public static int adjustSpacesForNextRow(int spaces)
   {
      spaces--;
      return spaces;
   }
   public static int adjustSymbolsForNextRow(int symbols)
   {
      symbols = symbols + 2;
      return symbols;
   }
   public static void main(String[] args)
   {
      char thisSymbol = getSymbol();
      int maxNumberOfSymbols = inputMaxNumberOfSymbols();
      int numberOfSpaces = (maxNumberOfSymbols + 1)/ 2;
      int numberOfSymbols = 1;

      while (numberOfSymbols <= maxNumberOfSymbols)
      {
         outputChars(numberOfSpaces, SPACE);
         outputChars(numberOfSymbols, thisSymbol);
         System.out.println();
         numberOfSpaces = adjustSpacesForNextRow(numberOfSpaces);
         numberOfSymbols = adjustSymbolsForNextRow(numberOfSymbols);
      }
   }
}
```

Discussion Point:

Can you see how the two procedures OutputSpaces and OutputSymbols have been replaced by a single procedure OutputChars without changing the effect of the program?

14.14 Arrays

Creating 1D arrays

VB.NET, Python and Java number array elements from 0 (the lower bound).

In pseudocode, a 1D array declaration is written as:

```
DECLARE <arrayIdentifier> : ARRAY[<lowerBound>:<upperBound>] OF <dataType>
```

Syntax definitions

Python	In Python, there are no arrays. The equivalent data structure is called a list. A list is an ordered sequence of items that do not have to be of the same data type.
VB.NET	`Dim <arrayIdentifier>(<upperBound>) As <dataType>`

Java	`<datatype>[] <arrayIdentifier>;`
	`<arrayIdentifier> = new int[<upperbound>+1];`

Pseudocode example:

```
   DECLARE List1 : ARRAY[1:3]   OF STRING  // 3 elements in this list
   DECLARE List2 : ARRAY[0:5]   OF INTEGER // 6 elements in this list
   DECLARE List3 : ARRAY[1:100] OF INTEGER // 100 elements in this list
   DECLARE List4 : ARRAY[0:25] OF STRING  // 26 elements in this list
```

Code examples

Python	`List1 = []`
	`List1.append("Fred")`
	`List1.append("Jack")`
	`List1.append("Ali")`
		As there are no declarations, the only way to generate a list is to initialise one. You can append elements to an existing list.

	`List2 = [0, 0, 0, 0, 0, 0]`	You can enclose the elements in `[ ]`.
	`List3 = [0 for i in range(100)]`	You can use a loop.
	`AList = [""] * 26`	You can provide an initial value, multiplied by number of elements required.

VB.NET	`Dim List1 As String () = {"","",""}`
	`Dim List2(5) As Integer`
	`Dim List3(100) As Integer`
	`Dim AList(0 To 25) As String`
		You can initialise an array at declaration time (as with `List1`). Note that `List3` has 101 elements. You can use a range as an array dimension (as with `AList`) however the lower bound must be 0.

Java	`String[] list1 = {"","",""};`
	`int[] list2;`
	`list2 = new int[5];`
	`int[] list3;`
	`list3 = new int[100];`
	`String[] aList;`
	`aList = new String[25];`
		You can initialise an array at declaration time (as with `list1`).

Accessing 1D arrays

A specific element in an array is accessed using an index value. In pseudocode, this is written as:

```
   <arrayIdentifier>[x]
```

Pseudocode example:

```
   NList[25] = 0  // set 25th element to zero
   AList[3] = "D" // set 3rd element to letter D
```

Code examples

Python	`NList[24] = 0`
	`AList[3] = "D"`

VB.NET	`NList(25) = 0`
	`AList(3) = "D"`

Java	`nList[25] = 0;`
	`aList[3] = "D";`

In Python, you can print the whole contents of a list using `print(List)`. In VB.NET and Java, you need to use a loop to print one element of an array at a time.

TIP

When writing a solution using pseudocode, always use a loop to print the contents of an array.

TASK 14.13

1 Write program code to implement the pseudocode from Worked Example 13.01 in Chapter 13.

2 Write program code to implement the pseudocode from Worked Example 13.02 in Chapter 13.

3 Write program code to implement the improved algorithm from Worked Example 13.03 in Chapter 13.

Creating 2D arrays

In pseudocode, a 2D array declaration is written as:

```
DECLARE <identifier> : ARRAY[<lBound1>:<uBound1>,
<lBound2>:<uBound2>] OF <dataType>
```

Syntax definitions

Python	In Python, there are no arrays. The equivalent data structure is a list of lists.
VB.NET	`Dim <arrayIdentifier>(<uBound1, uBound2>) As <dataType>`

Java	`<dataType> <arrayIdentifier>;`
	`<arrayIdentifier> = new <datatype>[<uBound1>][<uBound2>];`

To declare a 2D array to represent a game board of six rows and seven columns, the pseudocode statement is:

```
DECLARE Board : ARRAY[1:6, 1:7] OF INTEGER
```

Code examples

Python	`Board = [[0, 0, 0, 0, 0, 0, 0],`
	`         [0, 0, 0, 0, 0, 0, 0],`
	`         [0, 0, 0, 0, 0, 0, 0],`
	`         [0, 0, 0, 0, 0, 0, 0],`
	`         [0, 0, 0, 0, 0, 0, 0],`
	`         [0, 0, 0, 0, 0, 0, 0]]`
	`Board = [[0 for i in range(7)]`
	`  for j in range(6)]`
	`Board = [[0] * 7] * 6`
		2D lists can be initialised in a similar way to 1D lists.
		Remember that elements are numbered from 0.
		These are alternative ways of initialising a 6 × 7 list. The rows are numbered 0 to 5 and the columns 0 to 6.
		The upper value of the range is not included.

VB.NET	`Dim Board(6, 7) As Integer`	Elements are numbered from 0 to the given number. This declaration has one row and one column too many. However, the algorithm may be such that it is easier to convert to program code if row 0 and column 0 are ignored.

Java	`int[][] board = {`
	`         {0, 0, 0, 0, 0, 0, 0},`
		2D arrays can be initialised in a similar way to 1D arrays.
		Remember that elements are numbered from 0.

```
         {0, 0, 0, 0, 0, 0, 0},
         {0, 0, 0, 0, 0, 0, 0},
         {0, 0, 0, 0, 0, 0, 0},
         {0, 0, 0, 0, 0, 0, 0},
         {0, 0, 0, 0, 0, 0, 0}
      }
int[][] board;
board = new int[6][7];
```

Accessing 2D arrays

A specific element in a table is accessed using an index pair. In pseudocode this is written as:

```
<arrayIdentifier>[x, y]
```

Pseudocode example:

```
  Board[3,4] ← 0 // sets the element in row 3 and column 4 to zero
```

The following code examples demonstrate how to access elements in each of the three languages.

Code examples

Python	`Board[2][3] = 0`	Elements are numbered from 0 in Python, so [3] gives access to the fourth element.
VB.NET	`Board(3, 4) = 0`	We are ignoring row 0 and column 0.
Java	`board[2][3] = 0;`	Elements are numbered from 0 in Java, so [3] gives access to the fourth element.

TASK 14.14

1. Write program code to implement the pseudocode from Worked Example 13.04 in Chapter 13; first initialise the table and then output its contents.
2. Write program code to implement the pseudocode from Worked Example 13.05 in Chapter 13.

14.15 Text files

Writing to a text file

The following pseudocode statements provide facilities for writing to a file:

```
OPENFILE <filename> FOR WRITE       // open the file for writing
WRITEFILE <filename>, <stringValue> // write a line of text to the file
CLOSEFILE <filename>                // close file
```

The following code examples demonstrate how to open, write to and close a file called `SampleFile.TXT` in each of the three languages. If the file already exists, it is overwritten as soon as the file handle is assigned by the 'open file' command.

Code examples

Python	`FileHandle = open("SampleFile.TXT", "w")`
	`FileHandle.write(LineOfText)`
	`FileHandle.close()`
		You specify the filename and mode ('`w`' for write) when you call the `open` function. The line of text to be written to the file must contain the newline character "\n" to move to the next line of the text file.

VB.NET	`Dim FileHandle As IO.StreamWriter`
	`Dim LineOfText As String`
	`FileHandle = New IO.StreamWriter("SampleFile.TXT")`
	`FileHandle.WriteLine(LineOfText)`
	`FileHandle.Close()`
		The file is accessed through an object (see Chapter 27) called a `StreamWriter`.

	`Alternative method:`
	`Dim LineOfText As String`
	`Dim Channel As Integer = 1`
	`FileSystem.FileOpen(Channel, "SampleFile.TXT", OpenMode.Output, OpenAccess.Write)`
	`FileSystem.PrintLine(Channel, LineOfText)`
	`FileSystem.FileClose(Channel)`	

Java	`import java.io.FileWriter;`
	`import java.io.PrintWriter;`
	`import java.io.IOException;`
	`FileWriter fileHandle = new FileWriter("SampleFile.TXT", false);`
	`PrintWriter printLine = new PrintWriter(fileHandle);`
	`String lineOfText;`
	`printLine.printf("%s"+"%n", lineOfText);`
	`printLine.close();`
		Input output operations throw exceptions. The easiest way to manage these is to change your main heading to:
		`public static void main(String[] args) throws IOException`

Reading from a text file

An existing file can be read by a program. The following pseudocode statements provide facilities for reading from a file:

```
OPENFILE <filename> FOR READ          // open file for reading
READFILE <filename>, <stringVariable>  // read a line of text from the file
CLOSEFILE <filename>                   // close file
```

The following code examples demonstrate how to open, read from and close a file called `SampleFile.TXT` in each of the three languages.

Code examples

Python	`FileHandle = open("SampleFile.TXT", "r")`
	`LineOfText = FileHandle.readline()`
	`FileHandle.close ()`
		You specify the filename and mode ('r' for read) when you call the `open` function.

VB.NET	`Dim LineOfText As String`
	`Dim FileHandle As IO.StreamReader`
	`FileHandle = New IO.StreamReader("SampleFile.TXT")`
		The file is accessed through an object (see Chapter 27) called a `StreamReader`.

	`LineOfText = FileHandle.ReadLine()`
	`FileHandle.Close()`	

	Alternative method:
	`Dim LineOfText As String`
	`Dim Channel As Integer = 1`
	`FileSystem.FileOpen(Channel, "SampleFile.TXT", OpenMode.Input, OpenAccess.Read)`
	`FileSystem.Input(Channel, LineOfText)`
	`FileSystem.FileClose(Channel)`	

Java	`import java.io.IOException;`
	`import java.io.FileReader;`
	`import java.io.BufferedReader;`
	`FileReader fileHandle = new`
	`FileReader("SampleFile.TXT");`
	`BufferedReader textReader = new`
	`BufferedReader(fileHandle);`
	`String lineOfText = textReader.readLine();`
	`textReader.close();`
		There are other library classes that can be used for input/output, such as Scanner.

Appending to a text file

The following pseudocode statements provide facilities for appending to a file:

```
OPENFILE <filename> FOR APPEND      // open file for append
WRITEFILE <filename>, <stringValue> // write a line of text to the file
CLOSEFILE <filename>                // close file
```

The following code examples demonstrate how to open, append to and close a file called `SampleFile.TXT` in each of the three languages.

Code examples

Python	`FileHandle = open("SampleFile.TXT", "a")`
	`FileHandle.write(LineOfText)`
	`FileHandle.close()`
		You specify the filename and mode ('a' for append) when you call the `open` function.

VB.NET	`Dim FileHandle As IO.StreamWriter`
	`FileHandle = New`
	`IO.StreamWriter("SampleFile.TXT", True)`
	`FileHandle.WriteLine(LineOfText)`
	`FileHandle.Close()`
		The file is accessed through a `StreamWriter`. The extra parameter, `True`, tells the system to append to the object.

	Alternative method:
	`Dim LineOfText As String`
	`Dim Channel As Integer = 1`
	`FileSystem.FileOpen(Channel, "SampleFile.TXT", OpenMode.Append, OpenAccess.ReadWrite)`
	`FileSystem.Print(Channel, LineOfText)`
	`FileSystem.FileClose(Channel)`	

Java	`import java.io.FileWriter;`
	`import java.io.PrintWriter;`
	`import java.io.IOException;`
	`FileWriter fileHandle = new`
	`FileWriter("SampleFile.TXT"), true);`
	`PrintWriter printLine = new PrintWriter(fileHandle);`
	`String lineOfText;`
	`printLine.printf("%s"+"%n", lineOfText);`
	`printLine.close();`
		Input output throws exceptions. The easiest way is to change your main heading to:
		`public static void main(String[] args) throws IOException`

The end-of-file (EOF) marker

The following pseudocode statements read a text file and output its contents:

```
OPENFILE "Test.txt" FOR READ
WHILE NOT EOF("Test.txt") DO
    READFILE "Test.txt", TextString
    OUTPUT TextString
ENDWHILE
CLOSEFILE "Test.txt"
```

The following code examples demonstrate how to read and then output the contents of a file in each of the three languages.

Code examples

Python	`FileHandle = open("Test.txt", "r")`
	`LineOfText = FileHandle.readline()`
	`while len(LineOfText) > 0:`
	`   LineOfText = FileHandle.readline()`
	`   print(LineOfText)`
	`FileHandle.close()`
		There is no explicit EOF function. However, when a line of text has been read that only consists of the end-of-file marker, the line of text is of length 0.

VB.NET	`Dim LineOfText As String`
	`Dim FileHandle As System.IO.StreamReader`
	`FileHandle = New`
	`System.IO.StreamReader("Test.txt")`
	`Do Until FileHandle.EndOfStream`
	`   LineOfText = FileHandle.ReadLine()`
	`   Console.WriteLine(LineOfText)`
	`Loop`
	`FileHandle.Close()`
		When the end-of-file marker is detected, the `EndOfStream` method returns the value True and so the loop will end.

VB.NET	Alternative method:
	`Dim LineOfText As String`
	`Dim Channel As Integer = 1`
	`FileSystem.FileOpen(Channel, "Test.txt",`
	`OpenMode.Input, OpenAccess.Read)`
	`Do While Not FileSystem.EOF(Channel)`
	`   FileSystem.Input(Channel, LineOfText)`
	`   Console.WriteLine(LineOfText)`
	`Loop`
	`FileSystem.FileClose(Channel)`	

Java	`import java.io.IOException;`
	`import java.io.FileReader;`
	`import java.io.BufferedReader;`
	`FileReader fileHandle = new FileReader("Test.txt");`
	`BufferedReader textReader = new`
	`BufferedReader(fileHandle);`
	`String lineOfText = textReader.readLine();`
	`while (lineOfText != null)`
	`{`
	`   System.out.println(lineOfText);`
	`   lineOfText = textReader.readLine();`
	`}`
	`textReader.close();`
		There is no explicit EOF function. However, when a line of text has been read that only consists of the end-of-file marker, the line of text is effectively null.

TASK 14.15

Fred surveys the students at his college to find out their favourite hobby. He wants to present the data as a tally chart.

Fred plans to enter the data into the computer as he surveys the students. After data entry is complete, he wants to output the total for each hobby.

1	Reading books	\\\
2	Play computer games	\\\\\\\\
3	Sport	\\\\\
4	Programming	\\
5	Watching TV	\\\\\\\\\\\

He starts by writing an algorithm:

```
Initialise Tally array
REPEAT
```

```
    INPUT Choice // 1 for Reading, 2 for computer games,
                 // 3 for Sport, 4 for Programming, 5 for TV
                 // 0 to end input
    Increment Tally[Choice]
UNTIL Choice = 0
FOR Index = 1 TO 5
    OUTPUT Tally[Index]
NEXT Index
```

1 Write program code to declare and initialise the array `Tally : ARRAY[1:5] OF INTEGER`.

2 Write program code to implement the algorithm above.

3 Write program code to declare an array to store the hobby titles and rewrite the `FOR` loop of your program in part 2 so that the hobby title is output before each tally.

4 Write program code to save the array data in a text file.

5 Write program code to read the data from the text file back into the initialised array.

Reflection Point:

How much practice have you had writing programs? Did you get them to work?

How difficult did you find the different constructs?

Put the following in order of difficulty:

- Using a FOR loop
- Using a WHILE loop
- Using an IF ELSE statement
- Declaring a variable of a standard data type
- Declaring and using a constant
- Using a 1D array
- Using a nested loop to access each element in a 2D array
- Reading from and writing to a text file
- Using a built-in function
- Writing a procedure and calling it from the main program
- Writing a function and using its return value in an expression in the main program
- Using parameters with procedures and functions

Summary

- Programming constructs in Python, VB.NET and Java include:
  - declaration and assignment of constants and variables
  - the basic constructs of assignment, selection, repetition, input and output
  - built-in data types and functions.
- Code should be commented where it helps understanding.
- Boolean expressions are needed for conditions.
- Declaration of subroutines (functions and procedures) is done before the main program body.
- Calling a procedure is a program statement.

- Calling a function is done within an expression, for example an assignment.
- VB.NET and Java functions return exactly one value.
- Parameters can be passed to a subroutine. This is known as the interface.
- VB.NET passes parameters by value, as a default, but can return one or more values via parameters if they are declared as reference parameters.
- In Python, parameters can only pass values into a subroutine. The only way to update a value of a variable in the calling program is to return one or more values from a function.
- In Java, parameters can only pass values into a subroutine. The only way to update a value of a variable in the calling program is to return one value from a function. Note that object parameters are always passed by reference.
- When a subroutine is defined, parameters are the ‘placeholders’ for values passed into a subroutine.
- Arguments are the values passed to the subroutine when it is called.

Exam-style Questions

1 Preeti wants a program to output a conversion table for ounces to grams (1 ounce is 28.35 grams). She writes an algorithm using Structured English:

```
OUTPUT "Ounces Grams"
FOR Ounces FROM 1 TO 30
    SET Grams TO Rounded(Ounces * 28.35) // whole number of grams only
    OUTPUT Ounces and Grams
```

Write pseudocode to implement the algorithm. Include formatting, so that the output is tabulated. [7]

2 Write pseudocode to accept an input string `UserID`. The pseudocode is to test the `UserID` format. A valid format `UserID` consists of three upper case letters and four digits. The program is to output a message whether `UserID` is valid or not. [5]

3 Write pseudocode for a procedure `OutputTimesTable` that takes one integer parameter, `n`, and outputs the times table for `n`. For example the procedure call `OutputTimesTable(5)` should produce:

1 × 5 = 5
2 × 5 = 10
3 × 5 = 15
4 × 5 = 20
5 × 5 = 25
6 × 5 = 30
7 × 5 = 35
8 × 5 = 40
9 × 5 = 45
10 × 5 = 50 [6]

4 Write pseudocode for a function `isDivisible()` that takes two integer parameters, `x` and `y`. The function is to return the value True or False to indicate whether `x` is exactly divisible by `y`. For example, `isDivisible(24, 6)` should return True and `isDivisible(24, 7)` should return False. [6]

5 A poultry farm packs eggs into egg boxes. Each box takes six eggs. Boxes must not contain fewer than six eggs.

Write pseudocode for a procedure `EggsIntoBoxes` that takes an integer parameter, `NumberOfEggs`. The procedure is to calculate how many egg boxes can be filled with the given number of eggs and how many eggs will be left over. The procedure is to return two values as parameters, `NumberOfBoxes` and `EggsLeftOver`. [9]

6 In a certain country, car registrations consist 7 alphanumerical characters. The format of a car registration is either

LLNNLLL

or

LLLNNLL

where L is any capital letter and N is any numeral 0 to 9.

Use pseudocode to write a function that takes a string as parameter and returns TRUE if the format is valid and FALSE otherwise.

The string-handling functions available are those listed in Table 14.06. [7]

Chapter 15:
Software development

Learning objectives

By the end of this chapter you should be able to:

- show understanding of the purpose of a development life cycle
- show understanding of the need of different development life cycles depending on the program being developed
- describe the principles, benefits and drawbacks of each type of life cycle: waterfall, iterative and rapid application development
- show understanding of the analysis, design, coding, testing and maintenance stages in the program development cycle
- use a structure chart to decompose a problem into sub-tasks
- use a structure chart to express the parameters passed between the various modules/procedures/functions which are part of the algorithm design
- describe the purpose of a structure chart
- construct a structure chart for a given problem
- derive equivalent pseudocode from a structure chart
- show understanding of the purpose of state-transition diagrams to document an algorithm
- show understanding of ways of exposing and avoiding faults in programs
- locate and identify the different types of errors (syntax errors, logic errors and run-time errors)
- correct identified errors
- show understanding of available testing methods (dry-run, walkthrough, white-box, black-box, integration, alpha, beta, acceptance, stub)
- select appropriate data for a given testing method
- show understanding of the need for a test strategy and test plan and their likely contents
- choose appropriate test data for a test plan (normal, abnormal, extreme/boundary)
- show understanding of the need for continuing maintenance of a system and the differences between each type of maintenance (corrective, adaptive, perfective)
- analyse an existing program and make amendments to enhance functionality.

15.01 Stages in the program development life cycle

Developing a program involves different stages. You solve a problem by designing the solution using Structured English, a flowchart and / or pseudocode (see Chapters 12 and 13). You write the program code and test it.

When large software systems are required to solve big problems, these stages are more formal, especially when more people are involved in the development. Before a solution can be designed, the problem needs to be analysed. When the program works and is being used, issues might arise that require changes. This is known as maintenance.

Analysis

The first step in solving a problem is to investigate the issues and the current system if there is one. The problem needs to be defined clearly and precisely. A 'requirements specification' is drawn up.

The next step is planning a solution. Sometimes there is more than one solution. You need to decide which is the most appropriate.

The third step is to decide how to solve the problem:

- bottom-up: start with a small sub-problem and then build on this
- top-down: stepwise refinement using pseudocode, flowcharts or structure charts.

Design

You have a solution in mind. How do you design the solution in detail? Chapter 12 (Section 12.05) showed that an identifier table is a good starting point. This leads you to thinking about data structures: do you need a 1D array or a 2D array to store data while it is processed? Do you need a file to store data long-term?

Plan your algorithm by drawing a flowchart or writing pseudocode.

Coding

When you have designed your solution, you might need to choose a suitable high-level programming language. If you know more than one programming language, you have to weigh up the pros and cons of each one. Looking at Chapter 14, you need to decide which programming language would best suit the problem you are trying to solve and which language you are most familiar with.

TIP

This stage is often referred to as implementation.

You implement your algorithm by converting your pseudocode into program code. When you start writing programs you might find it takes several attempts before the program compiles. When it finally does, you can execute it. It might 'crash', meaning that it stops working. In this case, you need to debug the code. The program might run and give you some output. This is the Eureka moment: 'It works!!!!'. But does the program do what it was meant to do?

Testing

Only thorough testing can ensure the program really works under all circumstances (see Sections 15.06 and 15.07).

There are several different development methodologies. These include the waterfall, the iterative and the rapid application development model.

Discussion Point:

Do you think that all programs can be totally error-free?

The program development life cycle

The program development life cycle follows the defined stages of analysis, design, coding (implementation), testing and maintenance. When maintenance no longer results in a program fit for purpose, the development starts again, therefore creating a cycle (see Figure 15.01).

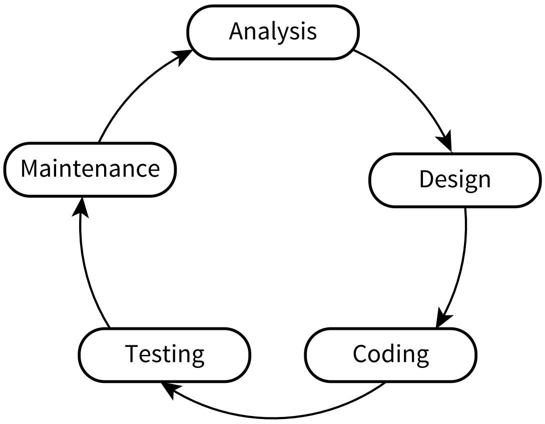


Figure 15.01 The program development life cycle

The waterfall model

Figure 15.02 shows the waterfall model.

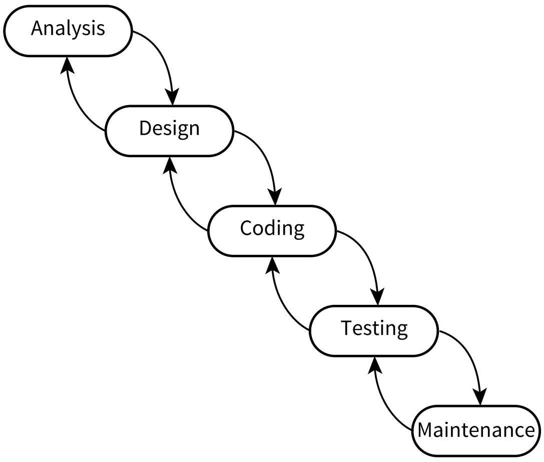


Figure 15.02 The waterfall model

The arrows going down represent the fact that the results from one stage are input into the next stage. The arrows leading back up to an earlier stage reflect the fact that often more work is required at an earlier stage to complete the current stage.

Benefits include the following.

- Simple to understand as the stages are clearly defined.
- Easy to manage due to the fixed stages in the model. Each stage has specific outcomes.
- Stages are processed and completed one at a time.
- Works well for smaller projects where requirements are very well understood.

Drawbacks include the following.

- No working software is produced until late during the life cycle.
- Not a good model for complex and object-oriented projects.
- Poor model for long and ongoing projects.
- Cannot accommodate changing requirements.

- It is difficult to measure progress within stages.
- Integration is done at the very end, which doesn't allow identifying potential technical or business issues early.

The iterative model

An iterative life cycle model does not attempt to start with a full specification of requirements. Instead, development starts with the implementation of a small subset of the program requirements. Repeated (iterative) reviews to identify further requirements eventually result in the complete system.

Benefits include the following.

- There is a working model of the system at a very early stage of development, which makes it easier to find functional or design flaws. Finding issues at an early stage of development means corrective measures can be taken more quickly.
- Some working functionality can be developed quickly and early in the life cycle.
- Results are obtained early and periodically.
- Parallel development can be planned.
- Progress can be measured.
- Less costly to change the scope/requirements.
- Testing and debugging of a smaller subset of program is easy.
- Risks are identified and resolved during iteration.
- Easier to manage risk – high-risk part is done first.
- With every increment, operational product is delivered.
- Issues, challenges and risks identified from each increment can be utilised/applied to the next increment.
- Better suited for large and mission-critical projects.
- During the life cycle, software is produced early, which facilitates customer evaluation and feedback.

Drawbacks include the following.

- Only large software development projects can benefit because it is hard to break a small software system into further small serviceable modules.
- More resources may be required.
- Design issues might arise because not all requirements are gathered at the beginning of the entire life cycle.
- Defining increments may require definition of the complete system.

The Rapid Application Development (RAD) model

RAD is a software development methodology that uses minimal planning. Instead it uses prototyping. A prototype is a working model of part of the solution.

In the RAD model, the modules are developed in parallel as prototypes and are integrated to make the complete product for faster product delivery. There is no detailed preplanning. Changes are made during the development process.

The analysis, design, code and test phases are incorporated into a series of short, iterative development cycles.

Benefits include the following.

- Changing requirements can be accommodated.
- Progress can be measured.
- Productivity increases with fewer people in a short time.
- Reduces development time.
- Increases reusability of components.
- Quick initial reviews occur.
- Encourages customer feedback.
- Integration from very beginning solves a lot of integration issues.

Drawbacks include the following.

- Only systems that can be modularised can be built using RAD.
- Requires highly skilled developers/designers.
- Suitable for systems that are component based and scalable.
- Requires user involvement throughout the life cycle.
- Suitable for projects requiring shorter development times.

15.02 Program design using structure charts

An alternative approach to modular design is to choose the sub-tasks and then construct a **structure chart** to show the interrelations between the modules. Each box of the structure chart represents a module. Each level is a refinement of the level above.

A structure chart also shows the interface between modules, the variables. These variables are referred to as 'parameters' (see Section 14.10). A **parameter** supplying a value to a lower-level module is shown as a downwards pointing arrow. A parameter supplying a new value to the module at the next higher level is shown as an upward pointing arrow.

Figure 15.03 shows a structure chart for a module that calculates the average of two numbers. The top-level box is the name of the module, which is refined into the three sub-tasks of Level 1. The input numbers (parameters Number1 and Number2) are passed into the 'Calculate Average' sub-task and then the Average parameter is passed into the 'OUTPUT Average' sub-task. The arrows show how the parameters are passed between the modules. This parameter passing is known as the 'interface'.

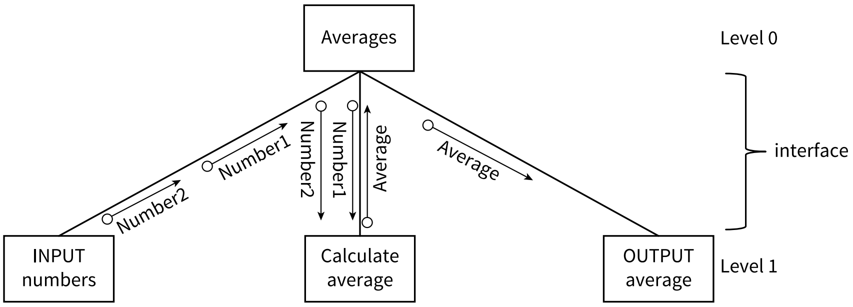


Figure 15.03 Structure chart for a module that calculates the average of two numbers

TASK 15.01

Draw a structure chart for the following module: Input a number of km, output the equivalent number of miles.

Structure charts can also show control information: selection and repetition.

The simple number-guessing game that was introduced in Chapter 12 (Section 12.06) could be modularised and presented as a structure chart, as shown in Figure 15.04.

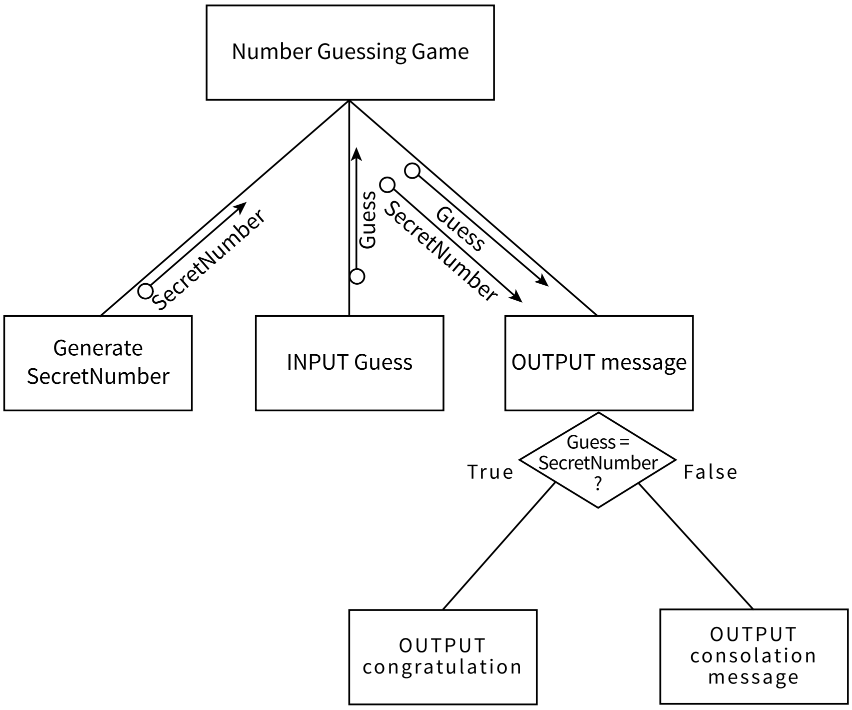


Figure 15.04 Structure chart for number-guessing game with only one guess allowed

The diamond shape shows a condition that is either True or False. So either one branch or the other will be followed.

Figure 15.05 shows the structure chart for the pyramid-drawing program from Worked Example 12.10. The semi-circular arrow represents repetition of the modules below the arrow. The label shows the condition when repetition occurs.

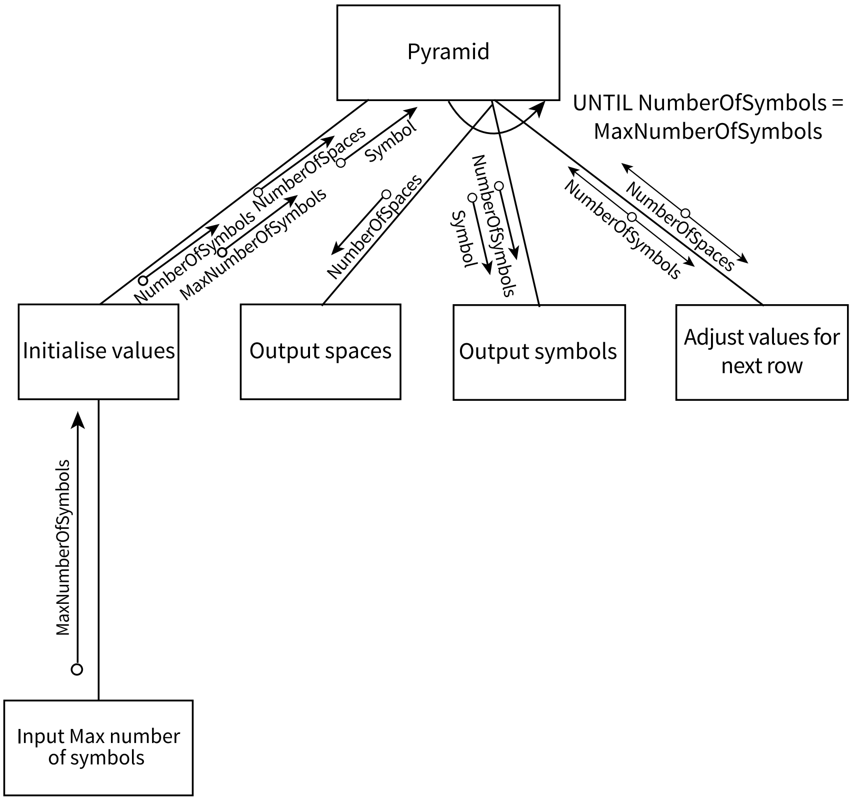


Figure 15.05 Structure chart for pyramid-drawing program

TASK 15.02

Amend the structure chart for the number-guessing game (Figure 15.04) to include repeated guesses until the player guesses the secret number. The output should include the number of guesses made.

TASK 15.03

Draw a structure chart for the following problem: A user attempts to log on with a user ID. User IDs and passwords are stored in two 1D arrays (lists). The algorithm searches the list of user IDs and looks up the password in the password list. The user is given three chances to input the correct password. If the correct password is entered, a suitable message is output. If the third attempt is incorrect, a warning message is output.

Structure charts help programmers to visualise how modules are interrelated and how they interface with each other. When looking at a larger problem this becomes even more important. Figure 15.06 shows a structure chart for the Connect 4 program (Task 13.06). It uses the following symbols:

- An arrow with a solid round end ●⟶ shows that the value transferred is a flag (a Boolean value)
- A double-headed arrow ⟵○⟶ shows that the variable value is updated within the module.

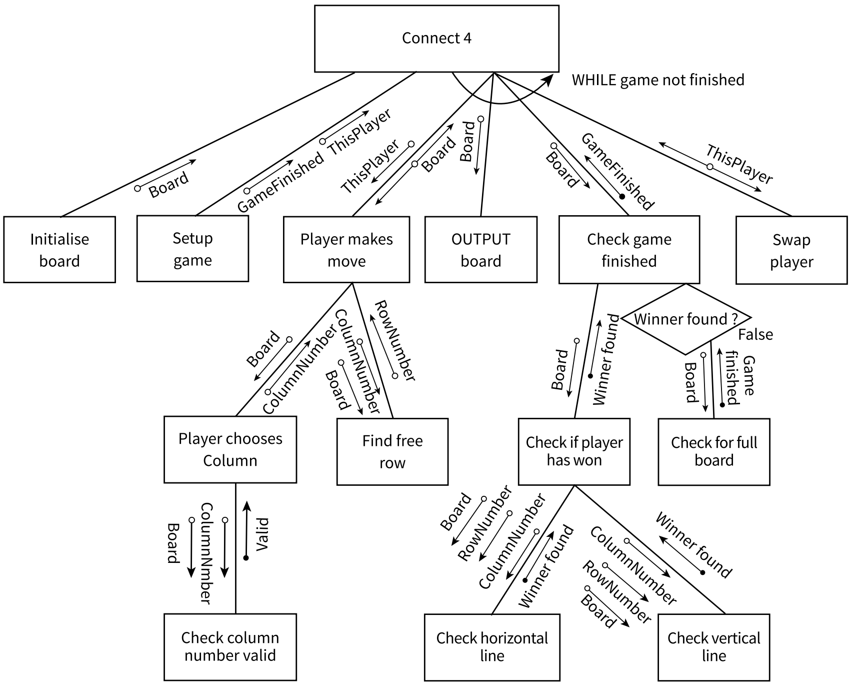


Figure 15.06 Structure chart for the Connect 4 game

15.03 Deriving pseudocode from a structure chart

Let’s look at the pyramid problem again (Figure 15.05). In Worked Example 12.10, a modular solution was created without using a structure chart and all variables were global. Now we are going to use local variables and parameters. The reason for using local variables and parameters is that modules are then self-contained and any changes to variables do not have accidental effects on a variable value elsewhere.

The top-level module, Pyramid, calls four modules. When a module is called, we supply the parameters in parentheses after the module identifier. This gives the following pseudocode:

```
MODULE Pyramid
    CALL SetValues(NumberOfSymbols, NumberOfSpaces, Symbol, MaxNumberOfSymbols)
    REPEAT
        CALL OutputSpaces(NumberOfSpaces)
        CALL OutputSymbols(NumberOfSymbols, Symbol)
        CALL AdjustValuesForNextRow(NumberOfSpaces, NumberOfSymbols)
    UNTIL NumberOfSymbols > MaxNumberOfSymbols
ENDMODULE

PROCEDURE SetValues(NumberOfSymbols, NumberOfSpaces, Symbol, MaxNumberOfSymbols)
    INPUT Symbol
    CALL InputMaxNumberOfSymbols
    NumberOfSpaces ← (MaxNumberOfSymbols – 1) / 2
    NumberOfSymbols ← 1
ENDPROCEDURE

PROCEDURE InputMaxNumberOfSymbols(MaxNumberOfSymbols)
    REPEAT
        INPUT MaxNumberOfSymbols
    UNTIL MaxNumberOfSymbols MOD 2 = 1
ENDPROCEDURE

PROCEDURE OutputSpaces(NumberOfSpaces)
    FOR Count ← 1 TO NumberOfSpaces
        OUTPUT Space // without moving to next line
    NEXT Count
ENDPROCEDURE

PROCEDURE OutputSymbols(NumberOfSymbols, Symbol)
    FOR Count← 1 TO NumberOfSymbols
        OUTPUT Symbol // without moving to next line
    NEXT Count
    OUTPUT Newline // move to the next line
ENDPROCEDURE

PROCEDURE AdjustValuesForNextRow(NumberOfSpaces, NumberOfSymbols)
    NumberOfSpaces ← NumberOfSpaces – 1
    NumberOfSymbols ← NumberOfSymbols + 2
ENDPROCEDURE
```

Note that a structure chart does not give details about how parameters are passed: by reference or by value.

TASK 15.04

1 Write pseudocode to implement the structure chart from Figure 12.03 (for the average of two numbers).

2 Write pseudocode to implement the structure chart from Figure 12.04 (for the number-guessing game).

3 Amend the pseudocode from Worked Example 13.05 to implement the interface shown in the structure chart from Figure 15.06.

Discussion Point:

The full rules of Connect 4 are that a diagonal of four tokens also is a winning line. Where in Figure

15.06 should the module to check for a diagonal be added? What parameters are required for this module? Does this additional module require further stepwise refinement?

15.04 Program design using state-transition diagrams

A computer system can be seen as a **finite state machine (FSM)**. An FSM has a start state. An input to the FSM produces a transformation from one state to another state.

The information about the states of an FSM can be presented in a **state-transition table**.

Table 15.01 shows an example FSM represented as a state-transition table.

- If the FSM is in state S1, an input of a causes no change of state.
- If the FSM is in state S1, an input of b transforms S1 to S2.
- If the FSM is in state S2, an input of b causes no change of state.
- If the FSM is in state S2, an input of a transforms S2 to S1.

A **state-transition diagram** can be used to describe the behaviour of an FSM. Figure 15.07 shows the start state as S1 (denoted by ●⟶). If the FSM has a final state (also known as the halting state), this is shown by a double-circled state (S1 in the example).

		current state	
		S1	**S2**
input	**a**	S1	S1
	b	S2	S2

Table 15.01 State-transition table

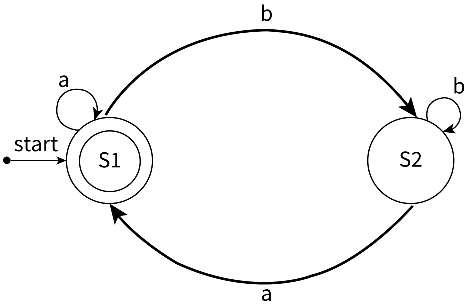


Figure 15.07 State-transition diagram

If an input causes an output this is shown by a vertical bar (as in Figure 15.08). For example, if the current state is S1, an input of b produces output c and transforms the FSM to state S2.

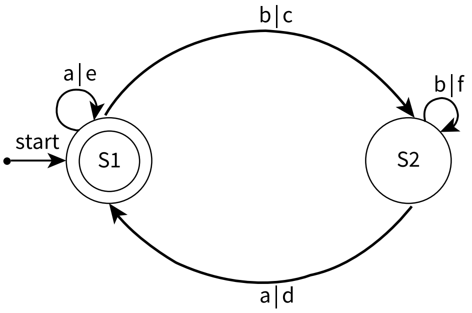


Figure 15.08 State-transition diagram with outputs

A Finite State Machine with outputs is also known as a Mealy Machine.

WORKED EXAMPLE 15.01

Creating a state-transition diagram for an intruder detection system

A program is required that simulates the behaviour of an intruder detection system.

Description of the system: The system has a battery power supply. The system is activated when the start button is pressed. Pressing the start button when the system is active has no effect. To de-activate the system, the operator must enter a PIN. The system goes into alert mode when a sensor is activated. The system will stay in alert mode for two minutes. If the system has not been de-activated within two minutes an alarm bell will ring.

We can complete a state-transition table (Table 15.02) using the information from the system description.

Current state	Event	Next state
System inactive	Press start button	System active
System active	Enter PIN	System inactive
System active	Activate sensor	Alert mode
System active	Press start button	System active
Alert mode	Enter PIN	System inactive
Alert mode	2 minutes pass	Alarm bell ringing
Alert mode	Press start button	Alert mode
Alarm bell ringing	Enter PIN	System inactive
Alarm bell ringing	Press start button	Alarm bell ringing

Table 15.02 State-transition table for intruder detection simulation

The start state is ‘System inactive’. We can draw a state-transition diagram (Figure 15.09) from the information in Table 15.02.

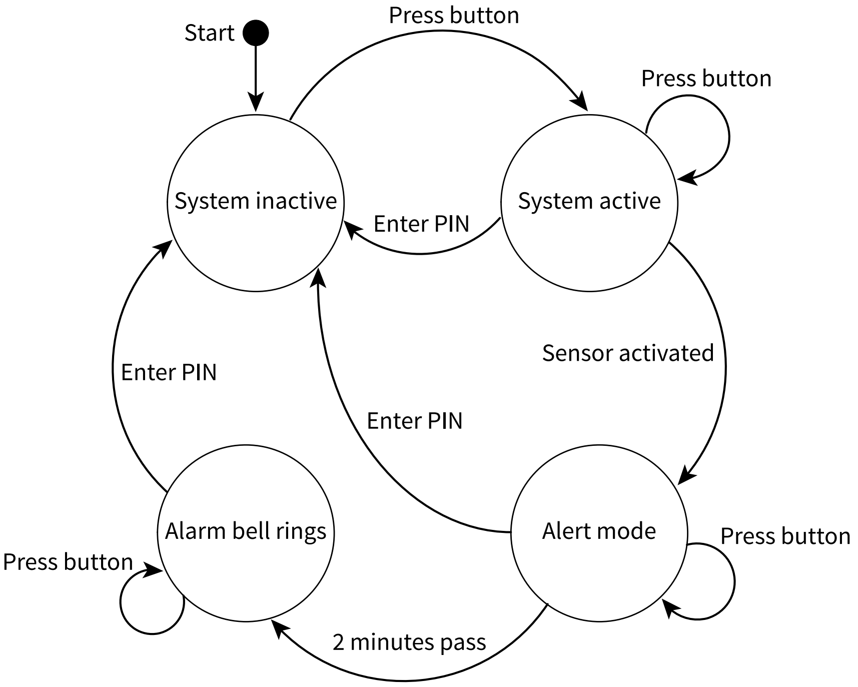


Figure 15.09 State-transition diagram for intruder detection system

WORKED EXAMPLE 15.02

Creating a state-transition diagram for a two’s complement FSM

A finite state machine has been designed that will take as input a positive binary integer, one bit at a time, starting with the least significant bit. The FSM converts the binary integer into the two's complement negative equivalent. The method to be used is as follows.

1 Output the bits input up to and including the first 1.

2 Output the other bits following this scheme:

2.1 For each 1, output a 0.

2.2 For each 0, output a 1.

This information is represented in the state-transition table shown in Table 15.03.

Current state	S1	S1	S2	S2
Input bit	0	1	0	1
Next state	S1	S2	S2	S2
Output bit	0	1	1	0

Table 15.03 State-transition table with outputs

This method can be represented as the state-transition diagram in Figure 15.10.

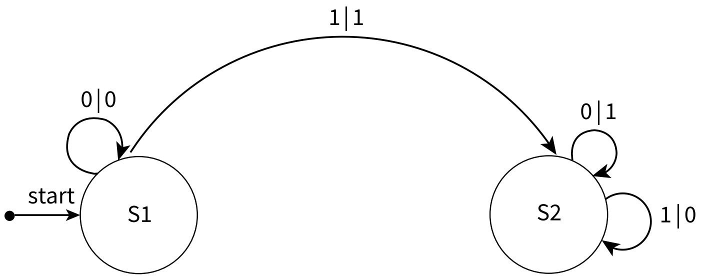


Figure 15.10 State-transition diagram for a two's complement FSM

TASK 15.05

Write a program that simulates the intruder detection system in Worked Example 15.01.

Question 15.01

What is the output from the FSM represented by the state-transition diagram in Figure 15.10, when the input is `0101`?

Extension Question 15.01

Does the FSM in Figure 15.10 work for converting a negative binary number into its positive equivalent?

15.05 Types of error

Why errors occur and how to find them

Software may not perform as expected for a number of reasons, such as:

- the programmer has made a coding mistake
- the requirement specification was not drawn up correctly
- the software designer has made a design error
- the user interface is poorly designed, and the user makes mistakes
- computer hardware experiences failure.

How are errors found? The end user might report an error. This is not good for the reputation of the software developer. Testing software before it is released for general use is essential. Research has shown that the earlier an error can be found, the cheaper it is to fix it. It is very important that software is tested throughout its development.

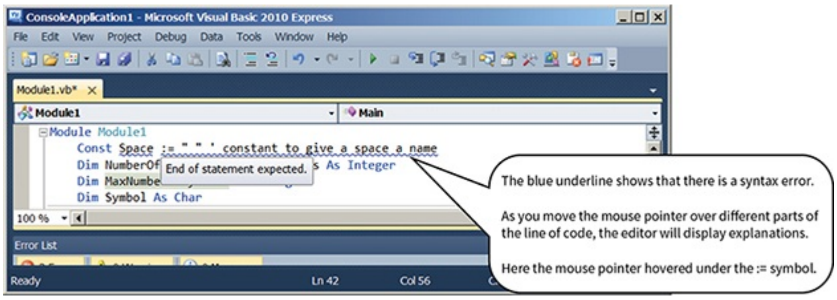
The purpose of testing is to discover errors. Edsger Dijkstra, a famous Dutch computer scientist, said ‘Program testing can be used to show the presence of bugs, but never to show their absence!’.

Finding syntax errors is easy. The compiler/interpreter will find them for you and usually gives you a hint as to what is wrong.

Depending on your development environment editor, some syntax errors may be flagged up by your editor, so you can correct these as you go along. A **syntax error** is a ‘grammatical’ error, in which a program statement does not follow the rules of the high-level language constructs.

Some syntax errors might only become apparent when you are using an interpreter or compiler to translate your program. Interpreters and compilers work differently (see Chapter 8, Section 8.05, and Chapter 20, Section 20.06). When a program compiles successfully, you know there will be no syntax errors remaining.

This is not the case with interpreted programs. Only statements that are about to be executed will be syntax checked. So, if your program has not been thoroughly tested, it might even have syntax errors remaining.


ConsoleApplication1 - Microsoft Visual Basic 2010 Express
File Edit View Project Debug Data Tools Window Help
Module1.vb*
Module1
Main
100 %
Error List
Ready
Ln 42
Col 56

```
Module Module1
    Const Space := " " ' constant to give a space a name
    Dim NumberOf                       s As Integer
    Dim MaxNumbe
    Dim Symbol As Char
```

End of statement expected.

The blue underline shows that there is a syntax error.

As you move the mouse pointer over different parts of the line of code, the editor will display explanations.

Here the mouse pointer hovered under the := symbol.

Figure 15.11 Syntax error in a Visual Basic program

Figure 15.11 gives an example of how a compiler flags a syntax error. The compiler stops when it first notices a syntax error. The error is often on the previous line. The compiler can’t tell until it gets to the next line of code and finds an unexpected keyword.

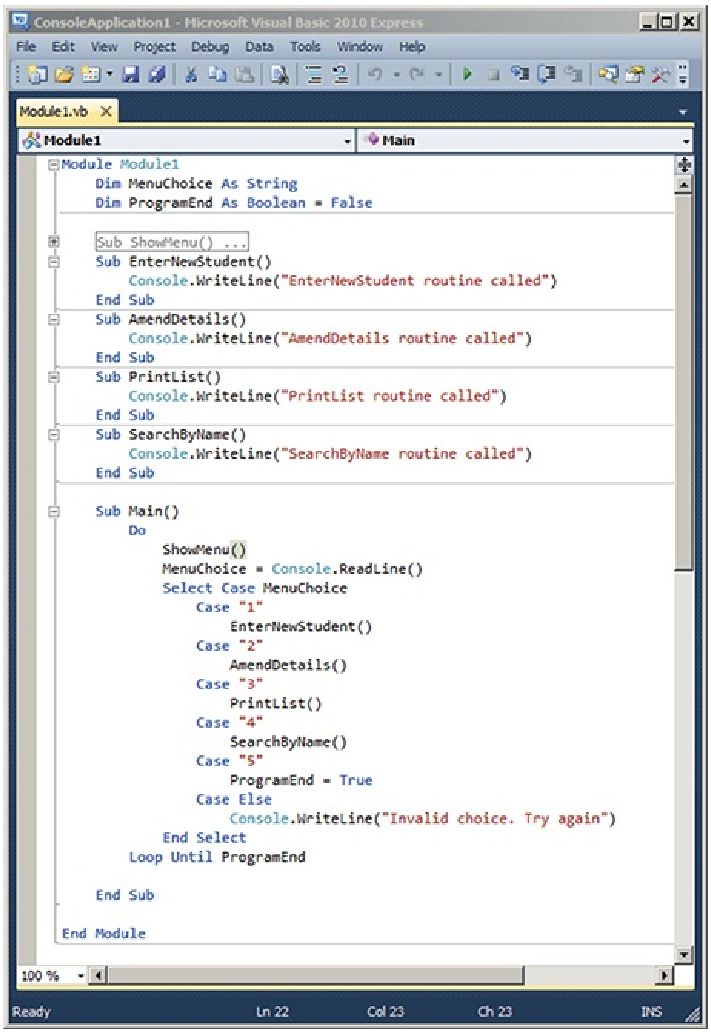
Much more difficult to find are **logic errors** and **run-time errors**. A run-time error occurs when program execution comes to an unexpected halt or ‘crash’ or it goes into an infinite loop and ‘freezes’.

Both of these types of error can only be found by careful testing. The danger of such errors is that they may only show up under certain circumstances. If a program crashes every time it is executed, it is obvious there is an error. If the program is used frequently and appears to work until a certain set of data causes a malfunction, that is much more difficult to discover without perhaps serious consequences.

15.06 Testing methods

Stub testing

When you develop a user interface, you might wish to test it before you have implemented all the facilities. You can write a ‘stub’ for each procedure (see Figure 15.12). The procedure body only contains an output statement to acknowledge that the call was made. Each option the user chooses in the main program will call the relevant procedure.


ConsoleApplication1 - Microsoft Visual Basic 2010 Express
File Edit View Project Debug Data Tools Window Help
Module1.vb
Module1
Main
100 %
Ready
Ln 22
Col 23
Ch 23
INS

```
Module Module1
    Dim MenuChoice As String
    Dim ProgramEnd As Boolean = False

    Sub ShowMenu() ...
    Sub EnterNewStudent()
        Console.WriteLine("EnterNewStudent routine called")
    End Sub
    Sub AmendDetails()
        Console.WriteLine("AmendDetails routine called")
    End Sub
    Sub PrintList()
        Console.WriteLine("PrintList routine called")
    End Sub
    Sub SearchByName()
        Console.WriteLine("SearchByName routine called")
    End Sub

    Sub Main()
        Do
            ShowMenu()
            MenuChoice = Console.ReadLine()
            Select Case MenuChoice
                Case "1"
                    EnterNewStudent()
                Case "2"
                    AmendDetails()
                Case "3"
                    PrintList()
                Case "4"
                    SearchByName()
                Case "5"
                    ProgramEnd = True
                Case Else
                    Console.WriteLine("Invalid choice. Try again")
            End Select
        Loop Until ProgramEnd

    End Sub

End Module
```

Figure 15.12 VB.NET stub testing

Black-box testing

As the programmer, you can see your program code and your testing will involve knowledge of the code (see white-box testing).

As part of thorough testing, a program should also be tested by other people, who do not see the program code and don’t know how the solution was coded.

Such program testers will look at the program specification to see what the program is meant to do, devise **test data** and work out expected results. Test data usually consists of normal data values, extreme/boundary data values and erroneous/abnormal data values.

The tester then runs the program with the test data and records their results. This method of testing is called **black-box testing** because the tester can’t see inside the program code: the program is a ‘black box’.

Where the actual results don’t match the expected results, a problem exists. The programmer needs to find the reason for this discrepancy before correcting the program (see Section 15.08). Once black-box testing has established that there is an error, debugging software or dry-running have to be used to find the lines of code that need correcting.

White-box testing

How can we check that code works correctly? We choose suitable test data that checks every path through the code. This is called **white-box testing**.

WORKED EXAMPLE 15.03

White-box testing of pseudocode

This is the pseudocode from Worked Example 12.02 in Chapter 12:

```
INPUT Number1
INPUT Number2
INPUT Number3
IF Number1 > Number2
  THEN            // Number1 is bigger
    IF Number1 > Number3
      THEN
        OUTPUT Number1
      ELSE
        OUTPUT Number3
    ENDIF
  ELSE            // Number2 is bigger
    IF Number2 > Number3
      THEN
        OUTPUT Number2
      ELSE
        OUTPUT Number3
    ENDIF
ENDIF
```

To test it, we need four sets of numbers with the following characteristics.

- The first number is the largest.
- The first number is larger than the second number; the third number is the largest.
- The second number is the largest.
- The second number is larger than the first number; the third number is the largest.

Note that it does not matter what exact values are chosen as test data. The important point is that the values differ in such a way that each part of the nested `IF` statement is checked. Table 15.04 lists four sets of test data and the results from them. The parts of the algorithm not entered for a particular set of data are greyed out. This makes it easier to see that each part has been checked after all four tests have been done.

Line of algorithm	Test 1	Test 2	Test 3	Test 4
`INPUT Number1`	15	12	12	8
`INPUT Number2`	12	8	15	12
`INPUT Number3`	8	15	8	15
`IF Number1 > Number2`	TRUE	TRUE	FALSE	FALSE

`THEN`
`IF Number1 > Number3`
	TRUE	FALSE		

`THEN`	Output 15			

`OUTPUT Number1`				

`ELSE`
`OUTPUT Number3`
`ENDIF`
		Output 15		

`ELSE`
`IF Number2 > Number3`
			TRUE	FALSE

`THEN`
`OUTPUT Number2`
			Output 15	

`ELSE`
`OUTPUT Number3`
`ENDIF`
`ENDIF`
				Output 15

Table 15.04 Testing the validity of the nested IF statement

Dry-running an algorithm

A good way of checking that an algorithm works as intended is to **dry-run** the algorithm using a **trace table** and different test data. This is also known as a **walk through**.

The idea is to write down the current contents of all variables and conditional values at each step of the algorithm.

WORKED EXAMPLE 15.04

Tracing an algorithm

Here is the algorithm of the number-guessing game:

```
SecretNumber ← 34
INPUT "Guess a number: " Guess
NumberOfGuesses ← 1
REPEAT
    IF Guess = SecretNumber
      THEN
        OUTPUT "You took ", NumberOfGuesses, " guesses”
      ELSE
        IF Guess > SecretNumber
          THEN
            INPUT "Guess a smaller number: " Guess
          ELSE
            INPUT "Guess a larger number: " Guess
        ENDIF
         NumberOfGuesses ← NumberOfGuesses + 1
    ENDIF
UNTIL Guess = SecretNumber
```

To test the algorithm, construct a trace table (Table 15.05) with one column for each variable used in the algorithm and also for the condition `Guess > SecretNumber`

Now carefully look at each step of the algorithm and record what happens. Note that we do not tend to write down values that don’t change. Here `SecretNumber` does not change after the initial assignment, so the column is left blank in subsequent rows.

SecretNumber	Guess	NumberOfGuesses	Guess > SecretNumber	Message
34	5	1	FALSE	...larger...
	55	2	TRUE	...smaller...

	30	3	FALSE	...larger...
	42	4	TRUE	...smaller...
	36	5	TRUE	...smaller...
	33	6	FALSE	...larger...
	34	7		... 7 guesses

Table 15.05 Trace table for number-guessing game

We only make an entry in a cell when an assignment occurs. Values remain in variables until they are overwritten. So a blank cell means that the value from the previous entry remains.

It is important to start filling in a new row in the trace table for each iteration (each time round the loop).

TIP

When learning to complete trace tables and to ensure you follow every line of code in the correct sequence, you can number the lines of the algorithm and add a column for the line numbers in your trace table (see Worked Example 15.05 Trace Table 15.06).

WORKED EXAMPLE 15.05

Tracing an algorithm

To test the improved algorithm of Worked Example 13.03 (bubble sort), dry-run the algorithm by completing the trace table (Table 15.06).

```
01 MaxIndex ← 7
02 n ← MaxIndex - 1
03 REPEAT
04     NoMoreSwaps ← TRUE
05     FOR j ← 1 TO n
06         IF MyList[j] > MyList[j + 1]
07           THEN
08             Temp ← MyList[j]
09             MyList[j] ← MyList[j + 1]
10             MyList[j + 1] ← Temp
11             NoMoreSwaps ← FALSE
12         ENDIF
13     NEXT j
14     n ← n – 1
15 UNTIL NoMoreSwaps = TRUE
```

Line Numbers	Max Index	n	No-MoreSwaps	j	MyList[j] > MyList[j + 1]	Temp	MyList						
							[1]	[2]	[3]	[4]	[5]	[6]	[7]
01,02	7	6					5	34	98	7	41	19	25
03,04,05,06,12			TRUE	1	FALSE								
13,05,06,12				2	FALSE								
13,05,06,07,08,09,10				3	TRUE	98			7	98			
11,12			FALSE										
13,05,06,07,08,09,10				4	TRUE	98				41	98		
11,12			FALSE										
13,05,06,07,08,09,10				5	TRUE	98					19	98	
11,12			FALSE										
13,05,06,07,08,09,10				6	TRUE	98						25	98
11,12			FALSE										
13,14,15		5											
03,04,05,06,12			TRUE	1	FALSE								
13,05,06,07,08,09,10				2	TRUE	34		7	34				
11,12			FALSE										
13,05,06,12				3	FALSE								
13,05,06,07,08,09,10				4	TRUE	41				19	41		
11,12			FALSE										
13,05,06,07,08,09,10				5	TRUE	41					25	41	
11,12			FALSE										
13,14,15		4											
03,04,05,06,12			TRUE	1	FALSE								
13,05,06,12				2	FALSE								
13,05,06,07,08,09,10				3	TRUE	34			19	34			
11,12			FALSE										
13,05,06,07,08,09,10				4	TRUE	34				25	34		
11,12			FALSE										
13,14,15		3											
03,04,05,06,12			TRUE	1	FALSE								
13,05,06,12				2	FALSE								
13,05,06,12				3	FALSE								
13,14,15		2											

Table 15.06 Trace table for improved bubble sort algorithm

TASK 15.06

Design a trace table for the following algorithm:

```
FUNCTION ConvertFromHex(HexString : STRING) RETURNS INTEGER
    DECLARE ValueSoFar, HexValue, HexLength, i : INTEGER
    DECLARE HexDigit : CHAR
    ValueSoFar ← 0
    HexLength ← Length(HexString)
    FOR i ← 1 TO HexLength
        HexDigit ← HexString[i]
        CASE OF HexDigit
          'A': HexValue ← 10
          'B': HexValue ← 11
          'C': HexValue ← 12
          'D': HexValue ← 13
          'E': HexValue ← 14
          'F': HexValue ← 15
          OTHERWISE HexValue ← StringToInt(HexDigit)
        ENDCASE
        ValueSoFar ← ValueSoFar * 16 + HexValue
    NEXT i
    RETURN ValueSoFar
ENDFUNCTION
```

Dry-run the function call `ConvertFromHex('A5')` by completing the trace table.

These testing methods are used early on in software development, for example when individual modules are written. Sometimes programmers themselves use these testing methods. In larger software development organisations, separate software testers will be employed.

Discussion Point:

Do you think that a program tester will find errors the programmer did not know about? You can try out the idea by letting your friends test a program that you think works perfectly.

Software often consists of many modules, sometimes written by different programmers. Each individual module might have passed all the tests, but when modules are joined together into one program, it is vital that the whole program is tested. This is known as **integration testing**. Integration testing is usually done incrementally. This means that a module at a time is added and further testing is carried out before the next module is added.

Software will be tested in-house by software testers before being released to customers. This type of testing is called **alpha testing**.

Bespoke software (written for a specific customer) will then be released to the customer. The customer will check that it meets their requirements and works as expected. This stage is referred to as **acceptance testing**. It is generally part of the hand-over process. On successful acceptance testing, the customer will sign off the software.

When software is not bespoke but produced for general sale, there is no specific customer to perform acceptance testing and sign off the software. So, after alpha testing, a version is released to a limited audience of potential users, known as ‘beta testers’. These beta testers will use the software and test it in their own environments. This early release version is called a beta version and the chosen users perform **beta testing**. During beta testing, the users will feed back to the software house any problems they have found, so that the software house can correct any reported faults.

15.07 Test strategy, test plans and test data

During the design stage of a software project, a suitable testing strategy must be worked out to ensure rigorous testing of the software from the very beginning. Consideration should be given to which testing methods are appropriate for the project in question. A carefully designed test plan has to be produced.

It is important to recognise that large programs cannot be exhaustively tested but it is important that systematic testing finds as many errors as possible. We therefore need a test plan. In the first instance, an outline plan is designed, for example:

- flow of control: does the user get appropriate choices and does the chosen option go to the correct module?
- validation of input: has all data been entered into the system correctly?
- do loops and decisions perform correctly?
- is data saved into the correct files?
- does the system produce the correct results?

This outline test plan needs to be made into a detailed test plan.

How can we carry out these tests? We need to select data that will allow us to see whether it is handled correctly. This type of data is called 'test data'. It differs from real, live data because it is specifically chosen with a view of testing different possibilities. We distinguish between different types of test data, listed in Table 15.07.

Type of test data	Explanation
Normal (valid)	Typical data values that are valid
Abnormal (erroneous)	Data values that the system should not accept
Boundary (extreme)	Data values that are at a boundary or an extreme end of the range of normal data; test data should include values just within the boundary (that is, valid data) and just outside the boundary (that is, invalid data)

Table 15.07 Types of test data

WORKED EXAMPLE 15.06

Designing test data

Look at the Pyramid Problem (code shown in Section 14.13). This is a simple program, but we can use it to illustrate how to choose test data. There are just two user inputs: the number of symbols that make up the base and the symbol that is to be used to construct the pyramid. Let's consider just the test data for the number of symbols (Table 15.08).

Type of test data	Example test values	Explanation
Normal (valid)	7	7 is an odd integer, so should be accepted. Any odd positive integer would be suitable as test data. However, it should be bigger than 1 to check that the pyramid is correctly formed. More than one different value to test would be a good idea.
Abnormal (erroneous)		Any number that is not a positive odd integer. This will require several tests to ensure that the following types of data are not accepted:

	-7	• negative integer
	8	• even integer
	7.5	• real number
	'*'	• non-numeric input.
		You should not take shortcuts and choose one negative even integer or one negative real number and think you can test two things at the same time. You will not know whether the test fails for just one reason or both.
Boundary (extreme)	1	What is a boundary value? The smallest possible pyramid is a single symbol. So the value 1 is just within the boundary. Sometimes choosing test data throws up some interesting questions that need to be considered when designing the solution:
	0	• Should 0 be accepted? Is 0 an even number? Is it outside the boundary because a pyramid of 0 symbols is not really a pyramid? • Is there just one boundary? Should the program reject numbers that are too large?
	79 81	The output would not look like a pyramid if there were a wrap-around. So the program really should check how many symbols fit onto one line and not allow the user to input a number greater than this. If the number of characters across the screen is 80, then 79 would be just within the boundary but 81 would be outside the boundary, and should not be accepted. Note that by testing with values within the boundary you are also testing normal data, albeit at the extreme ends of the normal range.

Table 15.08 Test data for the pyramid problem

TASK 15.07

Look at the programs you wrote in Chapter 14.

1 Design test data for the number-guessing game (Task 14.09.2).

2 Design test data for the Connect 4 game (Task 14.11).

How to prevent errors

The best way to write a program that works correctly is to prevent errors in the first place. How can we minimise the errors in a program? A major cause of errors is poor requirements analysis. When designing a solution it is very important that we understand the problem and what the user of the system wants or needs. We should use:

- tried and tested design techniques such as structured programming or object-oriented design
- conventions such as identifier tables, data structures and standard algorithms
- tried and tested modules or objects from program libraries.

15.08 Corrective maintenance

Maintaining programs is not like maintaining a mechanical device. It doesn’t need lubricating and parts don’t wear out. **Corrective maintenance** of a program refers to the work required when a program is not working correctly due to a logic error or because of a run-time error. Sometimes program errors don’t become apparent for a long time because it is only under very rare circumstances that there is an unexpected result or the program crashes. These circumstances might arise because part of the program is not used often or because the data on an occasion includes extreme values. Earlier corrective maintenance may also introduce other errors.

When a problem is reported, the programmer needs to find out what is causing the bug. To find a bug, a programmer either uses program debugging software or a trace table (see Section 15.06).

TASK 15.08

1 Design a trace table for the following algorithm:

```
INPUT BinaryString
StringLength ← Length(BinaryString)
FOR i ← 1 TO StringLength
    Bit ← BinaryString[i]
    BitValue ← IntegerValue(Bit) // convert string to integer
    DenaryValue ← DenaryValue + 2 + BitValue
NEXT i
```

2 Dry-run the algorithm using '101' as the input. Complete the trace table.

3 The result should be 5. Can you find the error in the code and correct it?

15.09 Adaptive maintenance

Programs often get changed to make them perform functions they were not originally designed to do.

For example, the Connect 4 game introduced in Chapter 13 (Worked Example 13.03) allows two players, O and X, to play against each other. An amended version would be for one player to be the computer. This would mean a single player could try and win against the computer.

Adaptive maintenance is the action of making amendments to a program to enhance functionality or in response to specification changes.

TASK 15.09

Design the algorithm to simulate the computer playing the part of Player X in Connect 4.

15.10 Perfective maintenance

The program runs satisfactorily. However, there is still room for improvement. For example, the program may run faster if the file handling is changed from sequential access to direct access.

TASK 15.10

Analyse the pseudocode below and make amendments to enhance maintainability.

```
FUNCTION GetPositiveNumber
    DECLARE n : INTEGER
    OUTPUT "Enter a positive number: "
    INPUT n
    RETURN n
ENDFUNCTION
// main program
REPEAT
    Number1 ← GetPositiveNumber
    IF Number1 <= 0
      THEN
        OUTPUT "Not a positive number: "
    ENDIF
UNTIL Number1 > 0
REPEAT
    Number2 ← GetPositiveNumber
    IF Number2 <= 0
      THEN
        OUTPUT "Not a positive number: "
    ENDIF
UNTIL Number2 > 0
```

Reflection Point:

Have you used dry-running for programs you have written? You can check your trace table if you add output statements at key points in your program. You can then compare the program output with the contents of your trace table.

Summary

- The stages of the program development cycle consist of analysis, design, coding, testing and maintenance.
- Structure charts are graphical representations of the modular structure of solutions.
- A structure chart shows the interface between modules: parameters passed between calling module and the module being called.
- Structure charts show selection, where a module is called only under certain conditions.
- Structure charts show repetition, where modules are called repeatedly.
- A state transition diagram is another way of documenting an algorithm.
- Testing strategies include stub testing, black-box testing, white-box testing, integration testing, alpha and beta testing, and acceptance testing.
- Locating and correcting logic errors and run-time errors can be done by dry-running an algorithm or using a trace table.
- Corrective maintenance means fixing bugs that have come to light during use of the program.
- Adaptive maintenance involves altering an algorithm and data structure in response to required

changes.

- **Perfective maintenance** means enhancing performance or maintainability.

Exam-style Questions

1 Consider this code for a function:

```
FUNCTION Binary(Number : INTEGER) RETURNS STRING
    DECLARE BinaryString : STRING
    DECLARE PlaceValue : INTEGER
    BinaryString ← ''  // empty string
    PlaceValue ← 8
    REPEAT
        IF Number >= PlaceValue
          THEN
            BinaryString ← BinaryString & '1' // concatenates two strings
            Number ← Number – PlaceValue
          ELSE
            BinaryString ← BinaryString & '0'
        ENDIF
        PlaceValue ← PlaceValue DIV 2
    UNTIL Number = 0
    RETURN BinaryString
ENDFUNCTION
```

a Dry-run the function call `Binary(11)` by completing the given trace table.

Number	BinaryString	PlaceValue	Number >= PlaceValue
11	''	8	

What is the return value? [5]

b i Now dry-run the function call `Binary(10)` by completing the given trace table.

Number	BinaryString	PlaceValue	Number >= PlaceValue
10	''	8	

What is the return value? [3]

ii The algorithm is supposed to convert a denary integer into the equivalent binary number, stored as a string of 0s and 1s. Explain the result of each dry-run and what needs changing in the given algorithm. [3]

2 A procedure to output a row in a tally chart has been written using pseudocode:

```
PROCEDURE OutputTallyRow(NumberToDraw : INTEGER)
    IF Count > 0
      THEN
        FOR Count ← 1 TO NumberToDraw
            IF (Count MOD 5) = 0
              THEN
                OUTPUT('\') // every 5th bar slants the other way
              ELSE
```

```
                OUTPUT('/')
            ENDIF
        NEXT Count
    ENDIF
    OUTPUT NewLine // move to next row
ENDPROCEDURE
```

Suggest suitable test data that will test the procedure adequately. Justify your choices in each case. [9]

3 A random number generator is to be tested to see whether all numbers within the range 1 to 20 are generated equally frequently. The structured English version of the algorithm is

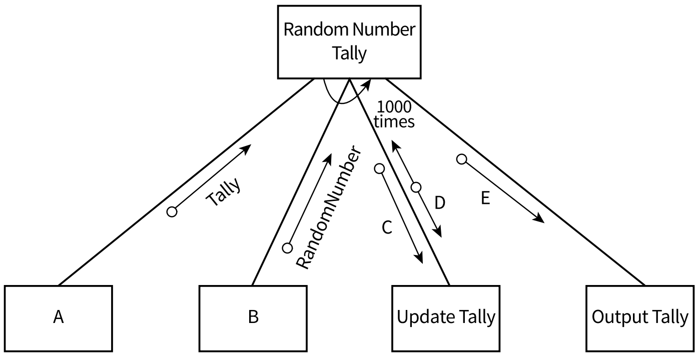
```
Initialise a tally for the numbers 1 to 20
Repeatedly generate numbers in range 1 to 20
For each number generated, increment the relevant count
Calculate how often each number should be generated (expected frequency)
Output expected frequency
Output the list of numbers as a table with actual frequency
```

The identifiers required are:

Identifier	**Data Type**	**Explanation**
`Tally`	`Array[1 : 20] OF INTEGER`	1D array to store the count of how many times each number has been generated
`RandomNumber`	`INTEGER`	The random number generated
`NumberOfTests`	`INTEGER`	The number of times a random number is to be generated (1000 in this example)
`ExpectedFrequency`	`INTEGER`	The number of times any one number would be generated if all numbers are generated equally frequently (1000/20 in this example)

a Complete the structure chart below by naming the labels A to E. [5]

b Develop pseudocode from the structure chart. [12]

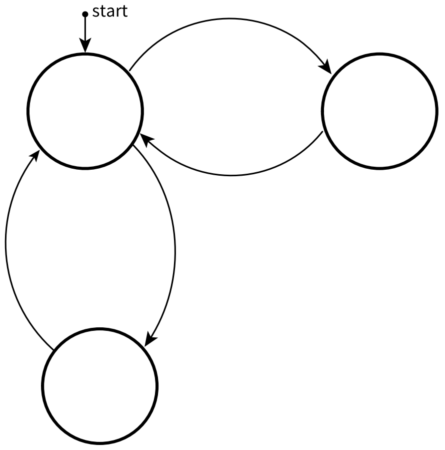


4 A car park has a barrier at the exit. The starting position of the barrier is lowered. When a car wants to exit the car park, the driver has to insert a coin into a coin slot at the barrier. The barrier raises and allows the car to drive out of the car park. After the car has passed through the barrier, the barrier lowers. In case of emergency, a member of staff can open the barrier using a remote control. The barrier will remain open until the remote control is used again to lower the barrier.

The barrier has three states: lowered, raised and open. The transition from one state to another is as shown in the state-transition table:

Current state	**Event**	**Next state**
Barrier lowered	Coin inserted	Barrier raised
Barrier lowered	Open remotely	Barrier open
Barrier open	Close remotely	Barrier lowered
Barrier raised	Car has exited	Barrier lowered

Complete the state-transition diagram for the barrier:



[7]

Part 3
Advanced theory

Chapter 16:
Data representation

Learning objectives

By the end of this chapter you should be able to:

- show understanding of why user-defined types are necessary
- define and use non-composite data types
- define and use composite data types
- choose and design an appropriate user-defined data type for a given problem
- show understanding of the methods of file organisation and select an appropriate method of file organisation and file access for a given problem
- show understanding of methods of file access
- show understanding of hashing algorithms
- describe the format of binary floating-point real numbers
- show understanding of the effects of changing the allocation of bits to mantissa and exponent in a floating-point representation
- convert binary floating-point real numbers into denary and vice versa
- normalise floating-point numbers
- show understanding of the consequences of a binary representation only being an approximation to the real number it represents (in certain cases)
- show understanding that binary representations can give rise to rounding errors.

16.01 Data types

Sections 13.01 to 13.05 of Chapter 13 introduced the concept of a variable being associated with a data type. Before a variable can be used in a program, the variable's data type has to be identified. Chapter 13 introduced the most frequently used data types that are available for association with a variable in a program. This chapter introduces some additional data types that might be used.

Built-in data types

Remember, for each built-in data type:

- the programming language defines the range of possible values that can be assigned to a variable when its type has been chosen.
- the programming language defines the operations that are available for manipulating values assigned to the variable.

User-defined data types

The term 'user' is regularly applied to someone who is provided with a 'user interface' by an operating system – the 'user' is the person supplying input to a running program and receiving output from it.

However, when writing a program, a programmer becomes a 'user' of a programming language. The term **user-defined data type** applies to this latter type of user.

A user-defined data type is a data type for which the programmer has included the definition in the program. Once the data type has been defined, variables can be created and associated with the user-defined data type. Note that, although the user-defined data type is not a built-in data type, using the user-defined data type is only possible if a programming language offers support for the construct.

TIP

Make sure that you do not confuse user-defined data types and abstract data types (defined in Section 13.07 of Chapter 13).

Non-composite data types

A **non-composite data type** is one which has a definition which does not involve a reference to another data type. The simple built-in data types, such as integer or real, are examples of built-in non-composite data types. It is also possible for a user-defined data type to be non-composite.

Enumerated data type

An enumerated data type is an example of a user-defined non-composite data type. When a specific **enumerated data type** is defined, every single possible value for it is identified. The following pseudocode shows two examples of enumerated data type definitions:

```
TYPE
TDirections = (North, East, South, West)
TDays = (Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday)
```

Following these definitions, variables can be declared and assigned values, for example:

```
DECLARE Direction1 : TDirections
DECLARE StartDay : TDays
Direction1 ← North
StartDay ← Wednesday
```

It is important to note the following points.

- The values of the enumerated type look like string values but they are not. The values must not be enclosed in quotes.
- The values defined in an enumerated data type are ordinal. This means that enumerated data types

have an implied order of values.

The ordering can be put to many uses in a program. For example, a comparison statement can be used with the values of the variables of an enumerated data type:

```
DECLARE Weekend : Boolean
DECLARE Day : TDays
Weekend = TRUE IF Day > Friday
```

The enumerated data type is one reason why user-defined data types are sometimes needed. There could not be a built-in generic definition of an enumerated data type because the possible values would not be known. The values can only be known when the programmer has identified them in the type definition.

Composite user-defined data types

A composite user-defined data type has a definition with reference to at least one other type. There are two very important examples of composite user-defined data type.

1. The **record data type** (introduced in Chapter 13). Although there could be built-in record data types the expectation is for a record data type to be user-defined. This allows the programmer to create record data types with components that precisely match the data requirements of the particular program. Note that Python is a language that does not support the use of a record data type.
2. The **class**. A class is a data type which is used for an object in object-oriented programming. For a given object-oriented programming language there are likely to be a number of built-in classes. However, if a programmer intends to utilise the benefits of the object-oriented approach, then the programmer will have to create a number of user-defined classes.

Pointer data type

A **pointer variable** is one for which the value is a reference to a memory location. The following is a commentary on some examples of pseudocode involving the use of the pointer data type.

1. An example of the definition of a pointer type which requires only the identification of a data type for which the pointer is to be used.

   ```
   TYPE

   TIntegerPointer ← ^Integer
   ```

2. An example of the declaration of a variable of the pointer data type which does not require the use of the caret (^) symbol.

   ```
   DECLARE MyIntegerPointer : TIntegerPointer
   ```

3. An example of the declaration of two ordinary variables of type integer and the assignment of a value for one of them.

   ```
   DECLARE Number1, Number2 : INTEGER

   Number1 ← 100
   ```

4. An example of an assignment to a pointer variable of a value which is the address of a different variable.

   ```
   MyIntegerPointer ← @Number1
   ```

5. An example of an assignment which uses the 'dereferenced' value which has been stored at the address defined by the pointer variable. This assigns the value 200 to Number2.

   ```
   Number2 ← MyIntegerPointer^ * 2
   ```

Not all programming languages offer support for the use of a pointer data type. Those languages that do so will have their own version of the symbolism illustrated above with ^ and @.

Because arithmetic can be performed on pointer variables, it is possible to use pointer variables to construct dynamically varying data structures. For some programming languages it is necessary to declare an array with a large upper bound to ensure that the array is unlikely to be fully populated with values. If the language supports the use of a pointer variable, the size of an array can expand while a program is running. The details of how this can be done are beyond the scope of this discussion.

Set data type

A **set** data type allows a program to create sets and to apply the mathematical operations defined in set theory. A set is a mathematical concept with important properties.

- It contains a collection of data values.
- There is no organisation of the data values within the set.
- Duplicate values are not allowed.
- Operations that can be performed on a set include:
    - checking if a value exists in a set
    - adding a new data value
    - removing an existing data value
    - adding one set to another set.

A set variable can be created if a programming language supports the set data type. It is difficult to classify the set data type. Because the set contains multiple components it is tempting to say that the set is a structured data type. However, this contradicts the fact that the set has no structure and therefore no indexing can be associated with the members of the set.

The most useful property of a set is the fact that duplicate values are not allowed. A list or a one-dimensional array might be created but has to be checked to remove duplicate values. A simple way of removing duplicate values would be to convert the structure to a set and then convert the set back to the original structure.

A slightly different example would be if students were allocated to groups for studying a particular subject. For each subject, the students' names would be entered into a data structure defined for that subject. Set data types could then, for example, find out which students were studying both computer science and physics. The students studying both subjects would be found by applying the 'intersection' operation on the two individual sets.

16.02 File organisation

In everyday computer usage, a wide variety of file types is encountered. Examples are graphic files, word-processing files, spreadsheet files and so on. Whatever the file type, the content is stored using a specific binary code that allows the file to be used as intended.

For the specific task of storing data to be used for input to a computer program or for output from a computer program, there are only two defined file types. A file is either a text file or a **binary file**.

A text file, as discussed in Chapter 13 (Section 13.06) contains data stored according to a character code of the type described in Chapter 1 (Section 1.04). It is possible, by using a text editor, to create a text file to be used as input to a program.

A program may create a binary file as output with the intention of subsequently using it for input. A binary file stores data in its internal representation, for example an integer value might be stored in two bytes in two’s complement representation.

The organisation of a binary file is based on the concept of a **record**. A file contains records and each record contains fields. Each field consists of a value. For a text file the number of data items per line must be known and the number of characters per item must be known. If these are not known then item separator characters must be used. The file has repeating lines which are defined by an end-of-line character or characters.

For a binary file the number of fields per record must be known. If any of the fields represent a string, the length of the string must be known. For any other field the internal representation will define the number of bytes required to store the field value. There is no need for field separator characters or for an end-of-record character.

Discussion Point:

A record is a user-defined data type. It is also a component of a file. Can there be or should there be any relationship between these two concepts?

Serial files

A serial file contains records that have not been organised in any defined order. A typical use of a serial file would be for a bank to record transactions involving customer accounts. A program would be running. Each time there was a withdrawal or a deposit the program would receive the details as data input and would record the data in a transaction file. In a serial file each new record is simply appended to the file so that the only ordering in the file is the time order of data entry.

Sequential files

A sequential file has records that are ordered. In the bank example, a sequential file could be used as a master file for an individual customer account. At regular periods of time, the transaction file would be read, and all affected customer account master files would be updated. In order to allow a sequential file to be ordered, there has to be a key field for which the values are unique and sequential but not necessarily consecutive. When a new record is to be added to a sequential file it would be possible to simply append the record, with the intention of sorting the file later. A more likely approach is for the file to be read sequentially and each record written to a new file. This is continued until the appropriate position for the new record is reached. The new record is then written to the new file before the remaining records in the old file are copied in.

Extension Question 16.01

Can you think of reasons why you might want to use binary files with variable-length records? How would you make sure a binary file with variable-length records would be read correctly?

Direct-access files

Direct-access files are sometimes referred to as ‘random-access’ files but, as with random-access memory, the randomness is only that the access can be to any record in the file without sequential reading of the file. Direct access can be achieved with a sequential file. A separate index file is created which has two fields per record. The first field has the key field value and the second field has a value for the position of this key field value in the main file.

The alternative is to use a hashing algorithm when a record is entered into the direct-access file.

One simple hashing algorithm is applicable if there is a numeric key field in each record. The algorithm chooses a suitable number and divides this number by the value in the key field. The remainder from this division then identifies the address in the file for storage of that record. The suitable number works best if it is a prime number of a similar size to the expected size of the file.

For simplicity this can be illustrated for 4-digit values in the key field where 1000 is used for the dividing number. The following represent three calculations:

0045/1000 gives remainder 45 for the address in the file

2005/1000 gives remainder 5 for the address in the file

3005/1000 gives remainder 5 for the address in the file

There are two facts apparent from these calculations. The first fact is that the addresses calculated do not have any order depending on the value in the key field. The second fact is that different key field values can produce the same remainder and therefore the same address in the file.

If the records do not have a suitable field with numeric digits, an alternative is to choose a field with some alphabetic characters. The ASCII code for each character can be looked up and the values then added. The sum is then used in the same way as described above, to calculate an address as the remainder from an integer division.

When the same address is calculated for different field values, it is usually referred to as a collision (the addresses are sometimes called synonyms). The best choice for a hashing algorithm is one that spreads the addresses most evenly and minimises the number of collisions. However, collisions cannot be avoided altogether so there has to be a defined method for dealing with collisons. There are a number of options, including the following:

- use a sequential search to look for a vacant address following the calculated one
- keep a number of overflow addresses at the end of the file
- have a linked list accessible from each address.

Question 16.01

Imagine the possible numeric values for a key field in a direct-access file are in the range of 1 to 30 but you want the file to have fewer than 30 file addresses.

You decide to test two examples of a modular division hashing algorithm. The first test uses 10 as the number for division, the second test uses 11.

a What are the two sets of addresses generated as remainders from the division for the key values 0 to 39 using 10 and 11?

b State one difference between the two sets of addresses.

c Is there any significant difference between the two sets of addresses?

d 11 is a prime number. Prime numbers are stated to give a better spread of use of the addresses in a file. Do you know when this is more likely to be true?

File access

Once a file organisation has been chosen and the data has been entered into a file, you need to consider how this data is to be accessed. For a serial file, the normal usage is to read the whole file record by record. If there was a need to search for a particular value in one of the fields, the only option would be to read the records from the beginning until the target record was found. If the data is stored in a sequential file and a particular value is needed, searching may have to be done in the same way. However, if the key field value is known for the record containing the wanted data, the process is faster because only key field values need to be read. For a direct-access file, the value in the key field is submitted to the hashing algorithm. The value is the same value that was used when entering the data originally and will provide the same value for the position in the file that was provided when the algorithm was used at the time of data input. This eliminates the need to read records from the beginning of the file. However, because of the collision problem some serial searching might be needed after the initial jump to the hashed position.

File access might also be needed to delete or edit data. For a sequential file the same method is used as when a new record was added. Records are copied from the old file to a new file until the record that needs to be deleted or edited is reached. Following deletion or editing all remaining records are copied to the new file.

For a direct-access file there is no need to create a new file. If a record needs editing it can be accessed directly and edited without disturbing any other content. However, if a record is to be deleted it is necessary to have a flag set in the record. Then, in a subsequent reading process, that record is skipped over.

Choice of file organisation

Serial file organisation is well suited to batch processing or for backing up data on magnetic tape. A direct access file is used if rapid access to an individual record in a large file is required. An example would be on a system with many users. In this case, the file that is used to check passwords when users log in should be direct-access. A sequential file is suitable for applications when multiple records are required from one search of the file. An example could be a family history file where a search could be used for all records with a particular family name.

At this point it is worth mentioning the difference between key fields in a file and primary keys in a database table. In the database table the primary key values must all be unique. This is not a requirement for key fields in any type of file. It may be sensible in certain applications to ensure key fields have unique values, but it is not mandatory.

16.03 Real numbers

A real number is one with a fractional part. When we write down a value for a real number in the denary system we have a choice. We can use a simple representation, or we can use an exponential notation (sometimes referred to as scientific notation). For example, the number 25.3 might be written as:

$$.253 \times 10^{2} \text{ or } 2.53 \times 10^{1} \text{ or } 25.3 \times 10^{0} \text{ or } 253 \times 10^{-1}$$

For this number, the simple expression is best. But if a number is very large or very small the exponential notation is the only sensible choice.

Floating-point and fixed-point representations

A binary code must be used for storing a real number in a computer system. One possibility is to use a fixed-point representation. In fixed-point representation, an overall number of bits is chosen with a defined number of bits for the whole number part and the remainder for the fractional part. The alternative is a **floating-point representation**. The format for a floating-point number can be generalised as:

$$\pm M \times R^{E}$$

In floating-point representation a defined number of bits are used for what is called the significand or mantissa, ±M. The remaining bits are used for the exponent or exrad, E. The radix, R is not stored in the representation; R has an implied value of 2.

A simple example can be used to illustrate the differences between the two representations. Let’s consider that a real number is to be stored in eight bits.

For the fixed-point option, a possible choice would be to use the most significant bit as a sign bit and the next five bits for the whole number part. This would leave two bits for the fractional part. Some important non-zero values in this representation are shown in Table 16.01. (The bits are shown with a gap to indicate the implied position of the binary point.)

Description	Binary code	Denary equivalent
Largest positive value	011111 11	31.75
Smallest positive value	000000 01	0.25
Smallest magnitude negative value	100000 01	−0.25
Largest magnitude negative value	111111 11	−31.75

Table 16.01 Example fixed-point representations (using sign and magnitude)

A possible choice for a floating-point representation would be four bits for the mantissa and four bits for the exponent with each using two’s complement representation. The exponent is stored as a signed integer. The mantissa has to be stored as a fixed-point real value. The question now is where the binary point should be.

Two of the options for the mantissa being expressed in four bits are shown in Table 16.02(a) and Table 16.02(b). In each case, the denary equivalent is shown, and the position of the implied binary point is shown by a gap. Table 16.02(c) shows the three largest magnitude positive and negative values for integer coding that will be used for the exponent.

a

First bit pattern for a real value	Real value in denary
011 1	3.5

b

Second bit pattern for a real value	Real value in denary
0 111	0.875

c

Integer bit pattern	Integer value in denary
0111	7
0110	6

011 0	3.0	0 110	0.75	0101	5
010 1	2.5	0 101	0.625	1010	–6
101 0	–3.0	1 010	–0.75	1001	–7
100 1	–3.5	1 001	–0.875	1000	–8
100 0	–4.0	1 000	–1.0		

Table 16.02 Coding a floating-point real value in eight bits (four for the mantissa and four for the exponent)

When the mantissa has the implied binary point immediately following the sign bit, a smaller spacing is produced between the values that can be represented. This is the preferred option for a floating-point representation. Using this option, the most important non-zero values for the floating-point representation are shown in Table 16.03. (The implied binary point and the mantissa exponent separation are shown by a gap.)

Description	Binary code	Denary equivalent
Largest positive value	0 111 0111	$0.875 \times 2^{7} = 112$
Smallest positive value	0 001 1000	$0.125 \times 2^{-8} = 1/2048$
Smallest magnitude negative value	1 111 1000	$-0.125 \times 2^{-8} = -1/2048$
Largest magnitude negative value	1 000 0111	$-1 \times 2^{7} = -128$

Table 16.03 Example floating-point representations

The comparison between the values in Tables 16.01 and 16.03 illustrate the greater range of positive and negative values available if floating-point representation is used.

Extension Question 16.02

a Using the methods suggested in Chapter 1 (Section 1.01) can you confirm for yourself that the denary equivalents of the binary codes shown in Tables 16.02 and Table 16.03 are as indicated?

b Can you also confirm that conversion from positive to negative (or the conversion from negative to positive) for a fixed-format real value still follows the rules defined in Chapter 1 (Section 1.02) for two's complement representation?

Precision and normalisation

You have to decide about the format of a floating-point representation in two respects. You have to decide the total number of bits to be used and decide on the split between those representing the mantissa and those representing the exponent. In practice, a choice for the total number of bits to be used will be available as an option when the program is written. However, the split between the two parts of the representation will have been determined by the floating-point processor. If you did have a choice you would base your decision on the fact that increasing the number of bits for the mantissa would give better precision for a value stored but would leave fewer bits for the exponent, which reduces the range of possible values.

To achieve maximum precision you have to normalise a floating-point number. (This normalisation is unrelated to the process associated with designing a database.) Precision increases with an increasing number of bits for the mantissa, so optimum precision will only be achieved if full use is made of these bits. In practice, that means using the largest possible magnitude for the value represented by the mantissa.

To illustrate this, we can consider the eight-bit representation used in Table 16.03. Table 16.04 shows possible representations for denary 2 using this representation.

Denary representation	Floating-point binary representation

0.125×2^4	0 001 0100
0.25×2^3	0 010 0011
0.5×2^2	0 100 0010

Table 16.04 Alternative representations of denary 2 using four bits each for mantissa and exponent

For a negative number we can consider representations for –4 as shown in Table 16.05.

Denary representation	Floating-point binary representation
-0.25×2^4	1 110 0100
-0.5×2^3	1 100 0011
-1.0×2^2	1 000 0010

Table 16.05 Alternative representations of denary −4 using four bits each for mantissa and exponent

When the number is represented with the highest magnitude for the mantissa, the two most significant bits are different. This fact can be used to recognise that a number is in a normalised representation. The values in Tables 16.03 and 16.04 also show how a number could be normalised. For a positive number, the bits in the mantissa are shifted left until the most significant bits are 0 followed by 1. For each shift left the value of the exponent is reduced by 1.

The same process of shifting is used for a negative number until the most significant bits are 1 followed by 0. In this case, no attention is paid to the fact that bits are falling off the most significant end of the mantissa.

Extension Question 16.03

Look at the data in Table 16.03. Do you see any conflict with the above discussion? What is likely to be the approach in a typical floating-point system?

Conversion of representations

In Chapter 1 (Section 1.01), a number of methods for converting numbers into different representations were discussed. These only considered integer values. We now need to consider the conversion of real numbers.

We can start by considering the conversion of a simple real number, such as 4.75, into a simple fixed-point binary representation. This looks easy because 4 converts to 100 in binary and .75 converts to .11 in binary so the binary version of 4.75 should be:

100.11

However, remember that a positive number should start with 0. Can we just add a sign bit? For a positive number we can. Denary 4.75 can be represented as 0100.11 in binary.

For negative numbers we still want to use two’s complement form. So, to find the representation of −4.75 we can start with the representation for 4.75 then convert it to two’s complement as follows:

0100.11 converts to 1011.00 in one’s complement

then to 1011.01 in two’s complement

To check the result, we can apply Method 2 from Worked Example 1.01 in Chapter 1. 1011 is the code for −8 + 3 and .01 is the code for .25; −8 + 3 + .25 = −4.75.

We can now consider the conversion of a denary value expressed as a real number into a floating-point binary representation. Before considering the conversion method it should be remembered that most fractional parts do not convert to a precise representation. This is because the binary fractional parts represent a half, a quarter, an eighth, a sixteenth and so on. Unless a denary fraction is a sum of a collection of these values, there cannot be an accurate conversion. In particular, of the values from .1 through to .9, only .5 converts accurately. This was mentioned in Chapter 1 (Section 1.03) in the discussion about storing currency values.

The method for conversion of a positive value is as follows.

1 Convert the whole-number part using the method described in Chapter 1 (Section 1.01).
2 Add the 0 sign bit.
3 Convert the fractional part choosing a method from one of the examples in Worked Example 16.01.
4 Combine the whole number and fractional parts and enter these into the most significant of the bits allocated for the representation of the mantissa.
5 Fill the remaining bits for the mantissa and the bits for the exponent with zeros.
6 Adjust the position of the binary point by changing the exponent value to achieve a normalised representation.

To convert a negative value the number is treated initially as positive and the same first five steps are followed. At this stage a two's complement conversion of the mantissa code is used to convert this to a negative value before step 6 is carried out.

WORKED EXAMPLE 16.01

Converting a denary value to a floating-point representation

Example 1

Let's consider the conversion of 8.75.

1 The 8 converts to 1000, adding the sign bit gives 01000.
2 The .75 can be recognised as being .11 in binary.
3 The combination gives 01000.11 which has exponent value zero.
4 Shifting the binary point gives 0.100011 which has exponent value denary 4.
5 The next stage depends on the number of bits defined for the mantissa and the exponent; if ten bits are allocated for the mantissa and four bits are allocated for the exponent the final representation becomes 0100011000 for the mantissa and 0100 for the exponent.

Example 2

Let's consider the conversion of 8.63. The first step is the same but now the .63 has to be converted by the 'multiply by two and record whole number parts' method. This works as follows:

.63 × 2 = 1.26 so 1 is stored to give the fraction .1
.26 × 2 = .52 so 0 is stored to give the fraction .10
.52 × 2 = 1.04 so 1 is stored to give the fraction .101
.04 × 2 = .08 so 0 is stored to give the fraction .1010

At this stage it can be seen that, multiplying .08 by 2 successively is going to give a lot of zeros in the binary fraction before another 1 is added so the process can be stopped. .63 has been approximated as .625. So, following Steps 3–5 in Example 1, the final representation becomes 0100010100 for the mantissa and 0100 for the exponent.

TASK 16.01

Convert the denary value –7.75 to a floating-point binary representation with ten bits for the mantissa and four bits for the exponent. Start by converting 7.75 to binary (make sure you add the sign bit!). Then convert to two's complement form. Finally, choose the correct value for the exponent to leave the implied position of the binary point after the sign bit. Convert back to denary to check the result.

Problems with using floating-point numbers

As illustrated above, the conversion of a real value in denary to a binary representation almost guarantees a degree of approximation. There is also a restriction of the number of bits used to store the mantissa.

Floating-point numbers are used in extended mathematical procedures involving repeated calculations. For example, in weather forecasting using a mathematical model of the atmosphere, or in economic forecasting. In such programming there is a slight approximation in recording the result of each calculation. These so-called rounding errors can become significant if calculations are repeated enough times. The only way of preventing the errors becoming a serious problem is to increase the precision of the floating-point representation by using more bits for the mantissa. Programming languages therefore offer options to work in ‘double precision’ or ‘quadruple precision’.

The other potential problem relates to the range of numbers that can be stored. Referring back to the simple eight-bit representation illustrated in Table 16.03, the highest value represented is denary 112. A calculation can easily produce a value higher than this. As Chapter 1 (Section 1.02) illustrated, this produces an overflow error condition. However, for floating-point values there is also a possibility that if a very small number is divided by a number greater than 1 the result is a value smaller than the smallest that can be stored. This is an underflow error condition. Depending on the circumstances, it may be possible for a program to continue running by converting this very small number to zero but this must involve risk.

Summary

- Examples of non-composite user-defined data types include enumerated and pointer data types.
- Record, set and class are examples of composite user-defined data types.
- File organisation allows for serial, sequential or direct access.
- Floating-point representation for a real number allows a wider range of values to be represented.
- A normalised floating-point representation achieves the best precision for the value stored.
- Stored floating-point values rarely give an accurate representation of the denary equivalent.

Reflection Point:

Whenever you are asked to create a binary representation from a denary value or vice-versa are you always checking your answer by converting it back to the original value?

Exam-style Questions

1 A programmer may choose to use a user-defined data type when writing a program.

a Give an example of a non-composite user-defined data type and explain why its use by a programmer is different to the use of an in-built data type. [3]

b A program is to be written to handle data relating to the animals kept in a zoo.

The programmer chooses to use a record user-defined data type.

i Explain what a record user-defined data type is. [2]

ii Explain the advantage of using a record user-defined data type. [2]

iii Write pseudocode for the definition of a record type which is to be used to store: animal name, animal age, number in zoo and location in the zoo. [5]

2 a A binary file is to be used to store data for a program.

i State the terms used to describe the components of such a file? [2]

ii Explain the difference between a binary file and a text file. [3]

b A binary file might be organised for serial, sequential or direct access.

i Explain the difference between the three types of file organisation. [4]

ii Give an example of file use for which a serial file organisation would be suitable.

Justify your choice. [3]

iii Give an example of file use when direct access would be advantageous.

Justify your choice. [3]

3 A file contains binary coding. The following are four successive bytes in the file:

10010101		00110011		11001000		00010001

a The four bytes represent two numbers in floating-point representation. The first byte in each case represents the mantissa. Each byte is stored in two's complement representation.

i Give the name for what the second byte represents in each case. [1]

ii State whether the representations are for two positive numbers or two negative numbers and explain why. [2]

iii One of the numbers is in a normalised representation. State which one it is and give the reason why. [2]

iv State where the implied binary point is in a normalised representation and explain why a normalised representation gives better precision for the value represented. [3]

v If two bytes were still to be used but the number of bits for each component was going to be changed by allocating more to the mantissa, state what effect this would have on the numbers that could be represented. Explain your answer. [2]

b Using the representation described in **part a**, give the representation of denary 12.43 as a floating-point binary number. [5]

4 a A particular programming language allows the programmer to define their own data types.

`ThisDate` is an example of a user-defined structured data type.

```
TYPE ThisDate
   DECLARE ThisDay      :    (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13,
                             14, 15, 16, 17, 18, 19, 20, 21, 22, 23,
                             24, 25, 26, 27, 28, 29, 30, 31)
   DECLARE ThisMonth    :    (Jan, Feb, Mar, Apr, May, Jun, Jul, Aug,
                              Sep, Oct, Nov, Dec)
   DECLARE ThisYear     :    INTEGER
ENDTYPE
```

A variable of this new type is declared as follows:

```
DECLARE DateOfBirth   :   ThisDate
```

i Name the non-composite data type used in the `ThisDay` and `ThisMonth` declarations. [1]

ii Name the data type of `ThisDate`. [1]

iii The month value of `DateOfBirth` needs to be assigned to the variable `MyMonthOfBirth`.

Write the required statement. [1]

b Annual rainfall data from a number of locations are to be processed in a program.

The following data are to be stored:

- location name
- height above sea level (to the nearest metre)
- total rainfall for each month of the year (centimetres to 1 decimal place).

A user-defined, composite data type is needed. The programmer chooses `LocationRainfall` as the name of this data type.

A variable of this type can be used to store all the data for one particular location.

i Write the definition for the data type `LocationRainfall`. [5]

ii The programmer decides to store all the data in a file. Initially, data from 27 locations will be stored. More rainfall locations will be added over time and will never exceed 100.

The programmer has to choose between two types of file organisation. The two types are serial and sequential.

Give **two** reasons for choosing serial file organisation. [2]

Cambridge International AS & A level Computer Science 9608 paper 31 Q3 June 2015

5 In a particular computer system, real numbers are stored using floating-point representation with:

- 8 bits for the mantissa
- 8 bits for the exponent
- two's complement form for both mantissa and exponent.

a Calculate the floating point representation of +3.5 in this system. Show your working.

Mantissa

●							

Exponent

[3]

b Calculate the floating point representation of –3.5 in this system. Show your working.

Mantissa

●							

Exponent

c Find the denary value for the following binary floating-point number. Show your working.

Mantissa

0 ●	1	1	1	0	0	0	0

Exponent

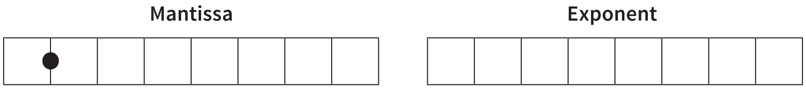
0	0	0	0	0	1	0	0

[3]

d **i** State whether the floating-point number given in **part (c)** is normalised or not normalised. [1]

ii Justify your answer given in **part (d)(i)**. [1]

e Give the binary two's complement pattern for the negative number with the largest magnitude.



[2]

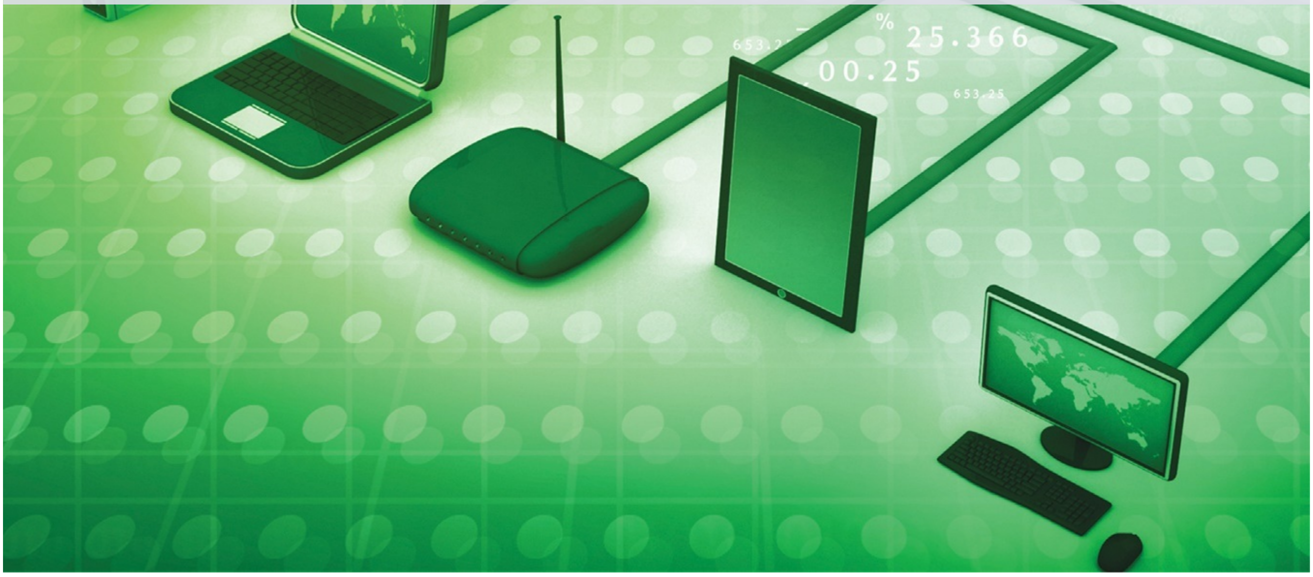
Cambridge international AS & A Level Computer Science 9608 paper 32 Q1 November 2016

Chapter 17: Communication and Internet technologies

Learning objectives

By the end of this chapter you should be able to:

- show understanding of circuit switching
- show understanding of packet switching
- show understanding of why a protocol is essential for communication between computers
- show understanding of how protocol implementation can be viewed as a stack, where each layer has its own functionality
- show understanding of the TCP/IP protocol suite
- show understanding of protocols (HTTP, FTP, POP3, IMAP, SMTP, BitTorrent) and their purposes.



17.01 Transmission modes

For communication over an internetwork there are two possible approaches: circuit switching or packet switching.

Circuit switching

Circuit switching is the method used in the traditional telephone system. Because the Public Switched Telephone Networks (PTSNs) have now largely converted to digital technology, the same method can be provided for data transfer that was traditionally used for voice communication. In Chapter 2 (Section 2.06) the concept of support for Internet usage by a PTSN was introduced. Typically, this is provided in a leased line service. The concept is illustrated in Figure 17.01, which shows end-systems connected to local exchanges which have a switching function and which are connected via a number of intermediate nodes with a switching function.

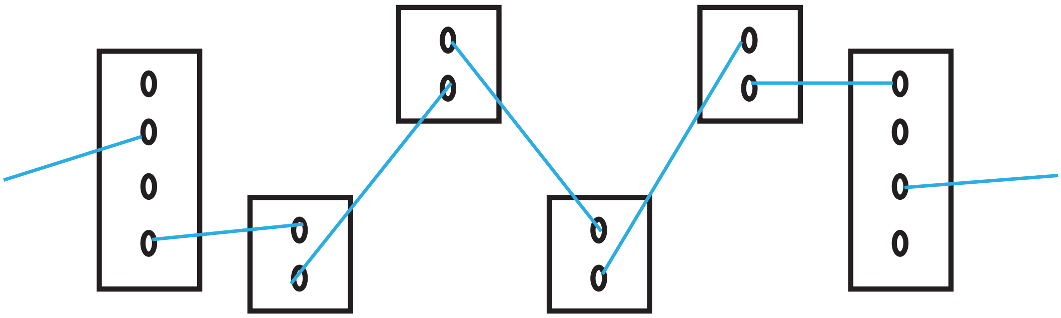


Figure 17.01 Circuit-switched data transmission

For data transfer to take place, the following has to happen.

1. The sender provides the identity of the intended receiver.
2. The system checks whether or not the receiver is ready to accept data.
3. If the receiver is available, a sequence of links is established across the network.
4. The data is transferred.
5. The links are removed.

It is not necessary for this discussion to define what could constitute a node in a circuit-switched network. The links that are provided between the nodes are dedicated channels in shared transmission media that guarantee unimpeded transmission. When a telephone call has finished there is a definite end to the call with removal of the links. However, for a leased-line data connection there might be a permanent circuit established.

Packet switching

The packet-switching method allows data transmission without a circuit being established. Data cannot be sent in a continuous stream. Instead data is packaged in portions inside packets. A packet consists of a header which contains instructions for delivery plus the data body. The method is similar to that used by the postal service but rather more complex! The network schematic shown in Figure 17.01 is still appropriate to describe packet switching except that the links used are not defined at the time a packet is transmitted by the sender. Furthermore, the nodes will have an extended functionality compared to that required in a circuit-switched transmission. How a router acts as a node and supports packet switching is discussed in Section 17.04.

When packet switching is used, there are two ways that the network can provide a service: connectionless service or connection-oriented service. If a connectionless service is provided, a packet is dispatched with no knowledge of whether or not the receiver is ready to accept the packet, and has no way of finding out if the transmission has succeeded. In a connection-oriented service the first packet sent includes a request for an acknowledgement. If the acknowledgement is received, the sender transmits further packets. If no acknowledgement is received, the sender tries again with the first packet.

17.02 Protocols

The basic definition of a **protocol** is simple – it is a set of rules. So, what do these rules relate to? Before answering, it should be understood that we often talk about ‘a protocol’ when we are referring to a protocol suite which contains more than one individual protocol. The complexity of networking requires many individual protocols. A further complication is that there might be a number of different versions of a protocol. Finally, there are often protocols that can be used to complement the use of another protocol.

The set of rules that constitute a protocol must be agreed between the sender and the receiver for any communication transmitted over a network. At the simplest level, a protocol could define that a positive voltage represents a bit with value 1. A protocol might define a transmission speed that a sender must not exceed. Many of the rules relate to the format of a message or of a component of a message. For example, a definition of the format of the first 40 bytes in a packet.

17.03 A protocol stack

For a protocol suite the protocols can be viewed as layers within a protocol stack. There are a number of aspects relating to this concept.

- Each layer can only accept input from the next higher layer or the next lower layer.
- There is a defined interface between adjacent layers which constitutes the only interaction allowed between layers.
- A layer is serviced by the actions of lower layers.
- With the possible exception of the lowest layer the functioning of a layer is created by installed software.
- A layer may comprise sub-layers.
- Any user interaction will take place using protocols associated with the highest level layer in the stack.
- Any direct access to hardware is confined to the lowest layer in the stack.

17.04 The TCP/IP protocol suite

TCP/IP is the protocol suite underpinning Internet usage. TCP/IP can be explained on the basis of the network model shown in Figure 17.02. It can be seen that TCP/IP only occupies the top three layers of this model.

TIP

Unfortunately there are different names used for two of the layers. Figure 17.02 shows the Network layer and the Data link layer. In some sources these will be named as the Internet layer and the Link layer.

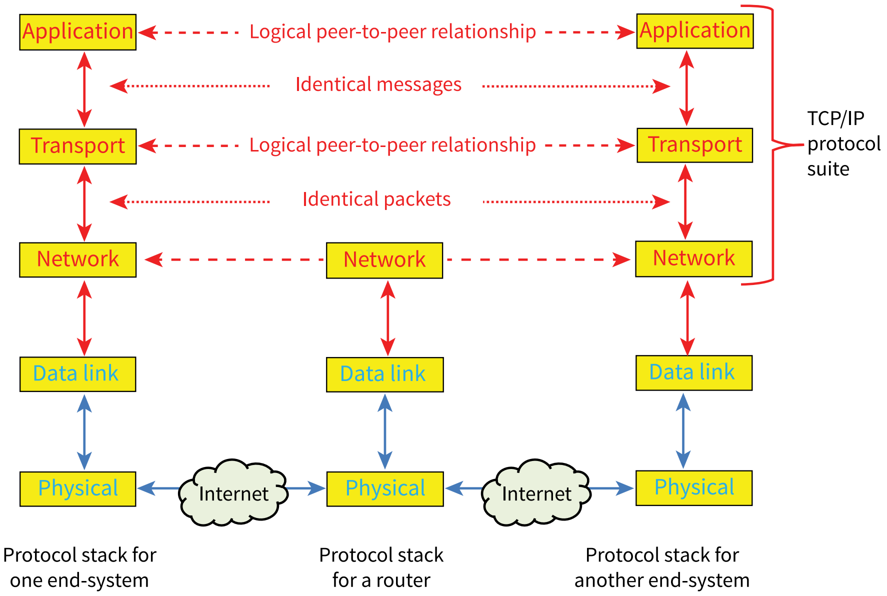


Figure 17.02 A network model relating to the TCP/IP protocol suite

There are two end-systems and the diagram shows the logical connections between corresponding layers in these two systems. An application can run on one end-system and behave as though there was a direct connection with an application running on a different end-system. The application layer protocol on the sender end-system sends a 'message' to the transport layer protocol on the same system. The transport layer protocol then initiates a process which results in the identical 'message' being delivered to the receiver end-system. On the receiver end-system, the final stage is the transport layer protocol delivering the 'message' to the application layer protocol.

The TCP/IP suite comprises a number of protocols, including the following:

- application layer: HTTP, SMTP, DNS, FTP, POP3, IMAP
- transport layer: TCP, UDP, SCTP
- network layer: IP, IGMP, ICMP, ARP.

The selection has been chosen to illustrate that the TCP/IP suite encompasses a very wide range of protocols that is still evolving. Some of the listed protocols will not be considered further.

TCP (Transmission Control Protocol)

If an application is running on an end-system where a 'message' is to be sent to a different end-system the application will be controlled by an application layer protocol as described above. The protocol will transmit the user data to the transport layer. The TCP protocol operating in the transport layer now has to take responsibility for ensuring the safe delivery of the 'message' to the receiver. The TCP protocol creates sufficient packets to hold all of the data. Each packet consists of a header plus the user data.

Question 17.01

Can you find out some details about the format of a packet that TCP would send to IP?

As well as ensuring safe delivery, TCP has to ensure that any response is directed back to the application protocol. So, one item in the header is the port number which identifies the application layer protocol. For example, for HTTP the port number is 80. The packet must also include the port number for the application layer protocol at the receiving end-system. However, TCP is not concerned with the address of the receiving end-system. If the packet is one of a sequence, a sequence number is included to ensure eventual correct reassembly of the user data.

The TCP protocol is connection-oriented. In accordance with the procedure described in Section 17.01, initially just one packet of a sequence is sent to the network layer. Once the network layer returns an acknowledgement to the Transport layer indicating that the connection has been established, TCP sends the other packets and receives response packets containing acknowledgements. This allows missing packets to be identified and re-sent.

IP (Internet Protocol)

The function of the network layer, and in particular of the IP, is to ensure correct routing over the Internet. To do this the IP protocol takes the packet received from the transport layer and adds a further header. The header contains the IP addresses of both the sender and the receiver. To find the IP address of the receiver, it is very likely to use the DNS service to find the address corresponding to the URL supplied in the user data. The DNS service is discussed in some detail in Chapter 2 (Section 2.09).

The IP packet, which is usually called a ‘datagram’, is sent to the data-link layer and therefore to a different protocol suite. The data-link layer assembles datagrams into ‘frames’ as discussed in the next section of this chapter. Once the IP packet has been sent to the data-link layer, IP has no further duty. IP functions as a connectionless service. Once a packet has been sent, IP will have no knowledge of whether or not it reached its destination. If IP receives a packet which contains an acknowledgement of a previously sent packet, it will simply pass the packet on to TCP with no awareness of the content.

The router

As Figure 17.02 shows, the frame sent by the data-link layer will arrive at a router during transmission (more likely at several routers). At this stage, the datagram content of the frame is given back to IP. It is now the function of the router software to choose the next target host in the transmission. The software has access to a routing table appropriate to that router. The size and complexity of the Internet prohibits a router from having a global routing table. Once the appropriate address has been inserted into the datagram, IP passes it back to the data-link layer of the router.

The routing table for every router has details of any current problems with any of the options for the next transmission step. This ensures that packets are delivered to their destination in the shortest possible time available.

The major distinction between a switch and a router as a node in a network is that when a frame arrives at a switch, it is transmitted on without any routing decision. A switch operates in the data-link layer but has no access to the network layer.

17.05 The Ethernet protocol stack

As discussed in Chapter 2 (Section 2.05), Ethernet is a protocol suite designed for use in a local area network (LAN). As such it can function in an isolated LAN with no connection to the Internet or any other network. However, it is now almost inevitable that a LAN will be connected to the Internet and, therefore, a LAN’s protocol suite will support the protocol suite in use for the Internet.

If we look at the protocol stack for one end-system, as shown in Figure 17.02, we can see that the TCP/IP protocol suite occupies the top three layers of the five-layer stack and is therefore supported by the lower two layers. TCP/IP has no concern with the functioning of these two lower layers; it is designed to be capable of being supported by whatever protocols are available. It should be noted that some sources only use a four-layer stack for an end-system. This is either a decision to only include layers that are handled entirely by software. Or it is a decision to amalgamate all of the support for TCP/IP into one layer.

Ethernet is the most likely protocol to be used to provide the functionality required of the two lower layers. Logically the Ethernet suite can be viewed as comprising two sub-layers for each of the Data link and Physical layers. This is illustrated in Figure 17.03.

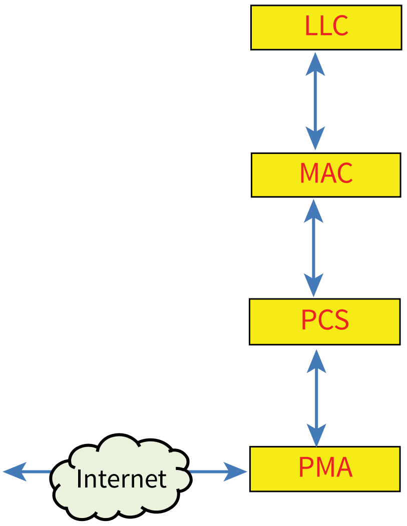


Figure 17.03 The sub-layers of Ethernet

The following points explain how Ethernet functions when supporting TCP/IP.

- The Logical Link Control (LCC) protocol is responsible for the interaction with the Network layer. It manages data transmissions and ensures the integrity of data transmissions. However, because Ethernet is a connectionless protocol it has no responsibility for checking that a transmission has been successfully delivered.
- The Medium Access Control (MAC) protocol is responsible for assembling the Ethernet packet which is referred to as a frame. Two components of this are the address of the transmitter and the address of the receiver. In addition the MAC protocol initiates frame transmission and handles recovery from transmission failure due to a collision (possibly using CSMA/CD).
- The Physical Coding Sublayer (PCS) protocol is responsible for coding data ready for transmission or decoding data received. It either receives a frame from the MAC protocol or sends one to it.
- The Physical Medium Attachment (PMA) protocol is responsible for signal transmitting and receiving.

MAC addresses

Both addresses in the Ethernet frame are examples of what are called physical or MAC addresses. A MAC address is the one which uniquely defines one NIC, as described in Chapter 2 (Section 2.04).

The reason that unique addresses have so far been guaranteed is that the 48 bits used for the definition have given a sufficient number of different addresses. However, there is a 64-bit alternative which is already used occasionally but is available for future use when 48 bits are insufficient. The 48-bit address is usually written in hexadecimal notation, for example:

4A:30:12:24:1A:10

In one version of this addressing scheme the first three bytes identify the manufacturer.

17.06 Application-layer protocols associated with TCP/IP

There are very many application-layer protocols. This discussion considers some that are used most often.

HTTP (HyperText Transfer Protocol)

TIP

Be careful not to confuse HTTP and HTML.

Because HTTP (HyperText Transfer Protocol) underpins the World Wide Web it has to be considered to be the most important application-layer protocol. Every time a user accesses a website using a browser, HTTP is used but its functionality is hidden from view.

HTTP is a transaction-oriented, client–server protocol. The transaction involves the client sending a ‘request’ message and the server sending back a ‘response’ message. The HTTP protocol defines the format of the message. The first line of a request message is the ‘request line’. Optionally this can be followed by header lines. All of this uses ASCII coding. The format of the request line is:

```
<Method> <URL> <Version>CRLF
```

where CR and LF are the ASCII carriage return and line feed characters. The request line usually has GET as the method. However, there are several alternatives to the GET method which makes HTTP potentially a more widely applicable protocol than just being used for webpage access. The version has to be specified because HTTP has evolved so there is more than one version in use.

Email protocols

The traditional method of sending and receiving emails is illustrated in Figure 17.04 which shows that three, individual, client–server interactions are involved. The email sender acting as a client has a connection to a mail server. This server then has to function as a client in the transmission to the mail server used by the email receiver acting as a client.

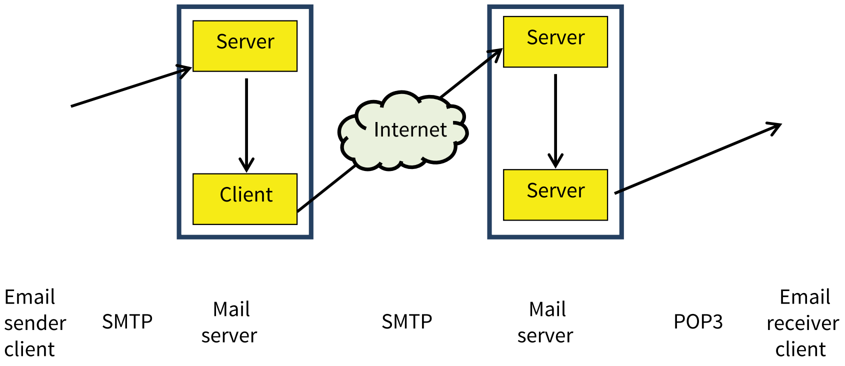


Figure 17.04 An email message being transmitted from a sender to a receiver

Of the two protocols shown being used, SMTP (Simple Mail Transfer Protocol) is a ‘push’ protocol whereas POP3 (Post Office Protocol version 3) is a ‘pull’ protocol. There is a more recent alternative to POP3, which is IMAP (Internet Message Access Protocol). The approach using POP3 is for emails to be downloaded onto the client computer. With IMAP the emails are not downloaded; they remain stored on the server but remain accessible from the client. It can be argued that POP3 is more secure to cyber-attack because emails are locally stored. However, the server will be backed up regularly whereas the local client might not be. The major advantage for IMAP is that the server can be accessed from any client. This makes it suitable for anyone on the move or if someone is using a system other than that normally used. POP3 emails are only accessible from the one client system.

SMTP has been largely replaced by the use of web-based mail. A browser is used to access the email application, so HTTP is now the protocol used. However, SMTP remains in use for transfer between the mail servers.

Question 17.02

Which protocols are being used in your email system? Does your email system give you any choices?

FTP (File Transfer Protocol)

For routine transfers of files from one user to another the most likely method is to attach the file to an email. However, this is not always a suitable method. FTP (File Transfer Protocol) is the application-layer protocol that can handle any file transfer between two end-systems. File transfer is not simple if the end-systems have different operating systems with different file systems. FTP handles this by separating the control process from the data-transfer process.

Question 17.03

If you were setting up a website, you would need to upload files to a server. Do you know how you can use FTP for this from the system you normally use?

17.07 Peer-to-peer (P2P) file sharing

The network traffic generated by peer-to-peer (P2P) file sharing is one of the main features of Internet use. P2P is an architecture that has no structure and no controlling mechanism. Peers act as both clients and servers and each peer is just one end-system. When a peer acts as a server it is called a 'seed'.

The BitTorrent protocol is the most used protocol because it allows fast sharing of files. There are three basic problems to solve if end-systems are using BitTorrent.

1 How does a peer find others that have the wanted content?

Every content provider should provide a content description, called a torrent, which is a file that contains the name of the tracker (a server that leads peers to the content) and a list of the chunks that make up the content. The torrent file is at least three orders of magnitude smaller than the content so can be transferred quickly. The tracker is a server that maintains a list of all the other peers (the 'swarm') actively downloading and uploading the content.

2 How do peers replicate content to provide high-speed downloads for everyone?

Peers download and upload chunks at the same time, but peers have to exchange lists of chunks and aim to download rare chunks for preference. Each time a rare chunk is downloaded it automatically becomes less rare!

3 How do peers encourage other peers to provide content rather just using the protocol to download for themselves?

This requires dealing with the free-riders or 'leechers' who only download. The solution is for a peer to initially randomly try other peers but then to only continue to upload to those peers that provide regular downloads. If a peer is not downloading or only downloading slowly, the peer will eventually be isolated or 'choked'.

It is worth noting that the language of BitTorrent is quite unusual and there are other terms used which have not been mentioned. Fortunately, the principles are straightforward.

Question 17.04

Who runs the tracker server? Are there alternative approaches for BitTorrent?

Reflection Point:

Networking is a large subject area with many interconnected concepts. Have you considered how you might make some structured notes to help with revision later?

Summary

- Circuit switching requires a dedicated circuit to be established between sender and receiver before transmission can start.
- In packet switching, packets can be transmitted without any circuit being created.
- Packet switching can use connectionless or connection-oriented transmission.
- A protocol suite is implemented as a layered stack.
- The TCP/IP protocol suite supports usage of the Internet.
- Ethernet is the most likely option for use in the Network and Data link layers.
- Examples of application-layer protocols are HTTP, SMTP, POP3, IMAP and FTP.
- Peer-to-peer file sharing on the Internet uses the BitTorrent protocol.

Exam-style Questions

1 The following represents the structure of an IP datagram.

IP Header	TCP Header	TCP Content

Using the code IP for IP Header, TCP for TCP Header and Content for TCP Content identify where the following data will be found:

a Destination address

b Sender address

c Destination port

d Sender port

e Packet sequence number

f Acknowledgement. [6]

2 Ethernet can be used in conjunction with the TCP/IP protocol suite.

a Draw a diagram to illustrate how the combination of Ethernet and the TCP/IP suite provides support for data communication. [5]

b Explain the meaning of the term ‘MAC address’. [3]

3 One end-system with an Internet connection has a file. A user on another end-system connected to the Internet needs a copy of the file. There are different methods that might be used to enable the user to obtain a copy of the file.

a Identify **three** possible methods with a brief explanation for each. [6]

b Identify the application-layer protocols that each method will use with a brief explanation for each one. [8]

4 For sending and receiving emails the following application protocols might be used:

SMTP

POP3

IMAP

HTTP

For each of these protocols give a brief explanation as to how they might be used in association with an email application. You might find it useful to include a diagram for some of your account. [12]

5 **a** Four descriptions and three protocols are shown below.

Draw a line to connect each description to the appropriate protocol.

Description	**Protocol used**
email client downloads an email from an email server	HTTP
email is transferred from one email server to another email server	POP3
email client sends email to email server	SMTP

browser sends a request for a web page to a web server

[4]

b Downloading a file can use the client-server model. Alternatively, a file can be downloaded using the BitTorrent protocol.

Name the model used. [1]

c For the BitTorrent protocol, explain the function of each of the following:

i tracker [2]

ii seed [2]

iii swarm [2]

Cambridge International AS & A level Computer Science 9608 paper 31 Q6 June 2015

6 An email is sent from one email server to another using packet switching.

a State **two items** that are contained in an email packet apart from the data. [2]

b Explain the role of routers in sending an email from one email server to another. [3]

c Sending an email message is an appropriate use of packet switching.

Explain why this is the case. [2]

d Packet switching is not always an appropriate solution.

Name an alternative communication method of transferring data in a digital network. [1]

e Name an application for which the method identified in **part (d)** is an appropriate solution. Justify your choice. [3]

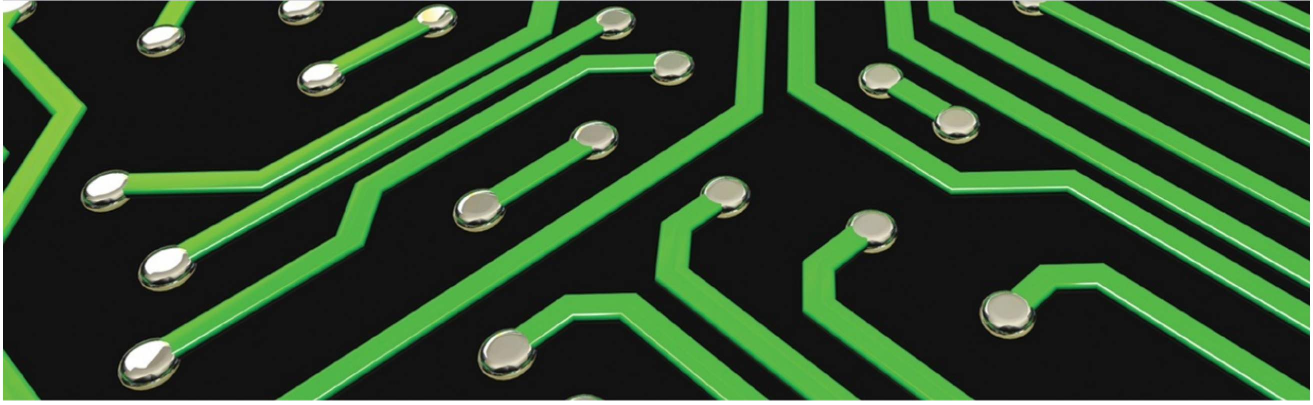
Cambridge International AS & A level Computer Science 9608 paper 31 Q3 November 2015

Chapter 18:
Hardware and virtual machines

Learning objectives

By the end of this chapter you should be able to:

- show understanding of Reduced Instruction Set Computers (RISC) and Complex Instruction Set Computers (CISC) processors
- show understanding of the importance/use of pipelining and registers in RISC processors
- show awareness of the four basic computer architectures: SISD, SIMD, MISD, MIMD
- show understanding of the characteristics of massively parallel computers
- show understanding of the concept of a virtual machine.



18.01 The control unit

While a program is being executed, the CPU is receiving a sequence of machine-code instructions. It is the responsibility of the control unit within the CPU to ensure that each machine instruction is handled correctly. There are two methods by which a control unit can be designed to allow it to perform its function.

One method is for the control unit to be constructed as a logic circuit. This is called the hard-wired solution. The machine-code instructions are handled directly by hardware.

The alternative method is for the control unit to use microprogramming. In this approach, the control unit contains a ROM component that stores the microinstructions or microcode for microprogramming, often referred to as firmware. In the next section we will see what might influence the choice of design of the control unit.

18.02 CISC and RISC processors

A processor will have an architecture which refers to its physical construction. However, a processor will also have what is termed an ‘instruction set architecture’.

This is concerned with:

- the instruction set
- the instruction format
- the addressing modes
- the registers accessible by instructions.

The choice of the instruction set is the main factor in distinguishing one instruction set architecture from another.

In the early days of computing a significant factor in choosing the instruction set architecture for a computer was that it should make the writing of a compiler for a high-level language easier. At that time the term did not exist, but we would now refer to this as being the architecture for a **Complex Instruction Set Computer (CISC)**.

This philosophy began to be challenged in the late 1970s. It was argued that using a **Reduced Instruction Set Computer (RISC)** would be a better approach. Table 18.01 contains a number of features that distinguish RISC from CISC.

RISC	CISC
Fewer instructions	More instructions
Simpler instructions	More complex instructions
Small number of instruction formats	Many instruction formats
Single-cycle instructions whenever possible	Multi-cycle instructions
Fixed-length instructions	Variable-length instructions
Only load and store instructions to address memory	Many types of instructions to address memory
Fewer addressing modes	More addressing modes
Multiple register sets	Fewer registers
Hard-wired control unit	Microprogrammed control unit
Pipelining easier	Pipelining more difficult

Table 18.01 Comparison of RISC with CISC

The following are some points to note.

- For RISC the term ‘reduced’ affects more than just the number of instructions.
- A reduction in the number of instructions is not the major driving force for the use of RISC.
- The reduction in the complexity of the instructions is a key feature of RISC.
- The typical CISC architecture contains many specialised instructions.
- The specialised instructions are designed to match the requirement of a high-level programming language.
- The specialised instructions require multiple memory accesses which are very slow compared with register accesses.
- The simplicity of the instructions for a RISC processor allows data to be stored in registers and

manipulated in them with no resource to memory access other than that necessary for initial loading and possible final storing.

- The simplicity of RISC instructions makes it easier to use hard-wiring inside the control unit.
- The complexity of many of the CISC instructions makes hard-wiring much more difficult so microprogramming is the norm.

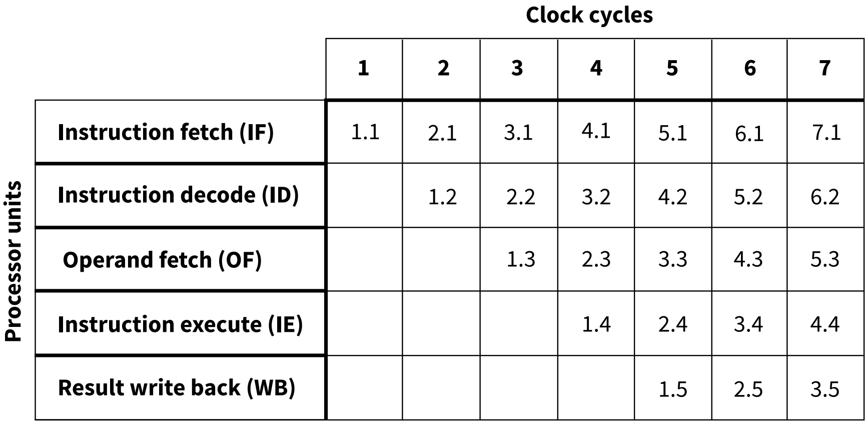
Extension Question 18.01

Can you find out whether the processors in any systems you are using are described as RISC or CISC?

Pipelining

One of the major driving forces for creating RISC processors was the opportunity they would provide for efficient **pipelining**. Pipelining is a form of parallelism applied specifically to instruction execution. Other forms of parallelism are discussed in Section 18.03.

The underlying principle of pipelining is that the fetch–decode–execute cycle described in Chapter 5 (Section 5.06) can be separated into a number of stages. One possibility is a five-stage model consisting of:



Clock cycles

Processor units

	1	2	3	4	5	6	7
Instruction fetch (IF)	1.1	2.1	3.1	4.1	5.1	6.1	7.1
Instruction decode (ID)		1.2	2.2	3.2	4.2	5.2	6.2
Operand fetch (OF)			1.3	2.3	3.3	4.3	5.3
Instruction execute (IE)				1.4	2.4	3.4	4.4
Result write back (WB)					1.5	2.5	3.5

Figure 18.01 Pipelining for five-stage instruction handling

Figure 18.01 shows how pipelining would work with this five-stage breakdown of instruction handling. For pipelining to be implemented, the construction of the processor must have five independent units, with each handling one of the five stages identified. This explains the need for a RISC processor to have many register sets; each processor unit must have access to its own set of registers. Figure 18.01 uses the representation 1.1, 1.2 and so on to define the instruction and the stage of the instruction. Initially only the first stage of the first instruction has entered the pipeline. At clock cycle 6 the first instruction has left the pipeline, the last stage of instruction 2 is being handled and instruction 6 has just entered.

Once under way, the pipeline is handling five stages of five individual instructions. In particular, at each clock cycle the complete processing of one instruction has finished. Without the pipeline the processing time would be five times longer.

One issue with a pipelined processor is interrupt handling. The discussion in Chapter 5 (Section 5.08) referred to a processor with instructions handled sequentially. This approach where a check for any interrupt is made following the execution of an instruction is applicable to a CISC processor. It would also be applicable to a RISC processor if there were no pipelining. However, this is an unlikely circumstance. In the pipelined system described above there will be five instructions in the pipeline when an interrupt occurs. One option for handling the interrupt is to erase the pipeline contents for the latest four instructions to have entered. Then the normal interrupt-handling routine can be applied to the remaining instruction. The other option is to construct the individual units in the processor with individual program counter registers. This option allows current data to be stored for all of the instructions in the pipeline while the interrupt is handled.

Discussion Point:

Consider the two consecutive instructions:

```
ADD R1, R2, R3
```

```
ADD R5, R1, R4
```

These are typical three-register instructions favoured for RISC. The first adds the contents of registers R2 and R3 and stores the result in R1. The next instruction is similar but uses the value stored in R1. In a pipelined structure, the second instruction will be reading the contents of R1 before the previous instruction has placed the value there. How could this potential problem be overcome?

18.03 The basic computer architectures

A well-established approach to describing different computer architectures is to consider the number of instruction streams and the number of data streams. This leads to four different types.

Single Instruction Stream Single Data Stream (SISD)

SISD is the typical arrangement found in early computers which was also adopted for the earliest microprocessors. The functioning is purely sequential with no parallelism. Figure 18.02 illustrates how individual elements in an array are manipulated using SISD.

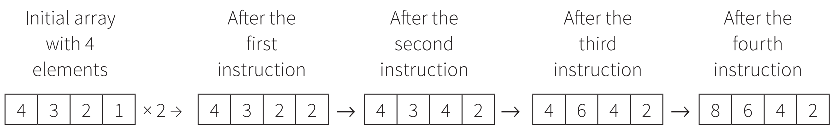

Initial array with 4 elements
After the first instruction
After the second instruction
After the third instruction
After the fourth instruction
4 3 2 1 × 2 →
4 3 2 2 →
4 3 4 2 →
4 6 4 2 →
8 6 4 2

Figure 18.02 SISD sequential processing of array elements

The same instruction is used repeatedly to carry out the multiplication by two.

Single Instruction Stream Multiple Data Stream (SIMD)

SIMD can be considered as parallelism applied to the data stream. The difference when array elements are processed is shown in Figure 18.03.

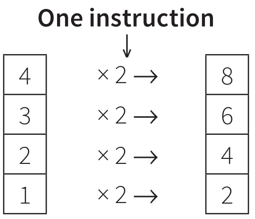


Figure 18.03 SIMD parallel processing of array elements

The architecture for SIMD is usually represented by a diagram such as Figure 18.04.

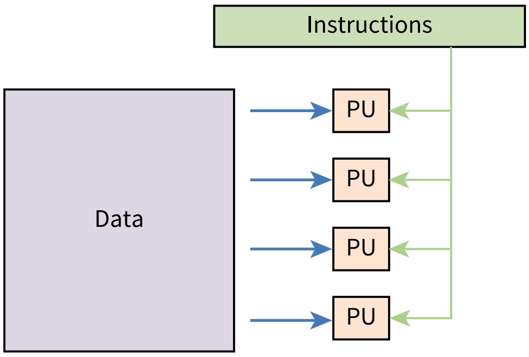


Figure 18.04 Schematic representation of the SIMD architecture

In Figure 18.04 the structure shows four data streams entering four individual components which are simultaneously supplied with the same instruction. The components are labelled PU (Processing Unit). Sometimes the components are labelled PE (Processing Element). Whatever name is used, the components are arithmetic logic units. One of the reasons for the different names is that there are options for how this architecture could be implemented.

TIP

Be careful to distinguish between the pipelining of instructions where different instructions are executed in parallel and SIMD when only one instruction is being simultaneously executed on different data streams.

One option has been used in computers that are called array or vector processors. These might have a parallel set of registers; one for each data stream. Alternatively, there would be a large register, perhaps with 64 or 128 bits which could store four data values at the same time. In this type of implementation, the parallelism is built into just one processor.

The alternative is the multi-core processor where four individual processors work in parallel. In this case each processor is likely to have its own dedicated cache memory to provide the data stream.

Multiple Instruction Stream Single Data Stream (MISD)

MISD is not evidenced in any individual computer architecture design. One example where the approach could be implemented would be in a fault-tolerant system. The same data stream could be fed into two or more processors. The output would only be accepted if the same output was produced by all of the processors.

Multiple Instruction Stream Multiple Data Stream (MIMD)

The **MIMD** architecture is similar to that for the type of SIMD architecture shown in Figure 18.04 with more than one processing unit receiving the parallel data streams. The difference is that each Processing unit does not execute the same instruction. The multiple data stream can be provided by a suitably partitioned single memory. Each Processing Unit might have a dedicated cache memory.

Massively parallel computer systems

The **MIMD** architecture can be implemented in multicomputer systems known as massively parallel computers. These are the systems used by large organisations for computations involving highly complex mathematical processing. They are the latest type of ‘supercomputer’. The major difference in architecture is that instead of having a bus structure to support multiple processors there is a network infrastructure to support multiple computer units. The programs running on the different computers can communicate by passing messages using the network.

An alternative type of multicomputer system is cluster computing using PCs (sometimes referred to as a ‘server farm’).

Whatever the technology used, these systems have an extremely large number of individual processors working in parallel.

Extension Question 18.02

The IBM Sequoia and a Beowulf cluster are different examples of cluster computing. You might wish to carry out some research to get some details of their computing power.

18.04 Virtual machines

A virtual machine is software not hardware. The most usual type of virtual machine is the **system virtual machine** which is software that emulates the hardware of a real computer system.

Remember that when a virtual machine is not being used an application program requires support from an operating system in order for the program to run on the hardware. The principle of a virtual machine is that a process interacts directly with a software interface provided by an operating system. The logical structure for the operation of a system virtual machine is shown in Figure 18.05.

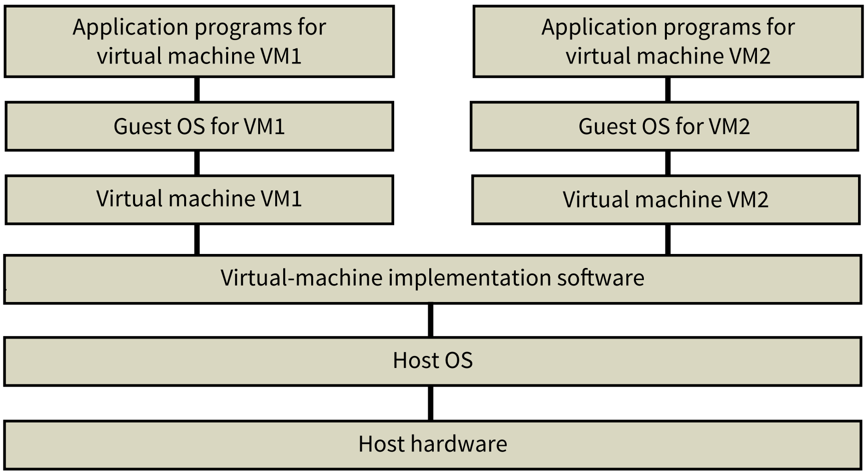

Application programs for virtual machine VM1
Application programs for virtual machine VM2
Guest OS for VM1
Guest OS for VM2
Virtual machine VM1
Virtual machine VM2
Virtual-machine implementation software
Host OS
Host hardware

Figure 18.05 Logical structure for a system virtual machine implementation

The following points should be noted when looking at Figure 18.05.

1. The application programs are installed with the assistance of a guest OS. This guest OS will support the running application by interacting with the virtual machine as though it were the hardware that the guest OS would normally run on.
2. The virtual machine implementation software can be considered to be a utility program which, when running, is supported by the particular host OS which is specific to the host hardware.
3. There can be application programs running at the same time directly on the host hardware under the control of the host OS.

The main advantage of the virtual machine approach is that more than one different operating system can be made available on one computer system. This is particularly valuable if an organisation has legacy systems and wishes to continue to use the old software but does not wish to keep the old hardware. Alternatively, the same operating system can be made available many times by companies with large mainframe computers that offer server consolidation facilities. Different companies can be offered their own virtual machine running as a server.

One drawback to using a virtual machine is the time and effort required for implementation. Another drawback is the fact that the implementation will not offer the same level of performance that would be obtained on a normal system.

Discussion Point:

Are there other advantages or disadvantages in using a system virtual machine?

The Java virtual machine discussed in Chapter 8 (Section 8.05) is an example of a process virtual machine, based on a different underlying concept. The process virtual machine provides a platform-independent programming environment that allows a program to execute in the same way on any platform. This is specific software that only supports running a Java program. A system virtual machine supports any application.

Reflection Point:

The description ‘virtual’ occurs in several places in this book. Are you clear about how these different examples have their own specific context and meaning?

Summary

- A control unit can be hard-wired or microprogrammed.
- RISC (Reduced Instruction Set Computers) processors have a number of advantages compared to CISC (Complex Instruction Set Computers).
- Pipelining is one of the reasons for choosing a RISC architecture.
- Parallelism can be based at the instruction level, processor level or computer level.
- A system virtual machine is software emulating hardware.
- The Java Virtual Machine is an example of a process virtual machine.

Exam-style Questions

1 **a** Computer systems are now often constructed with RISC processors.

i State what the acronym RISC stands for. [1]

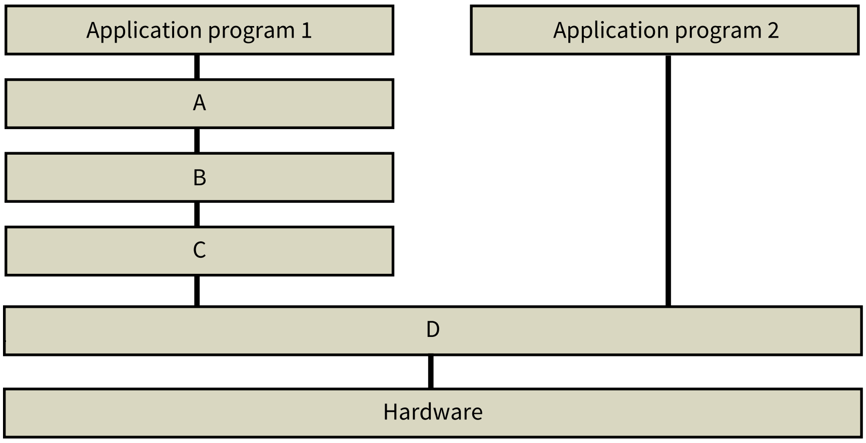
ii State **four** characteristics to be expected of a RISC system. [4]

b A RISC processor is likely to be ‘hard-wired’.

i Explain what this term means and which specific part of the processor will be hard-wired. [3]

ii State what the alternative to hard-wiring is and what hardware component is needed to be part of the processor to allow this alternative to be implemented. [2]

2 The following diagram represents a system which has implemented a virtual machine.



a For each of A, B, C, D give a suitable name. [4]

b For each of A, B, C and D give a brief description of the function of the feature. [8]

c Explain why the diagram is different for Application 2 [2]

3 **a** Parallelism can be achieved in a number of ways.

i Identify **three** different types of parallelism. [3]

ii Identify which type pipelining belongs to. [1]

iii Using a diagram, explain how pipelining works. [5]

b Interrupt handling is not so straightforward in a pipelined system. Explain why this is so and give a brief account of how problems can be avoided. [3]

4 **a** Three descriptions and two types of processor are shown below.

Draw a line to connect each description to the appropriate type of processor.

Description	**Type of processor**
Makes extensive use of general purpose registers	RISC
Many addressing modes are available	CISC
Has a simplified set of instructions	

[3]

b In a RISC processor three instructions (A followed by B, followed by C) are processed using pipelining.

The following table shows the five stages that occur when instructions are fetched and executed.

i The “A” in the table indicates that instruction A has been fetched in time interval 1.

Complete the table to show the time interval in which each stage of each instruction (A, B, C) is carried out.

Stage	**Time interval**								
	1	2	3	4	5	6	7	8	9
Fetch instruction	A								
Decode instruction									
Execute instruction									
Access operand in memory									
Write result to register									

[3]

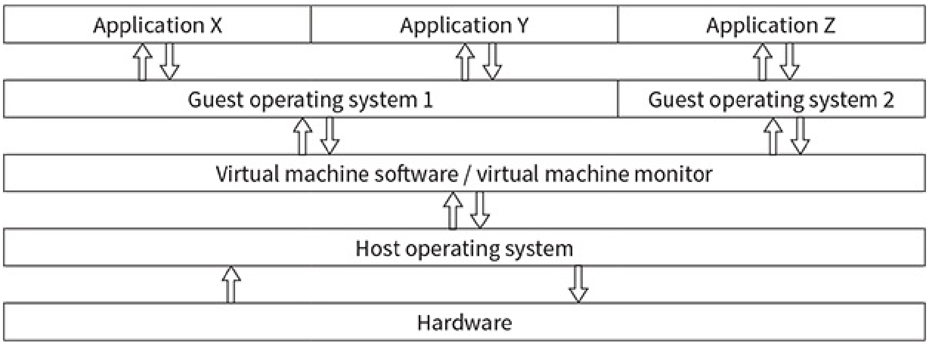
ii The completed table shows how pipelining allows instructions to be carried out more rapidly. Each time interval represents one clock cycle.

Calculate how many clock cycles are saved by the use of pipelining in the above example.

Show your working. [3]

Cambridge International AS & A level Computer Science 9608 paper 31 Q4 November 2015

5 **a** The following diagram shows how applications X, Y and Z can run on a virtual machine system.



i The virtual machine software undertakes many tasks.

Describe **two** of these tasks. [2]

ii Explain the difference between a guest operating system and a host operating system. [2]

b A company uses a computer as web server. The manufacturer will no longer support the computer’s operating system (OS) in six months time. The company will then need to decide on a replacement OS.

The company is also considering changing the web server software when the OS is changed.

Whenever any changes are made, it is important that the web server service is not disrupted.

In developing these changes, the company could use virtual machines.

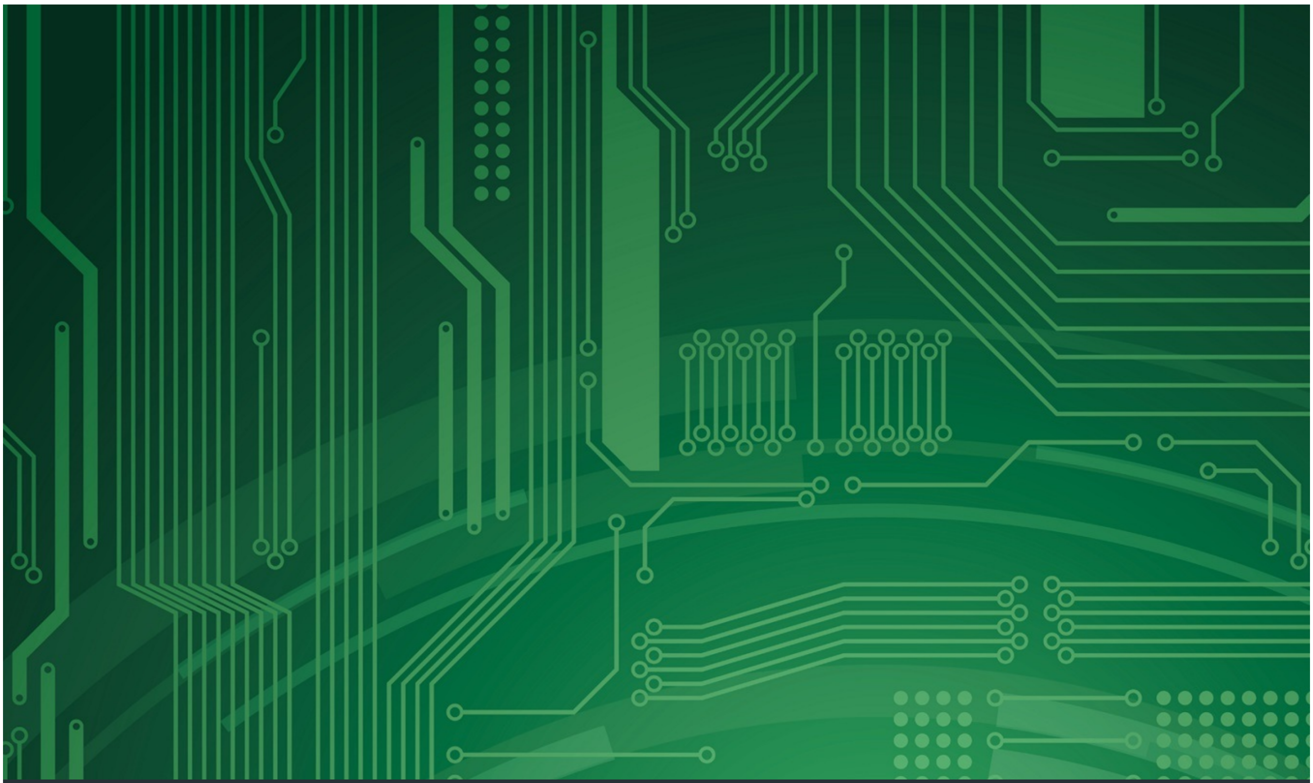
i Describe **two** possible uses of virtual machines by the company. [4]

The web server often has to handle many simultaneous requests.

ii The company uses a virtual machine to test possible solutions to the changes that they will need to make.

Explain **one** limitation of this approach. [2]

Cambridge international AS &; A Level Computer Science 9608 paper 31 Q3 June 2016



Chapter 19:
Logic circuits and Boolean algebra

Learning objectives

By the end of this chapter you should be able to:

- produce truth tables for logic circuits including half adders and full adders
- show understanding of a flip-flop (SR, JK)
- show understanding of Boolean algebra
- show understanding of Karnaugh Maps.

19.01 Logic circuits

Chapter 4 introduced the symbols for logic gates that are used in logic circuits and discussed the relationships between logic circuits, truth tables and logic expressions. This chapter introduces some specific circuits that are used to construct components that provide functionality in computer hardware.

The half adder

A fundamental operation in computing is binary addition. The result of adding two bits is either 1 or 0. However, when 1 is added to 1 the result is 0 but there is a carry bit equal to 1. This cannot be ignored if two numbers with several bits in each are being added.

The simplest circuit that can be used for binary addition is the **half adder**. This can be represented by the diagram in Figure 19.01. The circuit takes two input bits and outputs a sum bit (S) and a carry bit (C).

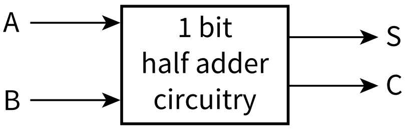


Figure 19.01 A half adder

The truth table for this circuit is shown in Table 19.01.

Input		Output	
A	B	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Table 19.01 The truth table for a half adder

One possible circuit can be defined directly by examination of the truth table. It can be seen that the only combination of inputs that produces a 1 for the carry bit is when two 1 bits are input. The truth table for the C output is in fact the AND truth table. The truth table for the S output can be seen to match that for the XOR operator which is shown in Figure 4.02 in Chapter 4 (Section 4.04). Therefore, one circuit that would produce the half adder functionality would contain an AND gate and an XOR gate with each gate receiving input from A and B.

This is only one of several circuits that would provide the functionality. The NAND and NOR gates are universal gates. Any logic circuit can be constructed using only NAND gates or only NOR gates. This fact combined with the ease of manufacture of these gates leads circuit manufacturers to prefer their use. The circuit shown in Figure 19.02 consisting only of NAND gates has the correct logic to produce the C and S outputs and is a likely choice for implementation.

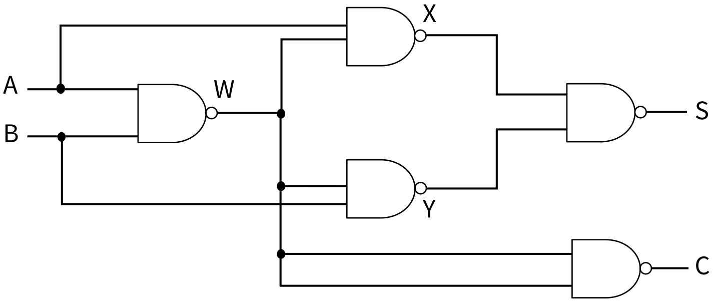


Figure 19.02 A half adder circuit constructed from NAND gates

Question 19.01

In Figure 19.02, can you identify the individual circuits that represent the AND operator and the XOR operator?

TASK 19.01

Use the intermediate points labelled W, X and Y to construct a truth table for the circuit shown in Figure 19.02. Check that this reproduces the truth table shown as Table 19.01.

The full adder

If two numbers expressed in binary with several bits to be added, the addition must start with the two least significant bits and then proceed to the most significant bits. At each stage, the carry from the previous addition has to be incorporated into the current addition. If a half adder is used each time, there has to be separate circuitry to handle the carry bit because the half adder only takes two inputs.

The **full adder** is a circuit that has three inputs including the previous carry bit. The truth table is shown as Table 19.02.

Input			Output	
A	**B**	**Cin**	**S**	**Cout**
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Table 19.02 The truth table for a full adder

One possible circuit for implementation contains two half adder circuits and an OR gate as shown in Figure 19.03.

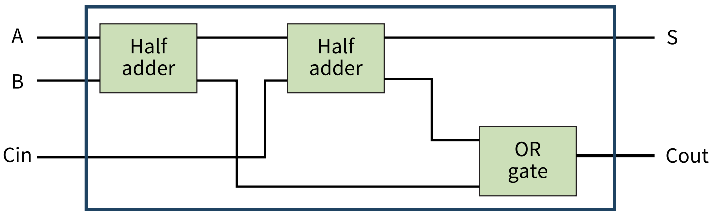


Figure 19.03 A possible implementation of a full adder

As before, it is possible to construct the circuit entirely from NAND gates as shown in Figure 19.04.

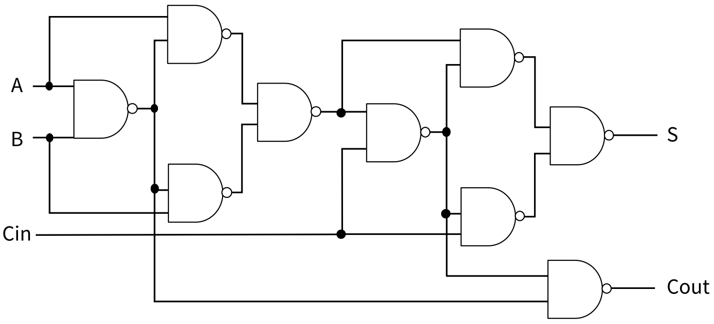


Figure 19.04 A full adder circuit constructed from NAND gates

Discussion Point:

Can you see how full adders could be combined to handle addition of, for example, four-bit binary numbers? What happens to the carry input for the first addition?

19.02 Sequential logic circuits

All of the circuits so far encountered in this book have been **combinational circuits**. For such a circuit the output is dependent only on the input values. An alternative type of circuit is a **sequential circuit** where the output depends on the input and on the previous output.

The SR flip-flop

The SR flip-flop or 'latch' is a simple example of a sequential circuit. It can be constructed with two NAND gates or two NOR gates. Figure 19.05 shows the version with two NOR gates

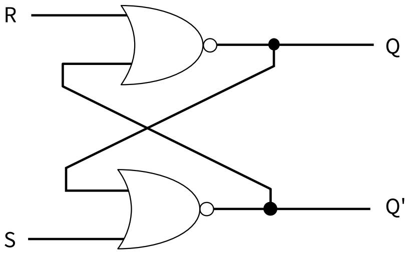


Figure 19.05 A circuit for an SR flip-flop using NOR gates

The cross-coupled circuit with output feedback has some interesting consequences. For example, if we consider the hypothetical situation where R, S, Q and Q' all had value 0 we can see that this would be self-contradictory because two inputs value 0 to a NOR gate produce a 1 output. A state with Q and Q' both set to value 1 would be self-contradictory because the output from a NOR gate is always 0 if either input is 1. The flip-flop is therefore a two-state device. Either it has Q=1 and Q'=0 or it has the reverse.

The truth table for the circuit can be presented as shown in Table 19.03. The two lines of the truth table where both S and R are input as 0 represent the condition when no signal is input to the flip-flop. If we consider the state with Q=1 and Q'=0:

- and we consider the condition that both S and R inputs are 0
- then the top NOR gate has inputs both 0
- giving output 1
- and the bottom NOR gate has inputs 1 and 0
- giving output 0.

This means that this is a self-consistent state which is referred to as the set state. A similar argument can be applied for the alternative to the set state which is the unset state with Q=0 and Q'=1.

The S and R inputs are for set and reset, respectively. The truth table shows that an input combination of S=0 and R=1 converts a set state to an unset state and an input combination of S=1 and R=0 converts an unset state to a set state.

Input signals		Initial state		Final state	
S	R	Q	Q'	Q	Q'
0	0	1	0	1	0
1	0	1	0	1	0
0	1	1	0	0	1
0	0	0	1	0	1
1	0	0	1	1	0
0	1	0	1	0	1

Table 19.03 A representation of a truth table for an SR flip-flop

These properties explain why the SR flip-flop can be used as a storage device for 1 bit and therefore could be used as a component in RAM because a value is stored but can be altered. The truth table does not contain rows for R=1 and S=1 because this leads to an invalid state with both Q and Q’ having value 0. Because of this the circuit must be protected from receiving an input signal on R and S simultaneously.

The alternative NAND gate circuit for the SR flip-flop has a similar structure but the labeling is different. The important difference is that setting is achieved with S=0 and R=1 and resetting with R=0 and S=1.

TASK 19.02

Find a diagram for a NAND gate SR flip-flop and construct the truth table.

The JK flip-flop

In addition to the possibility of entering an invalid state there is also the potential for a circuit to arrive in an uncertain state if inputs do not arrive quite at the same time. In order to prevent this, a circuit may include a clock pulse input to give a better chance of synchronising inputs. The JK flip-flop is an example.

The JK flip-flop can be illustrated by the symbol shown in Figure 19.06(a). A possible circuit is shown in Figure 19.06(b).

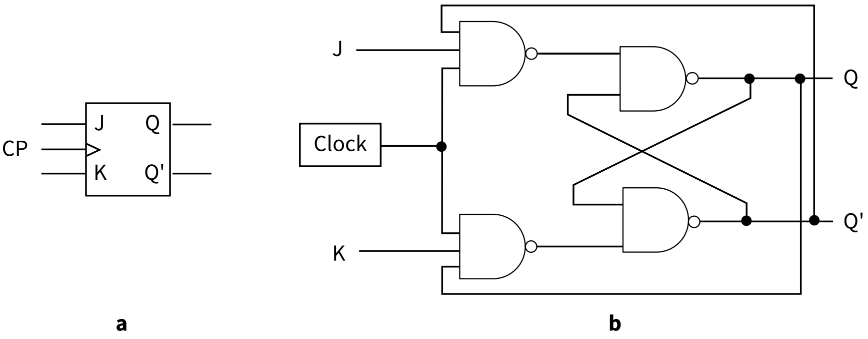


Figure 19.06 (a) A symbol for a JK flip-flop and (b) a possible circuit

TIP

There is one special case for a NAND gate; when all inputs are 1 the output is 0. There is one special case for a NOR gate; when all inputs are 0 the output is 1.

To understand the workings of the circuit you must first remember that a NAND gate gives a 1 output unless all inputs are 1. If the circuit is in the unset state with Q = 0 and Q’ =1 this is stable and self-consistent as shown by the following argument:

- the clock and R both input 0
- the top-left NAND gate therefore has output 1
- this then leads to two 1 inputs to the top-right NAND gate
- ensuring that Q = 0.

If the input from J now becomes 1 and the clock pulse switches to a 1 then:

- the top-left NAND gate has all 1 inputs
- so the output is 0
- this causes the top-right NAND gate to output a 1 for Q

- there are now two 1 inputs to the bottom-right NAND gate
- so the Q’ output becomes 0.

This shows how the J input is a set input. A similar argument shows that the K is a clear input. In this respect the JK flip-flop behaves in a similar way to the SR flip-flop as a storage device for one bit. However, there is an important difference in that if both J and K are input as a 1 then the values for Q and Q’ are toggled (they switch value). This makes the JK flip-flop a more reliable device because there is no combination of input states that leave uncertainty as to which values are stored. The significant part of the truth table is shown as Table 19.04.

J	K	Clock	Q
0	0	↑	Q unchanged
1	0	↑	1
0	1	↑	0
1	1	↑	Q toggles

Table 19.04 Part of the truth table for a JK flip-flop

Question 19.02

Can you follow the logic in the JK flip-flop circuit to see that the truth table is an accurate representation of the logic for all combinations of input?

19.03 Boolean algebra basics

Chapter 4 (Section 4.01) introduced logic expressions consisting of logic propositions combined using Boolean operators. Boolean algebra provides a simplified way of writing a logic expression and a set of rules for manipulating an expression.

Whenever a form of algebra is used it is vital that there is an understanding of its meaning. As a simple example we can consider the following four interpretations of the meaning of 1 + 1:

1 + 1 = 2

1 + 1 = 10

1 + 1 = 0

1 + 1 = 1

The first shows denary arithmetic, the second binary arithmetic and the third bit arithmetic. The last one applies if Boolean algebra is being used. This is because in Boolean algebra 1 represents TRUE, 0 represents FALSE, and + represents OR. Therefore the fourth statement represents the logic statement:

TRUE OR TRUE is TRUE

There are options for the representation of Boolean algebra. For example, the symbols for AND and OR are sometimes represented as ∧ and ∨. There is also the option of writing A.B or AB for A AND B. The dot notation is used in this book. Finally, there are options for how NOT A (the inverse of A) can be represented. A¯ is used here.

TIP

You do not need to remember the ∧ and ∨ notation.

Having established the notation for Boolean algebra we have to consider the rules that apply. These can formally be described as 'laws' or 'identities'. Table 19.05 contains a full listing.

Identity/Law	AND form	OR form
Identity	1.A = A	0 + A = A
Null	0.A = 0	1 + A = 1
Idempotent	A.A = A	A + A = A
Inverse	A.A¯=0	A.A¯=1
Commutative	A.B = B.A	A + B = B + A
Associative	(A.B).C = A.(B.C)	(A + B) + C = A + (B + C)
Distributive	A + B.C = (A + B).(A + C)	A.(B + C) = A.B + A.C
Absorption	A.(A + B) = A	A + A.B = A
De Morgan's	A.B¯=A¯+B¯	A+B¯=A¯.B¯
Double Complement	A¯¯=A	

Table 19.05 Boolean algebra identities (laws)

Some of the names used for the identities may be unfamiliar to you. This is not a concern. You should note that for all but one of the identities there is an AND form and an OR form. Furthermore, it is important to note that an identity written in one form can be transformed into the other by interchanging each 0 or 1 and each AND or OR. For example, 0.A = 0 which reads FALSE AND A is

FALSE transforms into TRUE OR A is TRUE, written in the algebra as $1 + A = 1$.

It can also be seen that some of the identities look like those applying in normal algebra with AND functioning as multiplication and OR functioning as addition. So you can use the terms ‘product’ and ‘sum’ in the context of Boolean algebra.

TASK 19.03

It is vital that you can interpret a Boolean expression correctly. Go through Table 19.05 item by item and in each case read out the full meaning. For example:

$1 + A = 1$ can be read as ‘one plus A equals 1’

but must be understood as ‘TRUE OR A is TRUE’.

Although De Morgan’s laws look complicated at first glance, they can be rationalised easily. The inverse of a Boolean product becomes the sum of the inverses of the individual values in the product. The inverse of a Boolean sum is the product of the individual inverses.

Unfortunately, using the algebra to simplify expressions is not something which can be learnt as a routine. It requires lateral thinking as Worked Example 19.01 will show.

WORKED EXAMPLE 19.01

Using Boolean algebra to simplify an expression

Let’s consider a simple example:

$$A+\bar{A}.B \text{ can be simplified to } A + B$$

In order to simplify the expression, we have to first make it more complicated! This is where the lateral thinking comes in. The OR form of the absorption identity is $A + A.B = A$. This can be used in reverse to replace A by $A + A.B$ to produce the following:

$$A+A.B+\bar{A}.B$$

We can for the moment ignore the A.

Applying the AND form of the commutative law and the OR form of the distributive law in reverse we can see that:

$$A.B+\bar{A}.B \text{ is the same as } B.A+B.\bar{A} \text{ which converts to } B.(A+\bar{A})$$

This allows us to use the OR form of the inverse identity which converts $A+\bar{A}$ into 1. As a result, the full expression, including the A that was ignored, has become:

$$A + B.1$$

The AND form of the identity law can now be applied to the B.1 term to change it to B so the expression then becomes $A + B$.

19.04 Boolean algebra applications

Creating a Boolean algebra expression directly from a truth table

One formal approach to creating a Boolean algebra expression for a particular problem is to start with the truth table and apply the sum of products method. This establishes a ‘minterm’ for each row of the table that results in a 1 for the output. This can be illustrated using the half adder circuit truth table shown in Figure 19.01. The only row of the table creating a 1 output for C has a 1 input for A and for B. The product becomes A.B and the sum of products has only this one term so we have:

$$C = A.B$$

For the S output, there are two rows that produce a 1 output so there is a sum of products containing two minterms:

$$S=\bar{A}.B+A.\bar{B}$$

Note that the 0 in a row is represented by the inverse of the input symbol.

In certain cases Boolean algebra laws can be applied directly to parts of the truth table. For example, consider a truth table which contains the following two rows:

A	B	C	X
0	1	0	1
0	1	1	1

The output for X is the same whether C = 0 or C = 1. There must therefore be a minterm which has no reference to C which in this case will be $\bar{A}.B$.

The Boolean algebra representation of a logic circuit

This approach can also be used as part of the process of creating a Boolean algebra logic expression from a circuit diagram. The truth tables for the individual logic gates are used and then some algebraic simplification is applied.

WORKED EXAMPLE 19.02

Creating a Boolean algebra logic expression for a half adder circuit

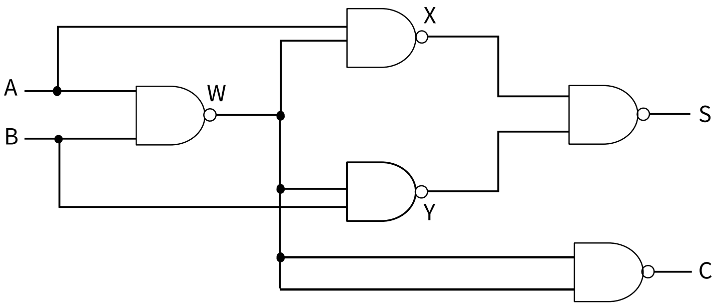


Figure 19.07 A half adder circuit

Figure 19.07 shows inputs A and B to a NAND gate with output W. The first three rows of the NAND truth table produce a 1 output so the sum of products has three minterms:

$$W=\bar{A}.\bar{B}+\bar{A}.B+A.\bar{B}$$

We can now consider the input of W to a NAND gate with A as the other input to produce the X output. Instead of using the NAND truth table we will use the fact that the NAND gate operates as an AND gate followed by a NOT gate.

The result of the AND operation is the product of the inputs so we get the following expression as output:

$$A.(\bar{A}.\bar{B}+\bar{A}.B+A.\bar{B})$$

This can be simplified by applying the distributive and inverse laws to give:

$$0+0+A.A.\bar{B} \text{ which is simply } A.\bar{B}$$

To complete the NAND operation and get the value for X we have to apply the NOT operation which means we have to take the inverse of the above expression.

This is where we need the AND version of De Morgan’s law which transforms the $A.\bar{B}$ into $\bar{A}+B$. So we have:

$$X=\bar{A}+B$$

The same laws applied to the output Y from the other intermediate NAND gate to give

$$Y=A+\bar{B}$$

Finally, we need to consider $A+\bar{B}$ and $\bar{A}+B$ being input to the final NAND gate. Again we can consider the AND operation first as the product of the inputs to produce the expression:

$$(A+\bar{B}).(\bar{A}+B)$$

We will not multiply this out; we will instead apply De Morgan’s law directly to the expression to perform the inverse operation to complete the NAND operation. This gives:

$$S=\bar{A}.B+A.\bar{B}$$

This is the value obtained directly from the truth table so the algebra has been used correctly.

Extension Question 19.01

Worked Example 19.02 did not show that the circuit produced the correct output for C. Also a shortcut was used to reach the final form of S. Can you use Boolean algebra to find the form of C from the circuit and can you convert the expression for S if you start by using the distributive law before applying De Morgan’s law?

19.05 Karnaugh maps (K-maps)

A Karnaugh map is a method of creating a Boolean algebra expression from a truth table. A K-map can make the process much easier than if you use sum-of-products to create minterms. If applied correctly a K-map produces the simplest possible form for the Boolean algebra expression.

A	B	X
0	0	0
0	1	1
1	0	1
1	1	1

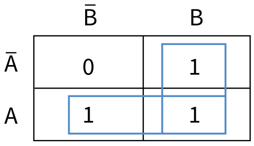
Table 19.06 The truth table for the OR operand

The truth table for an OR gate, shown as Table 19.06, can be used to illustrate the method. The sum of products method produces the following expression:

$$X=\bar{A}.B+A.\bar{B}+A.B$$

This is not instantly recognisable as A + B but, with a little effort, using Boolean algebra laws it could be shown to be the same.

The Karnaugh map approach is simpler. The corresponding K-map is shown in Figure 19.08. Each cell in a Karnaugh map shows the value of the output X for a combination of input values for A and B.



	$\bar{B}$	B
$\bar{A}$	0	1
A	1	1

Figure 19.08 A K-map of the truth table in Table 19.06

The interpretation of a Karnaugh map follows these rules.

- Only cells containing a 1 are considered.
- Groups of cells containing 1s are identified where possible, with a group being a row, a column or a rectangle.
- Groups must contain 2, 4, 8 and so on cells.
- Each group should be as large as possible.
- Groups can overlap and all overlapping groups must be used.
- If an individual cell cannot be contained in any group it is treated as being a group.
- Within each group, the only input values retained are those which retain a constant value throughout the group.

These rules define a column and a row group as indicated by the blue outlines. In the column group, B remains unchanged but A changes so B is retained. In the row group, it is A that remains unchanged. The Boolean algebra expression is then just the sum of these retained values:

$$X = A + B$$

Thus, the Karnaugh map has found the OR expression without using any algebra.

WORKED EXAMPLE 19.03

Using a K-map to interpret a three-input problem

Let’s consider the following truth table:

A	B	C	X
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Table 19.07 A sample truth table with three inputs

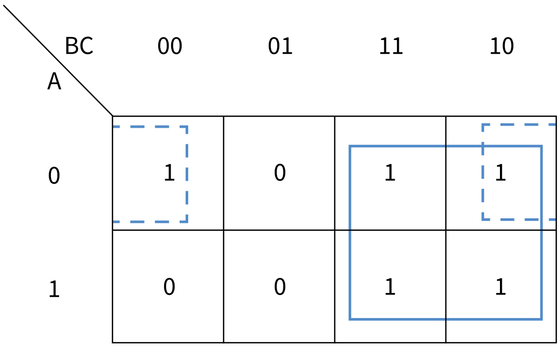
Before starting any application of a method it is always worth looking to see if there are any trends. In this case you can see that whenever B = 1 the output for X is 1. This means that the final algebra should have B + something.

Applying sum of products gives the following five-minterm expression:

A¯.B¯.C¯+A¯.B.C¯+A¯.B.C+A.B.C¯+A.B.C

There are two options for how the K-map is presented. We will choose to combine input values in the columns.

Figure 19.09 shows the result. This follows the convention of having the rows corresponding to values of A and the columns to combinations of values for B and C.



A \ BC	00	01	11	10
0	1	0	1	1
1	0	0	1	1

Figure 19.09 A K-map representation of the truth table shown in Table 19.07

It is important to note that the labelling of the columns does not follow a binary value pattern. Instead it follows the Gray coding sequence where only one bit changes value each time.

Following the rules stated above the first group to identify is the square of four cells with a value 1 as identified by the blue rectangle in the diagram. For these it can be seen that A has different values, B has a constant value but C changes values. So, only B is retained. Note this was anticipated from the initial inspection of the truth table.

This apparently only leaves the top left cell. It looks like an isolated cell but it is not because K-maps wrap round. The cell is defined by BC = 00. This has two adjacent cells under Gray coding rules. It is immediately obvious that the cell BC = 01 is adjacent but this contains 0 so it can be ignored. The other adjacent cell is the BC = 10 combination because of the wrap round rule. Thus, there is a row group containing BC = 00 and BC = 10, indicated by the dotted line partial group outlines. For this row the value A¯ remains unchanged, B changes but C¯ remains unchanged so the product A¯.C¯ results. By adding this to the B for the other group the final expression becomes:

A¯.C¯+B

This is much simpler than the expression with five minterms derived directly from the truth table.

Extension Question 19.02

Consider the Karnaugh map shown in Figure 19.10. This corresponds to a problem with four inputs. It wraps round horizontally and vertically. Use the map to create a Boolean algebra expression.

AB \ CD	00	01	11	10
00	1	0	1	1
01	0	0	1	1
11	0	0	1	1
10	1	0	0	0

Figure 19.10 A K-map for a four input problem

Reflection Point:

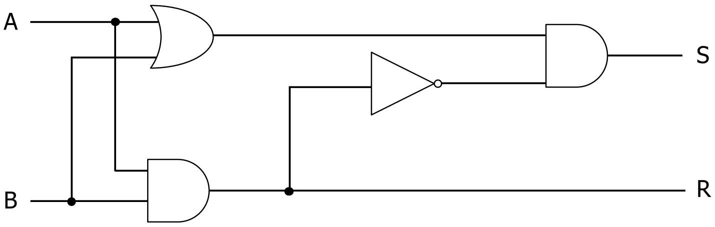
When using Karnaugh maps will you remember to use Gray coding if there are three inputs and will you remember to use all overlapping groups?

Summary

- Binary addition can be carried out using a half adder or a full adder circuit.
- SR or JK flip-flop circuits can be used to store a bit value.
- There are Boolean algebra laws that can be used to simplify logic expressions.
- The sum-of-products method can be used to create an equivalent logic expression containing minterms from a truth table.
- A Karnaugh map is a representation of a truth table that allows a simplified logic expression to be derived from a truth table.

Exam-style Questions

1 **a** Consider the following circuit:



i Identify the **three** different logic gates used. [2]

ii Complete the following truth table for the circuit for the inputs shown for A and B. [5]

Inputs		**Workspace**	**Outputs**	
A	**B**		**S**	**R**
0	0			
0	1			
1	0			
1	1			

b For the circuit shown in **part a**, identify the type of circuit and what the outputs represent. [3]

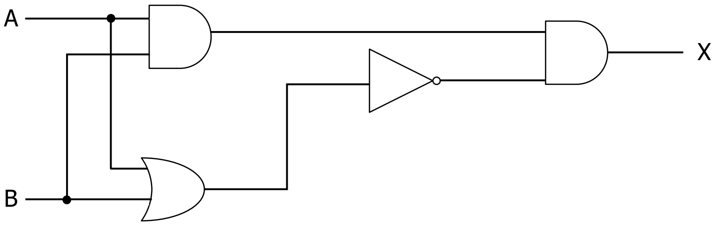
2 **a** Consider the following truth table:

A	**B**	**X**
0	0	1
0	1	0
1	0	1
1	1	1

i Using the sum-of-products approach, write a Boolean expression that matches the logic. [3]

ii For the rows that have A = 1, the output for X is 1. Explain how this would be reflected in a simplified form of Boolean expression matching the truth table. [2]

b Consider the following circuit:



i Using your knowledge of the truth table for an AND gate, write a Boolean algebra expression for the output from the first AND gate. [2]

ii Using your knowledge of the truth table for an OR gate write a Boolean algebra expression for the output from the OR gate. [3]

iii Using De Morgan's law, write the logic expression for the output from the NOT gate. [4]

3 **a** Consider the following truth table:

A	**B**	**X**
0	0	1

0	1	0
1	0	1
1	1	1

i Sketch a Karnaugh map to match this truth table. [4]

ii Use the Karnaugh map to write a Boolean algebra expression for this logic. [3]

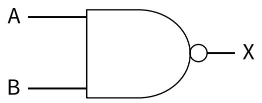
b Consider the truth table shown in **part a**.

i Use the sum-of-products method to write a Boolean algebra expression from the truth table. [3]

ii Use Boolean algebra to demonstrate that this expression can be simplified to give the same expression created from the Karnaugh map. Hint: you might wish to use the fact that

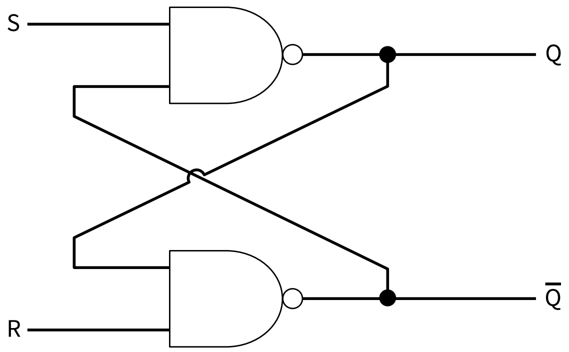
$A.\bar{B}+A.\bar{B}+A.\bar{B}$

4 **a** Complete the truth table for this NAND gate:



A	**B**	**X**
0	0	
0	1	
1	0	
1	1	

An SR flip-flop is constructed using two NAND gates. [1]



b **i** Complete the truth table for the SR flip-flop.

	S	**R**	**Q**	$\bar{Q}$
Initially	1	0	0	1
R change to 1	1	1		
S changed to 0	0	1		
S changed to 1	1	1		
S and R changed to 0	0	0		

ii One of the combinations in the truth table should not be allowed to occur. [4]

State the values of S and R that should not be allowed. Justify your choice. [3]

Another type of flip-flop is the JK flip-flop.

c **i** Give **one** extra input present in the JK flip-flop. [1]

ii Give **one** advantage of the JK flip-flop. [1]

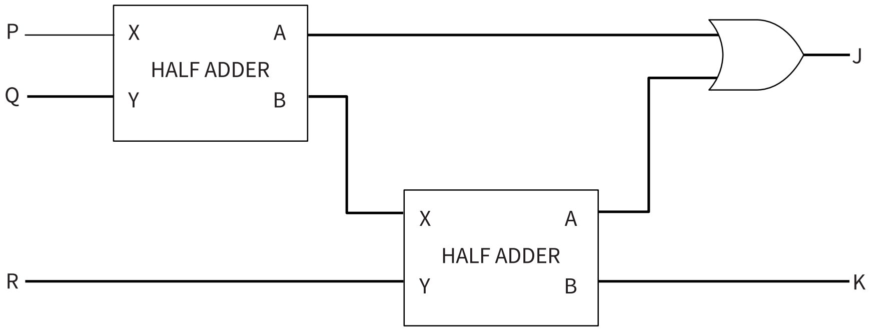
d Describe the role of flip-flops in a computer. [2]

Cambridge international AS & A Level Computer Science 9608 paper 31 Q5 June 2016

5 a i A half adder is a logic circuit with the following truth table.

Input		**Output**	
X	**Y**	**A**	**B**
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

The following logic circuit is constructed.



Complete the following truth table for this logic circuit.

Input			**Working space**	**Output**	
P	**Q**	**R**		**J**	**K**
0	0	0			
0	0	1			
0	1	0			
0	1	1			
1	0	0			
1	0	1			
1	1	0			
1	1	1			

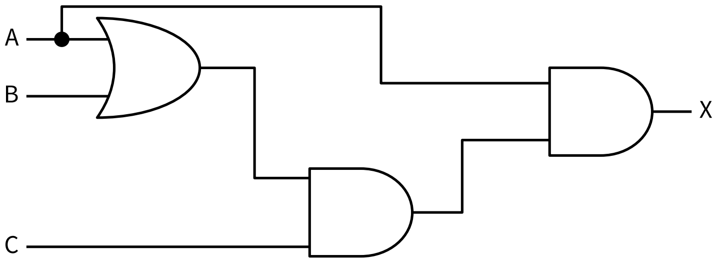
[2]

ii State the name given to this logic circuit. [1]

iii Name the labels usually given to **J** and **K**.

Explain why your answers are appropriate labels for these outputs. [4]

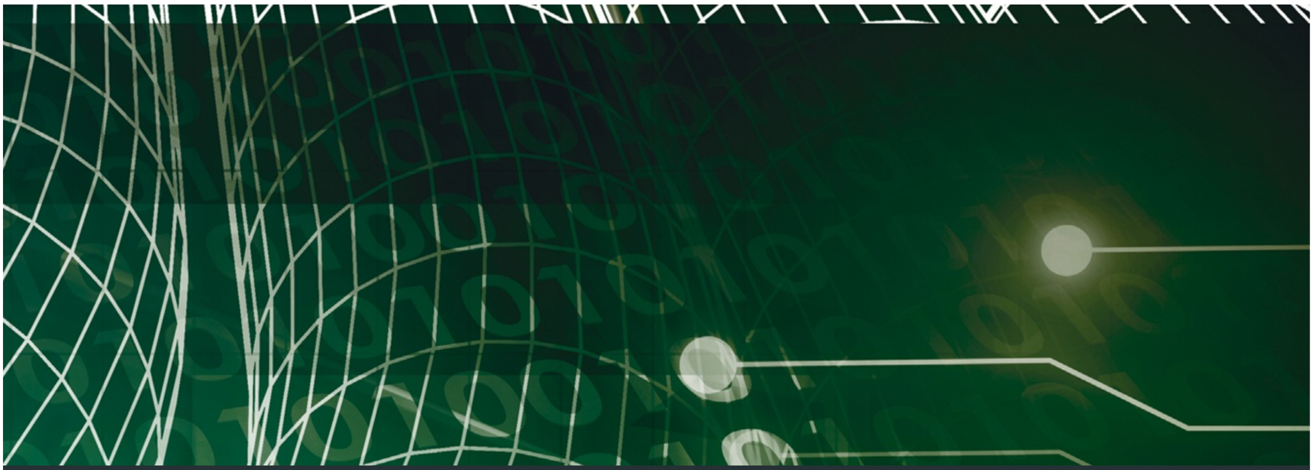
b i Write down the Boolean expression corresponding to the following logic circuit:



[2]

ii Use Boolean algebra to simplify the expression given in **part b(i)**.

Show your working. [4]



Chapter 20: System software

Learning objectives

By the end of this chapter you should be able to:

- show understanding of how an operating system (OS) can maximise the use of resources
- describe the ways in which the user interface hides the complexities of the hardware from the user
- show understanding of process-management
- show understanding of virtual memory, paging and segmentation for memory management
- show understanding of how an interpreter can execute programs without producing a translated version
- show understanding of the various stages in the compilation of a program
- show understanding of how the grammar of a language can be expressed using syntax diagrams or Backus-Naur Form (BNF) notation
- show understanding of how Reverse Polish Notation (RPN) can be used to carry out the evaluation of expressions.

20.01 The purposes of an operating system (OS)

Chapter 8 (Section 8.02) contained a categorised summary of the various activities that an operating system engages in. This chapter discusses some of them in more detail.

We can begin by considering a few aspects relating to the use of an OS.

1. A computer system needs a program that begins to run when the system is first switched on. At this stage, the operating system programs are stored on disk so there is no operating system. However, the computer has stored in ROM a basic input/output system (BIOS) which starts a bootstrap program. It is this bootstrap program that loads the operating system into memory and sets it running.
2. An operating system can provide facilities to have more than one program stored in memory. Only one program can access the CPU at any given time but others are ready when the opportunity arises, which is described as multi-programming and will happen for one single user. Some systems are designed to have many users simultaneously logged in, which is described as a time-sharing system.
3. The purposes of an operating system can usefully be considered from two viewpoints: an internal viewpoint and an external viewpoint. The internal viewpoint concerns how the activities of the operating system are organised to best use the resources available. The external viewpoint concerns the facilities made available for the user of the system.
4. The major resources associated with the internal viewpoint are the CPU, the memory and the I/O system.

Operating system structure

An operating system is structured to provide a platform for both resource management and the provision of facilities for users. The logical structure of the operating system provides two modes of operation. User mode is the one available for the user or an application program. The alternative has a number of different names but the most often used is ‘kernel mode’. The difference between the two is that kernel mode has sole access to part of the memory and to certain system functions that user mode cannot access.

The parts of the OS that provide the two modes of operation are separated. The kernel runs all of the time. The remainder of the OS runs in user mode so individual parts are only accessed when needed. One possibility then is to use a layered structure as illustrated in Figure 20.01. In this model, application programs or utility programs could make system calls to the kernel. However, to work properly each higher layer needs to be fully serviced by a lower layer (as in a network protocol stack).

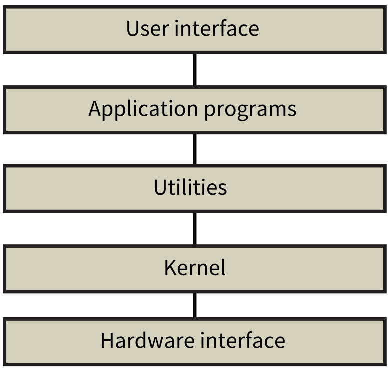


Figure 20.01 Layered structure for an operating system

A layered structure for an operating system is hard to achieve in practice. A more flexible approach uses a modular structure, illustrated in Figure 20.02. The structure works by the kernel calling on the individual services when required. It could be associated with a micro-kernel structure where the functionality in the kernel is reduced to the absolute minimum.

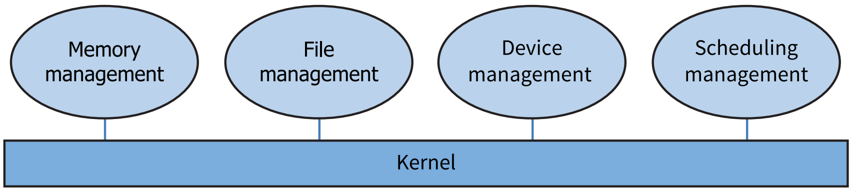


Figure 20.02 Modular structure for an operating system

20.02 The input/output (I/O) system

The I/O system does not just relate to input and output that directly involves a computer user. It also includes input and output to storage devices while a program is running. Figure 20.03 is a schematic diagram that illustrates the structure of the I/O system.

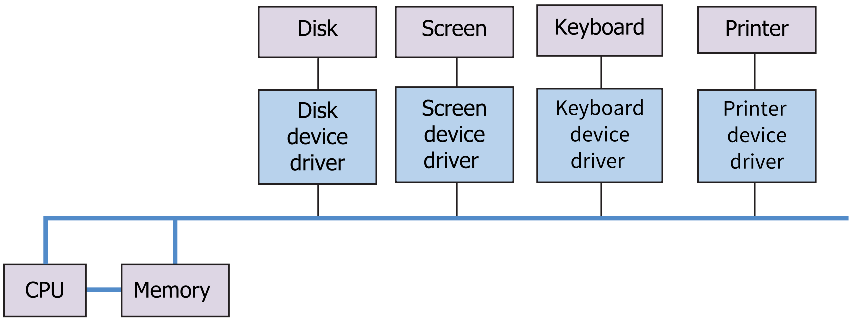


Figure 20.03 Main components associated with the I/O system

The bus structure in Figure 20.03 shows that there can be an option for the transfer of data between an I/O device and memory. The operating system can ensure that I/O passes via the CPU but for large quantities of data the operating system can ensure direct transfer between memory and an I/O device.

To understand the issues associated with I/O management, some discussion of timescales is required. It must be understood that one second is a very long time for a computer system. A CPU typically operates at GHz frequencies. One second sees more than one trillion clock cycles. Some typical speeds for I/O are given in Table 20.01.

Device	Data rate	Time for transfer of 1 byte
Keyboard	10 Bps	0.1 s
Screen	50 MBps	2×10^{-8} s
Disk	5 MBps	2×10^{-7} s

Table 20.01 Typical rates and times for data transfer

The slow speed of I/O compared to a typical CPU clock cycle shows that management of CPU usage is vital to ensure that the CPU does not remain idle while I/O is taking place. This is discussed in the next section.

20.03 Process scheduling

Resource management relating to the CPU mainly concerns scheduling to ensure efficient usage. The methods discussed here consider the CPU as a single unit; specific issues relating to a multiprocessor system are not considered.

Programs that are available to be run on a computer system are initially stored on disk. A user could submit a program as a ‘job’ which would include the program and some instructions about how it should be run. Figure 20.04 shows an overview of the components involved when a program is run.

A long-term or **high-level scheduler** program controls the selection of a program stored on disk to be moved into main memory. Occasionally a program has to be taken back to disk due to the memory getting overcrowded. This is controlled by a medium-term scheduler. When the program is installed in memory, a short-term or **low-level scheduler** controls when it has access to the CPU.

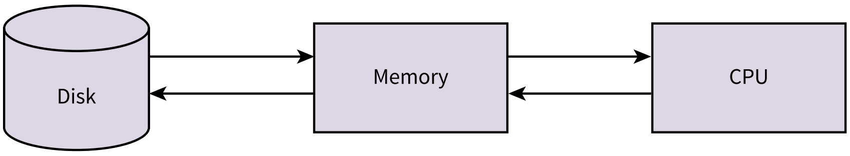


Figure 20.04 Components involved in running a program

Process states

In Chapter 8 (Section 8.02), it was stated that a **process** can be defined as ‘a program being executed’. This definition can be improved by including the state when the program first arrives in memory. At this stage a **process control block (PCB)** can be created in memory ready to receive data when the process is executed. Once in memory the state of the process can change.

The transitions between the states shown in Figure 20.05 can be described as follows.

- A new process arrives in memory and a PCB is created; it changes to the ready state.
- A process in the ready state is given access to the CPU by the dispatcher; it changes to the running state.
- A process in the running state is halted by an interrupt; it returns to the ready state.
- A process in the running state cannot progress until some event has occurred (I/O perhaps); it changes to the waiting state (sometimes called the ‘suspended’ or ‘blocked’ state).
- A process in the waiting state is notified that an event is completed; it returns to the ready state.
- A process in the running state completes execution; it changes to the terminated state.

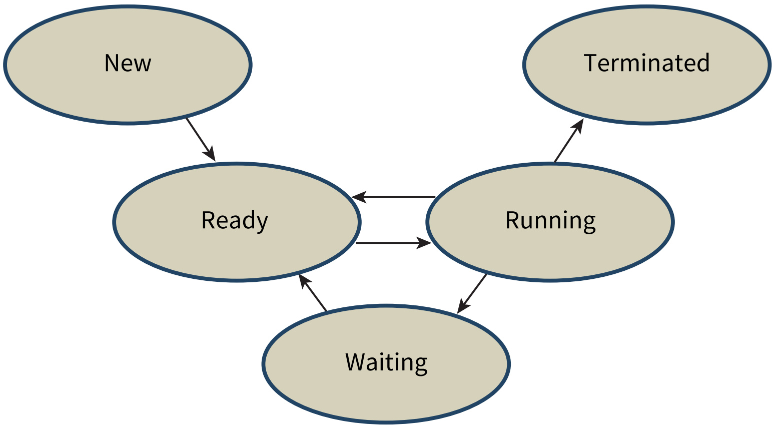


Figure 20.05 The five states defined for a process being executed

It is possible for a process to be separated into different parts for execution. The separate parts are called **threads**. If this has happened, each thread is handled as an individual process.

Interrupts

Some interrupts are caused by errors that prematurely terminate a running process. Otherwise there are two reasons for interrupts.

- Processes consist of alternating periods of CPU usage and I/O usage. I/O takes far too long for the CPU to remain idle waiting for it to complete. The interrupt mechanism is used when a process in the running state makes a system call requiring an I/O operation and has to change to the waiting state.
- The scheduler decides to halt the process for one of several reasons as discussed later under the heading ‘Scheduling algorithms’.

Whatever the reason for an interrupt, the OS kernel must invoke an interrupt-handling routine. This may have to decide on the priority of an interrupt. One required action is that the current values stored in registers must be recorded in the process control block. This allows the process to continue execution when it eventually returns to the running state.

Discussion Point:

What would happen if an interrupt was received while the interrupt-handling routine was being executed by the CPU? Does this require a priority being set for each interrupt?

Scheduling algorithms

Although the long-term or high-level scheduler will have decisions to make when choosing which program should be loaded into memory, we concentrate here on the options for the short-term or low-level scheduler.

A scheduling algorithm can be preemptive or non-preemptive. A preemptive algorithm can halt a process that would otherwise continue running undisturbed. If an algorithm is preemptive it may involve prioritising processes.

The simplest possible algorithm is first come first served (FCFS). This is a non-preemptive algorithm and can be implemented by placing the processes in a first-in first-out (FIFO) queue. The algorithm will be very inefficient if it is the only algorithm employed but it can be used as part of a more complex algorithm.

A round-robin algorithm allocates a time slice to each process and is therefore preemptive, because a process will be halted when its time slice has run out. It can be implemented as a FIFO queue. It normally does not involve prioritising processes. However, if separate queues are created for processes of different priorities then each queue could be scheduled using a round-robin algorithm.

A priority-based scheduling algorithm is more complicated. One reason for this is that every time a new process enters the ready queue or when a running process is halted, the priorities for the processes may have to be re-evaluated. The other reason is that whatever scheme is used to judge priority level it will require some computation. Possible criteria are:

- estimated time of process execution
- estimated remaining time for execution
- length of time already spent in the ready queue
- whether the process is I/O bound or CPU bound.

More than one of these criteria might be considered. Clearly, estimating a time for execution may not be easy. Some processes require extensive I/O, for instance printing wage slips for employees. There is very little CPU usage for such a process so it makes sense to allocate it a high priority so that the small amount of CPU usage can take place. The process will then change to the waiting state while the printing takes place.

20.04 Memory management

The term memory management includes a number of aspects.

- The provision of protected memory space for the OS kernel.
- The loading of a program into memory requires defining the memory addresses for the program itself, for associated procedures and for the data required by the program.
- In a multiprogramming system, this might not be straightforward. The storage of processes in main memory can get fragmented in the same way as happens for files stored on a hard disk. There may be a need for the medium-term scheduler to move a process out of main memory to ease the problem.
- Decisions have to be made about how large a part of memory should be allocated to individual processes sharing memory.

Partitions and segments

An early approach to memory management when different processes were loaded into memory simultaneously was to partition the memory. The aim was to load the whole of a process into one partition. This was wasteful of memory if the process size was less than the partition size. An improvement was dynamic partitioning where the partition size was allowed to adjust to match the process size. The rule of one process per partition remained.

An extension of this idea which allowed for larger processes to be handled was **segmentation**. The large process was divided into segments. Each segment was loaded into a dynamic partition in memory.

There were two factors that limited the efficiency of this approach. The first was that the segments were not constrained to be the same size. The second was that the size of process did not allow all of the segments for one process to be in memory at the same time. Segments had to be moved from disk to memory but then back again to disk when a different segment was needed in memory.

These two factors combined to cause fragmentation both of the memory and of the disk storage. This resulted in degradation in the performance of the system.

Paging and virtual memory

The modern approach is to use **paging**. The process is divided into equal-sized pages and memory is divided into frames of the same size. The secondary storage can also be divided into frames.

It could be possible to load all of the pages into memory at the same time. However, even if this were possible it is usually the case that not all parts of the program are needed at the same time. A large program is likely to have optional routes for the execution.

A special case for the use of paging is when a program is so large that the address space needed for it is larger than the size of the memory. Paging now supports what is known as **virtual memory** management.

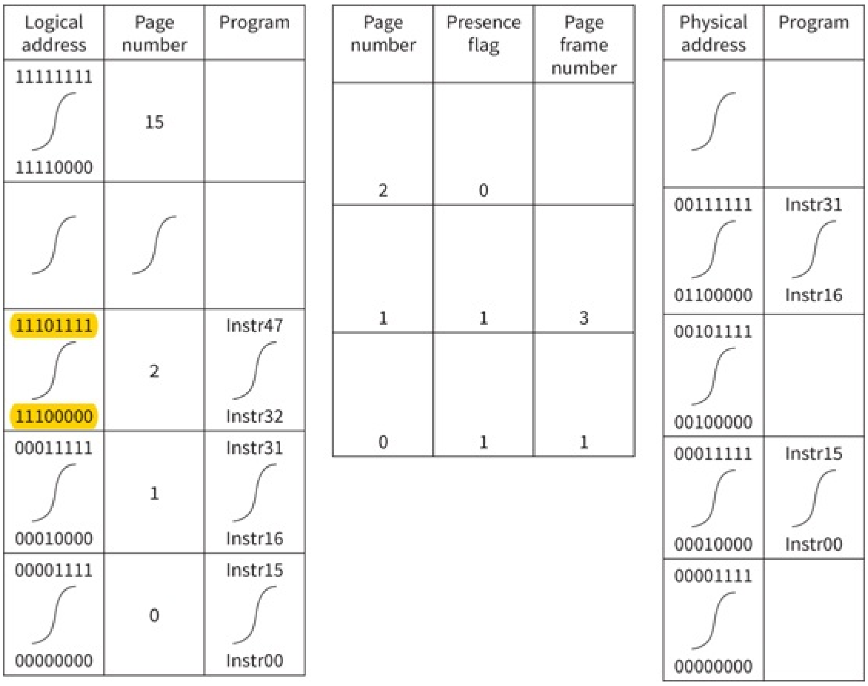
Paging requires the CPU to transfer address values to a memory management unit that allocates a corresponding address on a page. An address has to comprise two parts: the page number plus the offset from the start of the page. The memory management unit functions through the use of a page table. Figure 20.06 shows a very simplified system sufficient to illustrate how this works.

The left-hand part of Figure 20.06 shows a program with 48 instructions. These instructions occupy three pages. The three pages occupy the first 48 logical memory addresses. Because only 8 bits are used to store the logical address there are only 16 pages allowed. The logical address stores the page number in the four most significant bits and the page offset in the four least significant bits.

The right-hand part of Figure 20.06 shows the page frames with physical memory addresses. It shows a scenario where the first two program pages have been loaded into page frames. Note that the page frames used do not have to be adjacent. The centre part of Figure 20.06 shows the contents of the page table for this process (there will be a separate page table for each process in memory). This table has the page number functioning as an index. It has a value for the presence flag indicating whether or not the page is in memory. In the version shown here the third entry shows the page frame number. This might instead have recorded the physical address of the first item in the page frame.

TIP

Remember that you will often see numbering starting from 0.



Logical address	Page number	Program
11111111 ∫ 11110000	15	
∫	∫	
11101111 ∫ 11100000	2	Instr47 ∫ Instr32
00011111 ∫ 00010000	1	Instr31 ∫ Instr16
00001111 ∫ 00000000	0	Instr15 ∫ Instr00

Page number	Presence flag	Page frame number
2	0	
1	1	3
0	1	1

Physical address	Program
∫	
00111111 ∫ 01100000	Instr31 ∫ Instr16
00101111 ∫ 00100000	
00011111 ∫ 00010000	Instr15 ∫ Instr00
00001111 ∫ 00000000	

Figure 20.06 The use of a page table for a paging system

When paging is being used the starting situation is that the set of pages comprising a process are stored on disk. One or more of these pages is loaded into memory when the process is changing to the ready state. When the process is dispatched to the running state, the process starts executing. At some stage, the process will need access to a page that the page table indicates is not in memory. This is called a page fault condition. It is now almost inevitable that, in order to bring in the required page from secondary storage, a page will need to be taken out of memory first. This is when a page replacement algorithm is needed. There are a number of options for this. A simple algorithm would use a first-in first-out method. A more sensible method would be the least-recently-used page but this requires statistics of page use to be recorded.

The system overhead in running virtual memory can be a disadvantage. The worst problem is **disk thrashing**, when part of a process on one page requires another page which is on disk. When that page is loaded it almost immediately requires the original page again. This can lead to almost never-ending loading and unloading of pages. Algorithms have been developed to guard against this never-ending loading and unloading but the problem can still occur.

Extension Question 20.01

Find out information about the virtual memory capacity and page size of a computer system that you use. In what circumstances might these need to be changed?

20.05 Operating system facilities provided for the user

The user interface may be made available as a command line, a graphical display or a voice recognition system. But the function of a user interface is always to allow the user to interact with running programs. When a program involves use of a device, the operating system provides the device driver: the user just expects the device to work. (You might, however, wish to argue that printers do not always quite fit this description.)

The operating system will provide a file system for a user to store data and programs. The user has to choose filenames and organise a directory (folder) structure but the user does not have to organise the physical data storage on a disk. If the user is a programmer, the operating system supports the provision of a programming environment. The operating system allows a program to be created and run without the programmer being familiar with how the processor functions.

When a program is running it can be considered to be a type of user. The operating system provides a set of system calls that provide an interface to the services it offers. For instance, if a program specifies that it needs to read data from a file, the request for the file is converted into a system call that causes the operating system to take charge, find the file and make it available to the program. An extension of this concept is when an operating system provides an application programming interface (API). Each API call fulfils a specific function such as creating a screen icon. The API might use one or more system calls. The API concept aims to provide portability for a program, where a program can run on different operating systems with minimal changes.

20.06 Translation software

Chapter 8 (Section 8.05) provided an overview of how a compiler or an interpreter is used. We will now will consider some details of how a compiler works with a brief reference to the workings of an interpreter. Compiler or interpreter writing is a specialised task carried out by professionals who each will have their own particular methods. So, we will just look at some general principles and illustrations of some of the techniques that are likely to be used.

A compiler can be described as having a 'front end' and a 'back end'. The front-end program performs analysis of the source code and unless errors are found produces an intermediate code that expresses completely the semantics (the meaning) of the source code. The back-end program then takes this intermediate code as input and performs synthesis of object code. This analysis–synthesis model is represented in Figure 20.07.

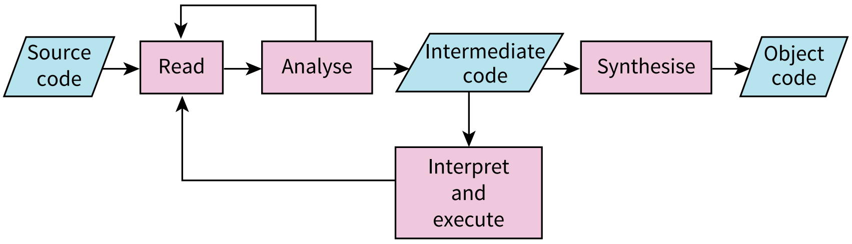


Figure 20.07 Analysis-synthesis model for a compiler

For simplicity, Figure 20.07 assumes no error in the source code. There is a repetitive process in which the source code is read line-by-line. For each line, the compiler creates matching intermediate code. Figure 20.07 also shows how an interpreter program would have the same analysis front-end: In this case, however, once a line of source code has been found to be error free and therefore converted to intermediate code, the line of source code is executed.

Front-end analysis stages

The four stages of front-end analysis, shown in Figure 20.08, are:

- lexical analysis
- syntax analysis
- semantic analysis
- intermediate code generation.

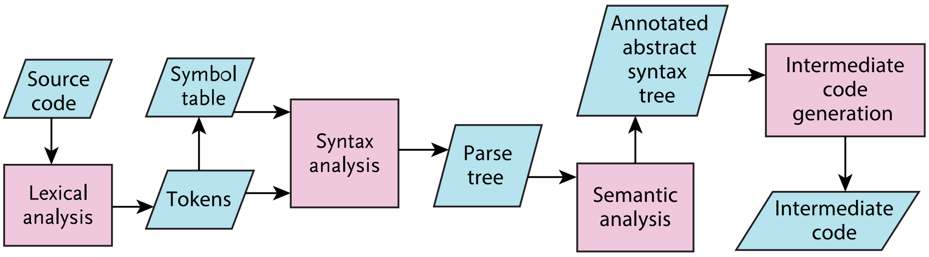


Figure 20.08 Front-end analysis

The source code that is the input data for a compiler or interpreter consists of a sequence of characters. Each meaningful individual character or collection of characters is referred to as a lexeme. A lexeme may be an identifier used by the programmer or may be a keyword, operator or symbol that is defined by the programming language. One approach to lexical analysis is first to remove all white space and all comments then to take each line of source code and identify each lexeme. This is a pattern-matching exercise. It requires the analyser to have knowledge of the components that can be found in a program written in the particular programming language.

For example, the declaration statement:

```
Var Count : integer;
```

would be recognised as containing five lexemes:

```
Var Count : integer ;
```

The assignment statement:

```
PercentMark[Count] := Score * 10
```

would be recognised as containing eight lexemes:

```
PercentMark [ Count ] := Score * 10
```

The lexical analyser must now categorise each lexeme in order to tokenise the line of code. For instance, in the first example, `Var` and `integer` must be recognised as keywords; `Count` recognised as an identifier and : and ; must be recognised as distinct lexemes.

For each identifier recognized there must be an entry made in the **symbol table** (which could have been called the identifier table). The symbol table contains identifier attributes such as the data type, where it is declared and where it is assigned a value. The symbol table is an important data structure for a compiler. Although Figure 20.08 shows it only being used by the syntax analysis program, it is also used by later stages of compilation. Furthermore, most compilers are multi-pass allowing the contents of the symbol table to be frequently upgraded.

Question 20.01

A symbol table is used by an assembler and by a compiler or interpreter. What are the differences between these?

Another table is used to identify the relevant token for each lexeme that is not an identifier so that the lexeme in the line of code can be replaced by a token. In some cases each identifier is also replaced by a token. Whatever scheme is applied the output from lexical analysis is a tokenised version of the source code. A wide variety of formats for the representation of tokens are mentioned in the literature.

Question 20.02

Can you think of any reason why a compiler might wish to tokenise white space rather than removing it all?

Syntax analysis, which is also known as parsing, involves analysis of the program constructs. The results of the analysis are recorded as a syntax or parse tree. Figure 20.09 shows the parse tree for the following assignment statement:

```
y := 2 * x + 4
```

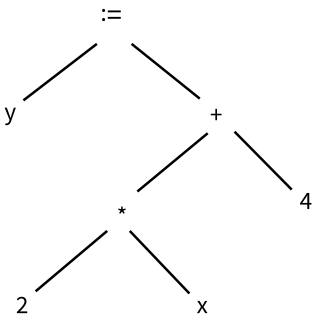


Figure 20.09 Parse tree for an assignment statement

Note that the hierarchical structure of the tree, if correctly interpreted, ensures that the multiplication of 2 by `x` is carried out before the addition of 4.

Semantic analysis is about establishing the full meaning of the code. An annotated abstract syntax tree is constructed to record this information. For the identifiers in this tree an associated set of attributes is established including, for example, the data type. These attributes are also recorded in the symbol table.

An often-used intermediate code created by the last stage of front-end analysis is a three-address code.

As an example the following assignment statement has five identifiers requiring five addresses:

```
y := a + (b * c − d) / e
```

The assignment statement could be converted into the following four statements, each requiring at most three addresses:

```
temp := b * c

temp := temp − d

temp := temp / e

y := a + temp
```

Representation of the grammar of a language

For each programming language, there is a defined grammar. This grammar must be understood by a programmer and also by a compiler writer.

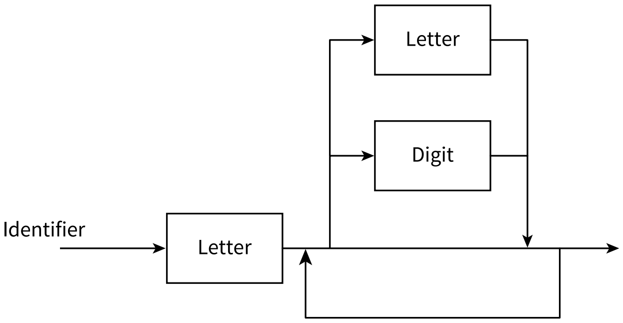


Figure 20.10 Syntax diagram defining an identifier

One method of presenting the grammar rules is a syntax diagram. Figure 20.10 represents the grammar rule that an identifier must start with a letter which can be followed by any combination of none or more letters or digits. The convention used here is that options are drawn above the main flow line and repetitions are drawn below it.

An alternative approach is to use Backus–Naur Form (BNF). A possible format for a BNF definition of an identifier is:

<Identifier> ::= <Letter>|<Identifier><Letter>|<Identifier><Digit>

<Digit> ::= 0|1|2|3|4|5|6|7|8|9

<Letter> ::= <UpperCaseLetter>|<LowerCaseLetter>

<UpperCaseLetter> ::= A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z

<LowerCaseLetter> ::= a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z

The use of | is to separate individual options. The ::= characters can be read as 'is defined as'. Note the recursive definition of <Identifier> in this particular version of BNF. Without the use of recursion the definition would need to be more complicated to include all possible combinations following the initial <Letter>.

A syntax diagram is only used in the context of a language. It has limited use because it cannot be incorporated into a compiler program as an algorithm. By contrast, BNF is a general approach which can be used to describe any assembly of data. Furthermore, it can be used as the basis for an algorithm.

Back-end synthesis stages

If the front-end analysis has established that there are syntax errors, the only back-end process is the presentation of a list of these errors. For each error, there will be an explanation and the location within the program source code.

In the absence of errors, the main back-end stage is machine code generation from the intermediate code. This may involve optimisation of the code. The aim of optimisation is to create an efficient program. One type of optimisation focuses on features that were inherent in the original source code and have been propagated into the intermediate code. As a simple example, consider these successive assignment statements:

```
x := (a + b) * (a − b)
```

```
y := (a + 2 * b) * (a − b)
```

The most efficient code would be:

```
temp := (a − b)
```

```
x := (a + b) * temp
```

```
y := x + temp * b
```

Question 20.03

Check the maths for the efficient code defined above.

Another example is when a statement inside a loop, which is therefore executed for each repetition of the loop, does the same thing each time. Optimisation would place the statement immediately before the loop.

The other type of optimisation is instigated when the machine code has been created. This type of optimisation may involve efficient use of registers or of memory.

Evaluation of expressions

An assignment statement often has an algebraic expression defining a new value for an identifier. The expression can be evaluated by first converting the infix representation in the code to Reverse Polish Notation (RPN). RPN is a postfix representation which never requires brackets and has no rules of precedence.

WORKED EXAMPLE 20.01

Manually converting an expression between RPN and infix

Converting an expression to RPN

We consider a very simple expression:

```
a + b * c
```

The conversion to RPN has to take into account operator precedence so the first step is to convert `b * c` to get the intermediate form:

```
a + b c *
```

We then convert the two terms to give the final RPN form:

```
a b c * +
```

If the original expression had been `(a + b) * c` (where the brackets were essential) then the conversion to RPN would have given:

```
a b + c *
```

Converting an expression from RPN

Consider this more complicated example of an RPN expression:

```
x 2 * y 3 * + 6 /
```

The RPN is scanned until two identifiers are followed by an operator. This combination is converted to give an intermediate form (brackets are used for clarification):

```
(x * 2) y 3 * + 6 /
```

This process is repeated to give the following successive versions:

```
(x * 2)(y * 3) + 6 /
```

```
(x * 2) + (y * 3) 6 /
```

```
((x * 2) + (y * 3)) / 6
```

Because of the precedence rules, some of the brackets are unnecessary; the final version could be written as:

```
(x * 2 + y * 3) / 6
```

WORKED EXAMPLE 20.02

Using a syntax tree to convert an expression to RPN

In the syntax analysis stage, an expression is represented as a syntax tree. The expression `a + b * c` would be presented as shown in Figure 20.11.

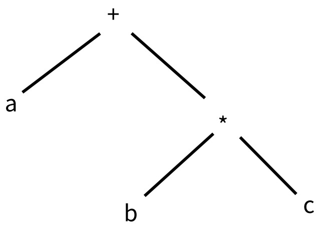


Figure 20.11 Syntax tree for an infix expression

To create this tree, the lowest precedence operator (+) is positioned at the root. If there are several with the same precedence, the first one is used. The RPN form of the expression can now be extracted by a post-order traversal. This starts at the lowest leaf to the left of the root and then uses left–right–root ordering which ensures, in this case, that the RPN representation is:

```
a b c * +
```

WORKED EXAMPLE 20.03

Using a stack with an RPN expression

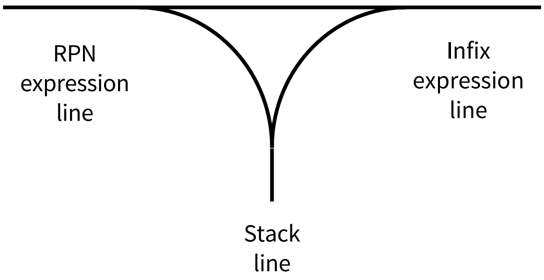


Figure 20.12 Shunting-yard algorithm

To convert an infix expression to RPN using a stack, the shunting-yard algorithm is used (Figure 20.12).

Converting an expression to RPN

The rules of the algorithm are to consider the string of tokens representing the infix expression. These represent the railroad wagons that are to be shunted from the infix line to the RPN line. The tokens are examined one by one. For each one, the rules are as follows.

- If it is an identifier, it passes straight through to the RPN expression line.
- If it is an operator, there are two options.
  - If the stack line is empty or contains a lower precedence operator, the operator is diverted into the stack line.

- If the stack line contains an equal or higher preference operator, then that operator is popped from the stack into the RPN expression line and the new operator takes its place on the stack line.

- When all tokens have left the infix line, the operators remaining on the stack line are popped one by one from the stack line onto the RPN expression line.

Consider the infix expression `a + b * c`. Table 20.02 traces the conversion process. The first operator to enter the stack line is the `+` so when the higher precedence `*` comes later it too enters the stack line. At the end the `*` is popped followed by the `+`.

Infix line	Stack line	RPN line
a + b * c		
+ b * c		a
b * c	+	a
* c	+	a b
C	+ *	a b
	+ *	a b c
	+	a b c *
		a b c * +

Table 20.02 Trace of the conversion process

Had the infix expression been `a * b + c` then `*` would have been first to enter the stack line but it would have been popped from the stack before `+` could enter.

Evaluating an RPN expression

A stack can be used to evaluate an RPN expression. Let's consider the execution of the following RPN expression when x has the value 3 and y has the value 4:

```
x 2 * y 3 * + 6 /
```

The rules followed here are that the values are added to the stack in turn. The process is interrupted if the next item in the RPN expression is an operator. This causes the top two items to be popped from the stack. Then the operator is used to create a new value from these two and the new value is added to the stack. The process then continues. Figure 20.13 shows the successive contents of the stack with an indication of when an operator has been used. The intermediate states of the stack when two values have been popped are not shown.

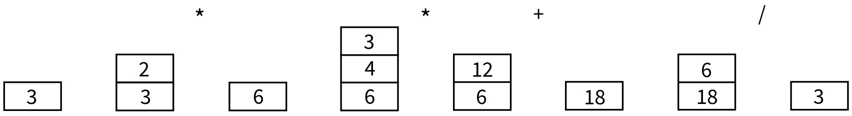


Figure 20.13 Evaluating a Reverse Polish expression using a stack

TASK 20.01

Practise your understanding of RPN.

1 Convert the following infix expressions into RPN using the methods described in Worked Examples 20.01, 20.02 and 20.03:

```
  (x − y) / 4

3 * (2 + x / 7)
```

2 Convert the following RPN expressions into the corresponding infix expressions:

```
4 a b + c + d + e + *

  y 2 ^ z 3 ^ + 6/
```

Note that the caret (^) symbol represents ‘to the power of’.

3 Using simple values for each variable in part 2, use the infix version to evaluate the expression. Then use the stack method to evaluate the RPN expression and check that you get the same result.

The use of RPN would be of little value if the simple processor with a limited instruction set discussed in Chapter 6 was being used. Modern processors will have instructions in the instruction set that handle stack operations, so a compiler can convert expressions into RPN knowing that conversion to machine code can utilise these and allow stack processing in program execution.

Reflection Point:

Have you learned that two data structures illustrated here, the binary tree and the stack, have many different uses in computer science?

Summary

- The operating system provides resource management including scheduling of processes, memory management and control of the I/O system.
- For the user, the operating system provides an interface, a file system and application programming interfaces.
- A modular approach provides a flexible structure for the operating system.
- There are five states for a process: new, ready, running, waiting and terminated.
- A process may be interrupted by an error, a need for an I/O activity or the scheduling algorithm.
- A virtual memory system uses paging and a memory management unit that uses a page table.
- Compiler operation has a front-end program providing analysis and a back-end program providing synthesis.
- Backus–Naur form is used to represent the rules of a grammar.
- Reverse Polish Notation is used for the evaluation of expressions.

Exam-style Questions

1 **a** In a multiprogramming environment, the concept of a process has been found to be very useful in controlling the execution of programs.

i Explain the concept of a process. [2]

ii In one model for the execution of a program, there are five defined process states. Identify **three** of them and explain the meaning of each. [6]

b The transition of processes between states is controlled by a scheduler.

i Identify **two** scheduling algorithms and for each classify its type. [4]

ii A scheduling algorithm might be chosen to use prioritisation. Identify **two** criteria that could be used to assign a priority to a process. [2]

2 **a** Three memory management techniques are partitioning, scheduling and paging.

i Give definitions of them. [3]

ii Identify **two** ways in which they might be combined. [2]

b Some systems use virtual memory.

i Identify which of the techniques in **part a** is used to create virtual memory. [1]

ii Explain **two** advantages of using virtual memory. [4]

iii Explain **one** problem that can occur in a virtual memory system. [2]

3 **a** A compiler is used to translate a program into machine code.

i A compiler is modelled as containing a front end and a back end. State the overall aim of the front end and of the back end. [2]

ii Identify **two** processes which are part of the front end. [2]

iii Identify **two** processes which are part of the back end. [2]

b Complete the following Backus–Naur definition of a signed integer:

i <Digit> ::=

ii <Sign> ::=

iii <Unsigned integer> ::=

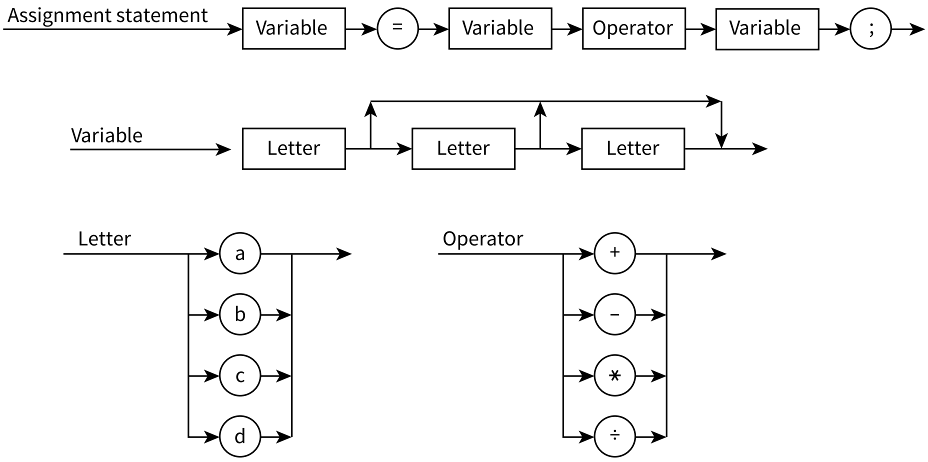
iv <Signed integer> ::= [4]

c Convert the expression `(a + 6) + b / c` into Reverse Polish Notation. [2]

d Convert the Reverse Polish Notation expression `a 3 b * 6 c * - +` into infix notation. [2]

4 The following syntax diagrams, for a particular programming language, show the syntax of:

- an assignment statement
- a variable
- a letter
- an operator



a The following assignment statements are invalid.

Give the reason in each case.

i `a = b + c` [1]

ii `a = b – 2;` [1]

iii `a = dd * cce;` [1]

b Write the Backus-Naur Form (BNF) for the syntax diagrams shown above.

i <assignmentstatement> ::=

ii <variable> ::=

iii <letter> ::=

iv <operator> ::= [6]

c Rewrite the BNF rule for a variable so that it can be any number of letters.

`<variable> ::=` [2]

d Programmers working for a software development company use both interpreters and compilers.

i The programmers prefer to debug their programs using an interpreter.

Give **one** possible reason why. [1]

ii The company sells compiled versions of its programs.

Give a reason why this helps to protect the security of the source code. [1]

Cambridge International AS & A level Computer Science 9608 Paper 31 Q1 June 2015

5 A number of processes are being executed in a computer.

a Explain the difference between a program and a process. [2]

A process can be in one of three states: running, ready or blocked.

b For each of the following, the process is moved from the first state to the second state. Describe the conditions that cause each of the following changes of the state of a process:

i From running to ready

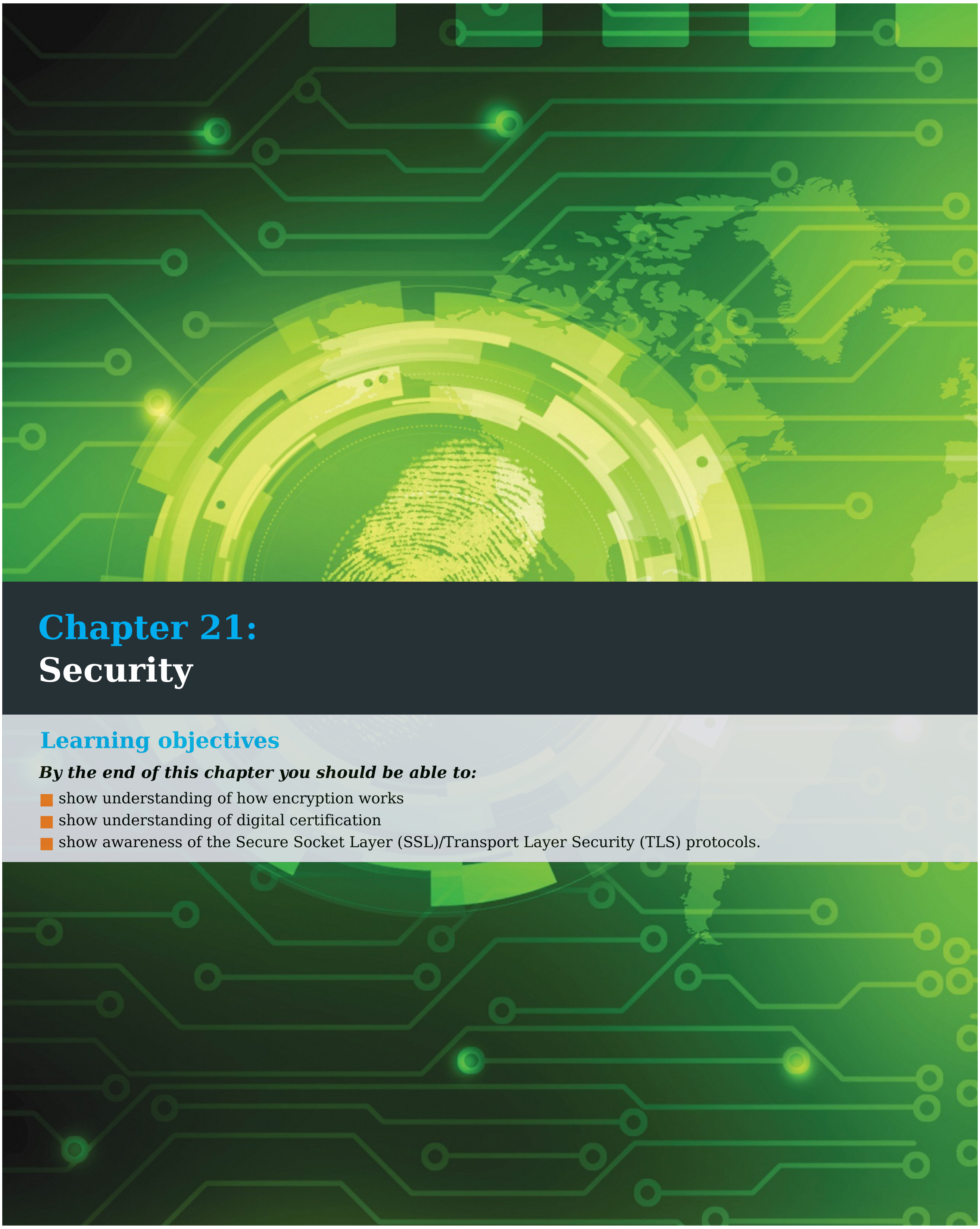
ii From ready to running

iii From running to blocked [6]

c Explain why a process cannot be moved from the blocked state to the running state. [3]

d Explain the role of the high-level scheduler in a multiprogramming operating system. [2]

Cambridge International AS & A level Computer Science 9608 Paper 31 Q6 November 2015



Chapter 21: Security

Learning objectives

By the end of this chapter you should be able to:

- show understanding of how encryption works
- show understanding of digital certification
- show awareness of the Secure Socket Layer (SSL)/Transport Layer Security (TLS) protocols.

21.01 Encryption fundamentals

Encryption can be used as a routine procedure when storing data within a computing system. However, the focus in this chapter is on the use of encryption when transmitting data over a network.

There are three issues that will be considered in this chapter:

- is the encryption algorithm sufficiently robust to prevent the encrypted data being decrypted by some unauthorised third-party?
- how is it possible to ensure that a secret key remains secret?
- how can the receiver of a communication be sure who sent the communication?

The use of encryption is illustrated in Figure 21.01.The process starts with original data referred to as **plaintext**, whatever form it takes. The plaintext is encrypted by an encryption algorithm which makes use of a key. The product of the encryption is **ciphertext**, which is transmitted to the recipient. When the transmission is received it is decrypted using a decryption algorithm and a key to produce the original plaintext.

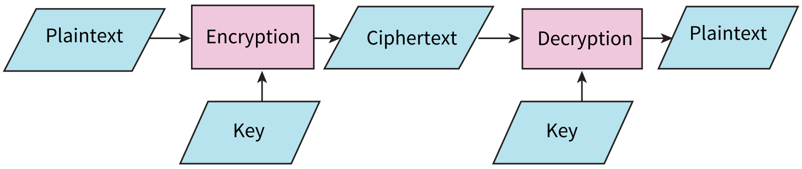


Figure 21.01 Overview of encryption and decryption

Security concerns

There are a number of security concerns relating to a transmission.

- Confidentiality: Only the intended recipient should be able to decrypt the ciphertext.
- Authenticity: The receiver must be certain who sent the ciphertext.
- Integrity: The ciphertext must not be modified during transmission.
- Non-repudiation: Neither sender nor receiver should be able to deny involvement in the transmission.
- Availability: Nothing should happen to prevent the receiver from receiving the transmission.

This chapter will consider only confidentiality, authenticity and integrity.

The confidentiality concern arises because a message could be intercepted during transmission and the contents read by an unauthorised person. The concern about integrity reflects the fact that the transmission might be interfered with deliberately but also that there might be accidental corruption of the data during transmission.

Encryption methods

The fundamental principle of encryption is that the encryption algorithm must not be a secret: it must be in the public domain. In contrast, an encryption key must be secret. However, there are two alternative approaches. One is **symmetric key encryption**, and the other is **asymmetric key encryption** also known as public key encryption.

In symmetric key encryption there is just one key. This key is a secret shared by the sender and the receiver of a message. The sender uses the encryption algorithm together with the key to encrypt some plaintext. The receiver decrypts the ciphertext using the same key.

The issue with symmetric key encryption is delivery of the secret key. The sender needs the key to encrypt but how can the key be securely delivered to the receiver to allow decryption?

In asymmetric key encryption two different keys are used, one for encryption and the other one for decryption. Only one of these is a secret.

If asymmetric encryption is to be used the process is initiated by someone in possession of two keys. One of these is a public key which is sent to anyone who is going to partake in an encrypted communication. The other is a secret private key which is never sent to anyone. Having a means of secure transmission of a secret key is no longer an issue.

The most likely scenario is that the holder of the two keys wishes to receive a transmission. In this case a sender uses the public key to encrypt some plaintext and sends the ciphertext to the receiver. The receiver is now the only person who can decrypt the message because the private and public keys are a matched pair. The public key can be provided to any number of different people allowing the receiver to receive a private message from any of the different people. There are two points to note here.

- If two people require two-way communication, both communicators need a private key and must send the matching public key to the other person.
- There are two requirements to ensure confidentiality should the transmission be intercepted and the message extracted: the encryption algorithm must be complex and the number of bits used to define the key must be large.

Question 21.01

One method used by an unauthorised person attempting to decrypt a message is called a brute-force attack where all possible values for the key are tried. Calculate how long it would take to try all possibilities for a 64-bit or 128-bit key, assuming each attempt took 1 second.

The above account does not completely answer the question of how encryption works. The missing factor is an organisation to provide keys and to ensure their safe delivery to individuals using them. This will be discussed in the next section.

21.02 Digital signatures and digital certificates

Using asymmetric encryption, the decryption–encryption works if the keys are used the other way round. An individual can encrypt a message with a private key and send this to a recipient who has the corresponding public key and who can then use this to decrypt the received ciphertext. This approach would not be used if the content of a message was confidential because anyone might be in possession of the public key. However, it could be used if it was important to verify who the sender was. Only the sender has the private key and the public key only works with that one specific private key. Therefore, if the recipient finds that the decryption is successful, the message has in effect been received with a digital signature identifying the sender.

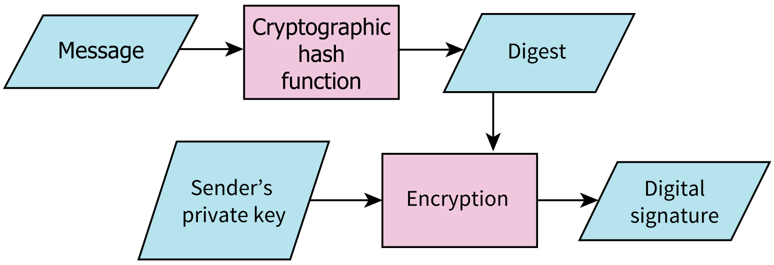


Figure 21.02 Sender using a one-way hash function to send a digital signature

There is a disadvantage in using this method of applying a digital signature: it is associated with an encryption of the whole of a message. An alternative is for the sender to use a public cryptographic one-way hash function which creates a number that is uniquely defined for the particular message, called a 'digest'. The process at the sender's end of the transmission is outlined in Figure 21.02. The private key is used to encrypt the digest. The encrypted digest is the digital signature. The message can be transmitted as plaintext together with the encrypted digest as a separate file. Because the digest is much smaller than the whole message the encryption and the transmission are faster processes than if the whole message were encrypted.

The processes that take place at the receiver end are outlined in Figure 21.03. The same public one-way hash function is used to create a digest from the received message. Then the encrypted version of the original digest is decrypted using the public key.

If the two digests are identical the receiver can be confident that the message is authentic and has been transmitted unaltered.

Note that the digital signature is different each time this process is used. This is because the digest is uniquely defined by the hash function being applied to that particular message.

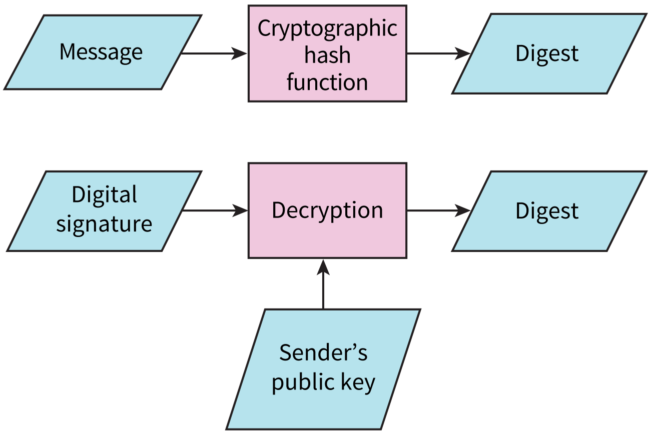


Figure 21.03 Receiver checking that the received transmission is authentic and unchanged

However, the authenticity only confirms to the receiver that the message was sent from the person who had sent them the public key. It does not consider the fact that someone might create a public key and pretend to be someone else.

Therefore, a more strict way of ensuring authentication is needed. This can be provided by a Certification Authority (CA) as part of a Public Key Infrastructure (PKI).

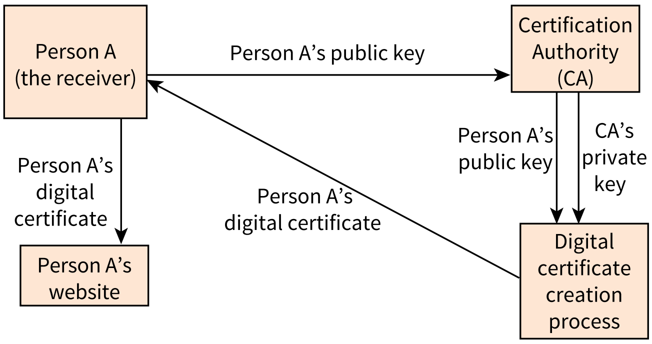

Person A (the receiver)
Person A's public key
Certification Authority (CA)
Person A's public key
CA's private key
Person A's digital certificate
Person A's digital certificate
Person A's website
Digital certificate creation process

Figure 21.04 Processes involved in obtaining a digital certificate

Let's consider a would-be receiver who has a public–private key pair. The receiver wants to be able to receive secure messages from other individuals, and these individuals want to be confident about the identity of the receiver. The public key must be made available in a way that ensures authentication. The steps taken by the would-be receiver to obtain a digital certificate to allow safe public key delivery are illustrated in Figure 21.04. The process can be summarised as follows.

1. An individual (person A) who is a would-be receiver and has a public–private key pair contacts a local CA.
2. The CA confirms the identity of person A.
3. Person A's public key is given to the CA.
4. The CA creates a public-key certificate (a digital certificate) and writes person A's public key into this document.
5. The CA uses encryption with the CA's private key to add a digital signature to this document.
6. The digital certificate is given to person A.
7. Person A posts the digital certificate on a website.

Figure 21.04 shows person A placing the digital certificate on that person's website but another option is to post it on a website designed specifically for keeping digital certificate data.

Anyone who wishes to extract the public key from the certificate has to use the CA's public key.

For this overall process to work there is a need for standards to be defined regarding the public key infrastructure and the production of the digital certificate. As ever, the name for the standard, X.509, is not very memorable.

TIP

There are two similar processes that have been discussed. In one case someone with a private key sends a public key to someone else. In another case the CA sends a digital certificate containing a public key. Try not to confuse these two.

The following are a few notes to summarise the options available.

- The starting position is someone who has a public–private key pair which are associated with a specific asymmetric key encryption algorithm.
- This person could just make the public key available to anyone who is going to be either a sender or a receiver.
- For optimum security the person instead sends the public key to a Certification Authority.
- The Certification Authority creates a digital certificate which contains the public key with proof of

the ownership of the public key.

- Anyone wishing to use the public key obtains it from this digital certificate.
- A message encrypted with the public key could be sent to the owner of the private key.
- A message encrypted with the private key could be sent to anyone having the public key.
- The owner of the private key could use it to create a digital signature that could be used to authenticate an email as was suggested in Chapter 9 (Section 9.03).

21.03 Symmetric key encryption methods

For many years the Data Encryption Standard (DES) was the normal choice for symmetric key encryption. As the weakness of DES became a problem, Triple DES took its place. In 2001 the Advanced Encryption Standard (AES) was introduced as a superior approach. For education purposes only a simplified DES (S-DES) was introduced which allowed the sort of operations performed in encryption to be better understood. The following is an overview of S-DES.

S-DES is an example of a block cipher which means encrypting blocks of bits. In S-DES 8-bit blocks are encrypted. A 10-bit key is chosen. The first stage is to create two 8-bit keys from the 10-bit key. The first step in this first stage is a permutation (reordering of digits) which can be illustrated by the following.

Suppose that the 10-bit key is chosen to be	0101010101
when subjected to a permutation which can be represented by	3 5 2 7 4 10 1 9 8 6
it becomes	0010110011

The numbering of the bit positions is read from left to right so the new 10-bit version has the old position 3 value followed by the old position 5 value and so on. The next step is to apply a circular left shift to the first 5 bits and to the last 5 bits. This produces 0101000111. Finally, the first of the two 8-bit keys to be used in the encryption is created using the permutation 6 3 7 4 8 5 10 9. In our example this key is 00011011. A slightly modified version of this is used to create a second 8-bit key.

The second stage is the encryption, which is a five step process.

1 An initial permutation.

2 Application of a function using the permuted code and the first encryption key.

3 A switch of the first and last 4-bit parts.

4 A repeat application of the function but this time with the second encryption key.

5 A final permutation using the reverse of the initial permutation sequence.

The decryption by the receiver of a transmission uses the same generated 8-bit keys and follows the reverse of the above process.

The AES standard defines the block length as 128 bits. The user can choose to use 128, 192 or 256 bits for the key. The chances of the key being identified from the transmitted ciphertext are small. The main concern is the safety of the method used to provide the key to both sender and receiver.

Extended Question 21.01

Would you like to investigate S-DES further? You could attempt an encryption and decryption.

21.04 Public key encryption methods

RSA (Rivest-Shamir-Adleman), the usual method for public key encryption, is named after the three people who created it. The major features of the method are the key generation algorithm and the encryption function.

The key generation can be summarised as follows.

1. Two very large prime numbers p and q are chosen and their product n is calculated.
2. The product (p-1)(q-1) is calculated.
3. A prime number e less than (p-1)(q-1) and not a factor of it is chosen (65537 is the usual choice).
4. Another number d is found which satisfies the condition that the product of d times e when divided by (p-1)(q-1) gives a remainder of 1.
5. The public key becomes the pair (n,e).
6. The private key becomes the pair (n,d).

The security of the algorithm depends on the fact that finding factors of a very large number is not feasible within any reasonable time scale. Computing n from p and q is straightforward but deducing p and q given n is not!

The encryption works on numbers so a text to be encrypted must first have the characters replaced by numbers according to a sensible coding scheme. If such a number x is to be encrypted as y then y is calculated so that the following relationship holds:

$$y = x^e \bmod n$$

A similar relationship involving d rather than e is used for decryption.

Public key encryption is inherently more secure than symmetric key encryption but the algorithms are not as fast. It is quite common for public key encryption to be used to deliver securely a key that can then be used for symmetric key encryption.

21.05 SSL and TLS

When we access a website, we have two concerns. The first is whether or not the website is genuine. The second is whether we can transfer sensitive personal data to the website, for example to buy a product offered for sale on the website. The Secure Socket Layer (SSL) protocol was created to give assurance to a website user when a client–server application is underway. As described in Chapter 17 (Section 17.04), the interface between an application and TCP uses a port number. In the absence of a security protocol, TCP services an application using the port number. The combination of an IP address and a port number is called a ‘socket’. When the Secure Socket Layer protocol is implemented it functions as an additional layer between TCP in the transport layer and the application layer. When the SSL protocol is in place, the application protocol HTTP becomes HTTPS.

The following are some facts concerning SSL.

- Although SSL is referred to as a protocol, it is in fact a protocol suite.
- There is a Record Protocol that deals with the format for data transmission.
- There is also a Handshake Protocol responsible for security.
- The operation of SSL happens without any action from the user.
- The starting point for SSL implementation is a connection between the client and the server being established by TCP.
- The client browser then invokes the Handshake Protocol from the SSP suite.
- The Handshake Protocol requests from the server its SSL certificate which is a digital certificate confirming its identity.
- The server sends this SSL certificate plus its public key.
- The browser uses this public key to encrypt a key which is to be used as a one-off session key for symmetric key encryption to be used for the data transfer during the session.
- There may also be a need at this time to agree which encryption algorithms are to be used.

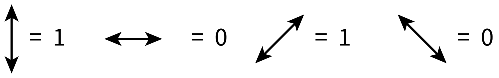
SSL was originally a proprietary protocol. However, it was taken over by the Internet Engineering Task Force (IETF) in order for a standardised version to be produced. This progressed to version 3.0. When the IETF realised that an improved version was required it decided that a new name was appropriate. Transport Layer Security (TLS) is an upgraded version of SSL recommended for use because of some security concerns with the use of SSL. Despite this SSL is still in general use.

Discussion Point:

The use of encryption has always been a controversial subject. There are two important aspects to this. The first is whether powerful, unbreakable encryption algorithms should be made available to the public. The second relates to the key escrow scheme, which allows governments access to all secret keys. You might wish to consider how the content here has relevance to some of the topics in Chapter 9 and Chapter 10.

21.06 Quantum cryptography

Quantum mechanics provides fundamental laws of physics applicable to the behaviour of particles. The particles that transmit light are called photons. Photons demonstrate wave behaviour, so that each photon appears to vibrate in a particular direction at right angles to its direction of travel. The direction each photon vibrates in is called its polarisation, and is represented in a diagram as a double-ended arrow. A photon can be created with a specific polarisation to represent a value for a bit. If we allow four possibilities for the state of polarisation there are two ways to represent a 1 and two ways to represent a 0. This can be illustrated as follows:



This scheme can be used to enable a sender and receiver to create a 'shared secret' code consisting of a number of bits. Table 21.01 illustrates the process.

Bit values sent	1	0	1	1	0	0	0	1	0	1
Polarisation basis	+	×	×	+	+	×	×	+	×	+
Polarisation chosen by receiver	×	+	+	+	×	×	+	×	×	+
Bit values confirmed				1		0			0	1

Table 21.01 Creation of a 'shared secret' code

In Table 21.01 the first row indicates the bit sent, the second row shows the basis used for this with + representing the one using vertical and horizontal and x representing the diagonal pair. The sender chooses the bit pattern at random and also the polarisation basis for each value at random. The third row shows the receiver's choice for the polarisation basis for each value. Again this is a random choice. Following the transmission the sender informs the receiver about the polarisation basis used for each value. The receiver responds by saying which ones were chosen to match. For these matches there is now a stored value for a bit. In the example shown a 'shared secret' code 1001 has been created.

The above scheme has been incorporated in Quantum Key Distribution (QKD) systems. Earlier in this chapter it was mentioned how a key might be transferred using asymmetric key encryption for subsequent use in symmetric key encryption. QKD offers an alternative for the key transfer. The key is still to be used in the normal way. The advantage of QKD is that the transfer does not involve defined values just photons. Anyone trying to intercept the flow of photons in an attempt to discover their polarisation will by the laws of quantum mechanics destroy the photons. A photon cannot be detected and measured then sent on again. The main drawback of quantum cryptography is that it cannot be implemented using standard communication media. It requires a dedicated, special purpose 'quantum channel' between sender and receiver. The costs of providing this are very high so routine use is unlikely.

There are ambitious hopes for the future of quantum computing but progress is slow. QKD is one of the few examples where there has been significant success evidenced.

TASK 21.01

The concept of a 'shared secret' has been used in traditional encryption schemes. An example is the Diffie–Hellman key agreement method. Investigate the principles behind how this works.

Reflection Point:

Chapter 21: Security has some difficult concepts. Have you made sure that you understand the overriding principles and definitions used?

Summary

- Encryption converts plaintext to ciphertext; decryption reverses the process.
- The five main security concerns when transmitting messages are: confidentiality, authenticity, integrity, non-repudiation and availability.
- Alternatives for encryption are symmetric, using one key, or asymmetric, using two different keys.
- Authentication can be achieved using a digital signature and a digital certificate.
- A digital certificate is provided by a certification authority within a public key infrastructure.
- DES and AES are examples of symmetric key encryption.
- RAS is an important asymmetric key method.
- Secure Socket Layer (SSL) which became Transport Layer Security (TLS) provides security when accessing a website.
- Quantum Key Distribution systems use polarised photons.

Exam-style Questions

1 **a** When transmitting data across a network three concerns relate to: confidentiality, authenticity and integrity.

Explain each of these terms. [4]

b Encryption and decryption can be carried out using a symmetric or an asymmetric key method.

Explain how keys are used in each of these methods. You are not required to describe the algorithms used. Your account must include reference to a public key, a private key and a secret key. [6]

c Digital signatures and digital certificates are used in message transmission.

Give an explanation of their use. [5]

2 Secure socket layer (SSL) and its upgraded version named Transport Layer Security (TLS) is described as a protocol suite.

a Explain the meaning of the description 'protocol suite'. [3]

b Describe the type of activity where SSL or TLS would be used. [4]

c Explain how digital certificates are used in the protocol suite. [3]

d Explain how encryption keys are used in the protocol suite. [5]

3 Digital certificates are used in Internet communications. A Certificate Authority (CA) is responsible for issuing digital certificates.

a Name **three** data items present in a digital certificate. [3]

b The method of issuing a digital certificate is as follows.

i A user starts an application for a digital certificate using their computer. On this computer a key pair is generated. This key pair consists of a public key and an associated private key.

ii The user submits the application to the CA. The generated .... (i) .... key and other application data are sent. The key and data are encrypted using the CA's .... (ii) .... key.

iii The CA creates a digital document containing all necessary data items and signs it using the CA's .... (iii) .... key.

iv The CA sends the digital certificate to the individual.

In the above method there are three missing words. Each missing word is either 'public' or 'private'.

State the correct word. Justify your choice. [6]

c Alexa sends an email to Beena.

Alexa's email program:

- produces a message digest (hash)
- uses Alexa's private key to encrypt the message digest
- adds the encrypted message digest to the plain text of her message
- encrypts the whole message with Beena's public key
- sends the encrypted message with a copy of Alexa's digital certificate.

Beena's email program decrypts the encrypted message using her private key.

i State the name given to the encrypted message digest. [1]

ii Explain how Beena can be sure that she has received a message that is authentic (not corrupted or tampered with) and that it came from Alexa. [2]

iii Name **two** uses where encrypted message digests are advisable. [2]

Cambridge international AS & A Level Computer Science 9608 paper 31 Q2 June 2016

4 Both clients and servers use the Secure Socket Layer (SSL) protocol and its successor, the Transport Layer Security (TLS) protocol.

a **i** What is a protocol? [2]

ii Name the client application used in this context. [1]

iii Name the server used in this context. [1]

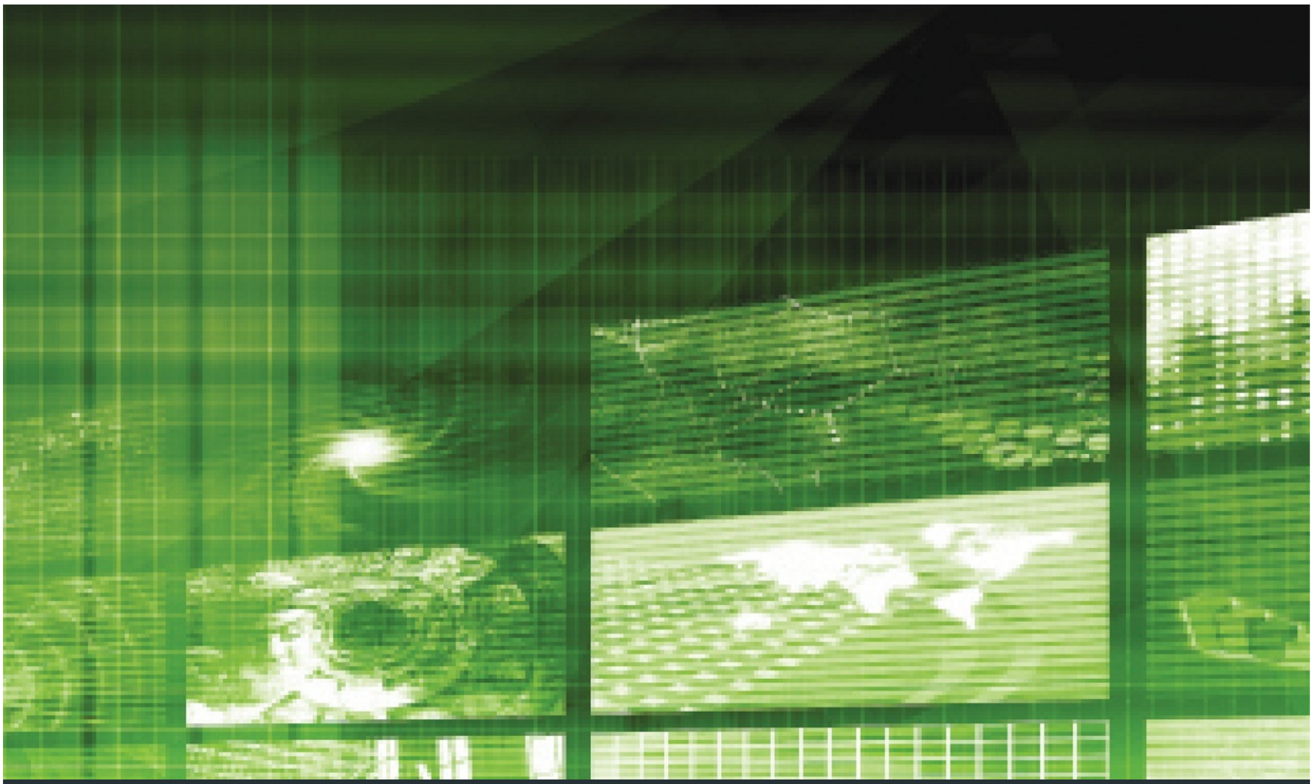
iv Identify **two** problems that the SSL and TLS protocols can help to overcome. [2]

b Before any application data is transferred between the client and the server, a handshake process takes place. Part of this process is to agree the security parameters to be used.

Describe **two** of these security parameters. [4]

c Name **two** applications of computer systems where it would be appropriate to use the SSL of TLS protocol. These applications should be different from the ones you named in **part (a)(ii)** and **part (a)(iii)**. [2]

Cambridge international AS & A Level Computer Science 9608 paper 32 Q4 November 2016



Chapter 22:
Artificial Intelligence (AI)

Learning objectives

By the end of this chapter you should be able to:

- show understanding of how graphs can be used to aid Artificial Intelligence (AI)
- show understanding of how artificial neural networks have helped with machine learning
- show understanding of Deep Learning, Machine Learning and Reinforcement Learning and the reasons for using these methods
- show understanding of back propagation of errors and regression methods in machine learning.

22.01 An overview

It is not easy to define ‘Artificial Intelligence’. A key issue is the definition of intelligence. For example, you could argue that a person needs intelligence to do mental arithmetic, such as 43 × 13. You could use a calculator to get the answer, though, and would not describe the calculator as having artificial intelligence. This means that a definition such as:

> Artificial intelligence involves the automation of intelligent behaviour.

is not entirely satisfactory.

There is agreement that AI is a part of computer science. It is also clear that the subject has many distinct sub-sections some of which will be considered in this chapter. The conclusion is that a vague definition is best. For example:

> Artificial Intelligence is concerned with “how to make computers do things at which, at the moment, people are better.”

(E. Rich. Artificial Intelligence. McGraw-Hill, 1983)

22.02 How graphs can be used in AI

A graph is a collection of nodes or vertices between which there can be edges. Each node has a name. An edge can have an associated label which is a numerical value. An example is presented in Figure 22.01.

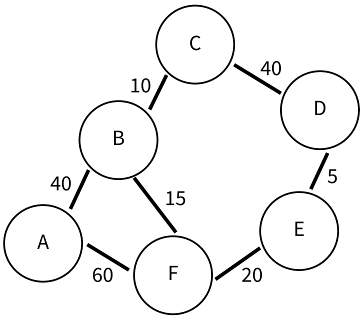


Figure 22.01 An example of a graph with labelled edges

A graph can be used to represent a variety of scenarios. One common representation is that the nodes represent places and the edge labels represent the distances between those places. Edges are only included in the graph when there is a route available for direct travel between the pair of nodes. Such graphs can, for example, find the shortest route between two nodes that are not adjacent to each other.

We could use our intelligence to find the shortest route between node A and node D by considering all of the possible routes and calculating the overall distance for each route. Using Figure 20.01, we would calculate the following values:

For A to B to C to D overall distance is 40 + 10 + 40 = 90

For A to B to F to E to D overall distance is 40 + 15 + 20 + 5 = 80, which is the shortest

For A to F to E to D overall distance is 60 + 20 + 5 = 85

For A to F to B to C to D overall distance is 60 + 15 + 10 + 40 = 125

For a graph containing 100 nodes this could be quite time consuming. Fortunately, a number of artificial intelligence algorithms have been developed to solve this type of problem.

Question 22.01

Can you think of two alternatives for what the graph in Figure 22.01 might represent? For each of these, state what the values shown as edge labels would represent.

Dijkstra’s algorithm

This algorithm finds the shortest path to each of the other nodes starting from one of the nodes.

The following is a Structured English design for a simplified version of the algorithm:

Identify the source node (S) where the path starts.

Create an empty set called the ShortestPath set.

Create another set called RemainingNodes and put all of the nodes into this including the source node (S).

Create a record that stores:

node names

calculated values for the distance to the node from the source node

the sequence of nodes in the route to the node.

Set the distance value for the source node S to be 0.

Set the distance value for all other nodes to be INFINITY where this is to be set as a large value greater than any value that will be calculated.

While the ShortestPath set does not include all of the nodes do the following:

Pick the node (N) from the RemainingNodes set that has the lowest distance value.

Move this node into the ShortestPath set.

For each node in the RemainingNodes set that is adjacent to N:

Calculate a new distance value by adding the value given by the label of the edge connecting the two nodes to the already stored distance for N.

If this value is less than the value currently stored replace this stored value by the new one that has been calculated.

If a new value has been stored enter the sequence of nodes used to obtain this value.

Table 22.01 shows how the algorithm progresses for the graph shown in Figure 22.01. The node names are presented in red when they are still in the RemainingNodes set. At each stage the node N is represented in black. The distance and route data is presented in grey when it is no longer changeable because the node has been moved to the ShortestPath set.

Content of the ShortestPath set	**Content of the record**					
{}	A	B	C	D	E	F
	0	∞	∞	∞	∞	∞
{A}	A	B	C	D	E	F
	0	40	∞	∞	∞	60
	A	A-B				A-F
{A,B}	A	B	C	D	E	F
	0	40	50	∞	∞	55
	A	A-B	A-B-C			A-B-F
{A,B,C}	A	B	C	D	E	F
	0	40	50	90	∞	55
	A	A-B	A-B-C	A-B-C-D		A-B-F
{A,B,C,F}	A	B	C	D	E	F
	0	40	50	90	75	55
	A	A-B	A-B-C	A-B-C-D	A-B-F-E	A-B-F
{A,B,C,E,F}	A	B	C	D	E	F
	0	40	50	80	75	55
	A	A-B	A-B-C	A-B-F-E-D	A-B-F-E	A-B-F
{A,B,C,D,E,F}	A	B	C	D	E	F
	0	40	50	80	75	55
	A	A-B	A-B-C	A-B-F-E-D	A-B-F-E	A-B-F

Table 22.01 Dijkstra’s algorithm applied to the graph in Figure 22.01

Note the general feature that some data don’t change once defined as is the case for nodes C and E, but in other cases such as for nodes D and F the data has to be changed at a later stage.

A* algorithm

Dijkstra's algorithm finds the best route from one node to all of the remaining nodes. In many cases the only requirement is to find the best route from one node to just one other node. It would be possible to modify the Dijkstra algorithm to make it stop once the optimum route to the target destination node had been established. However, the modified algorithm would still be likely to carry out far more calculations than were necessary. In particular, the algorithm could initially be exploring routes that were not in the right direction.

The A* algorithm is a modification of the Dijkstra algorithm designed to improve matters. The design of the A* algorithm is different in that the following step:

> Calculate a new distance value by adding the value given by the label of the edge connecting the two nodes to the already stored distance for N.

is expanded to the following:

> Calculate a new distance value by adding the value given by the label of the edge connecting the two nodes to the already stored distance for N.
>
> Calculate an estimated value for the distance of N from the destination node and add this to the new distance value.

The extra calculation of the distance still to be travelled requires the use of a heuristic function. When this function is chosen it must guarantee that any estimated value will be lower than the actual value.

As an example of the use of the A* algorithm we can consider a simple example of finding the shortest route between two towns using the existing roads. Figure 22.02 shows a basic map showing the locations of seven towns identified as A – G.

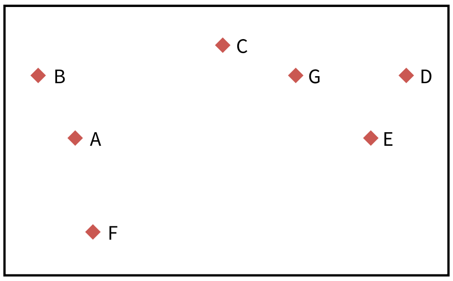


Figure 22.02 A simple map showing the positions of seven towns.

This map is included because it has been drawn by using a pair of x-y coordinates defining the position of each town. These coordinates are the basis of the heuristic function to be used.

Pythagoras's theorem can be used to calculate the direct distance between two positions as follows:

> Direct distance=((difference in × coordinates)2+(difference in y coordinates)2)

This calculated distance must be less than any actual distance travelled using existing roads. The coordinates used are shown in Table 22.02.

A	B	C	D	E	F	G
40,20	30,30	80,35	130,30	120,20	45,5	100,30

Table 22.02 The x-y coordinates for the towns A – G

The graph for this problem is shown in Figure 22.03. This is drawn as before using edges to define which towns have an existing road linking the town to another town. The label for each edge shows the distance of travel along that road.

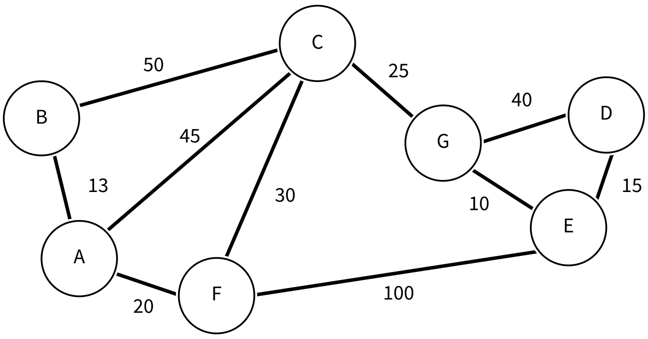


Figure 22.03 The graph used to illustrate the A* algorithm.

It must be understood that there are a variety of versions of the A* algorithm. The application of the algorithm demonstrated here does not match any specific version found in the literature.

The algorithm is documented by the following Structured English.

Create three lists: the initial list, the open list and the closed list.
Insert an empty record for each node in the graph into the initial list.
Store the x and y coordinates of the node in each record of the initial list.
Initialise the value for distance travelled to zero in each record of the initial list.
Create a look-up table that includes an entry for each pair of nodes that have a direct connecting road.
Store the distance of travel along that road for each pair of nodes in the table.
Identify the target node for the travel.
Identify the starting node for the travel and copy the record for this node into the open list.
Delete the record for the starting node from the initial list.

Now recursively apply the following algorithm until all possibilities have been examined:

For the parent node in the open list identify all adjacent nodes in the initial list.
Copy the record for each of the adjacent nodes into the open list.
Delete the records for the adjacent nodes from the initial list.
Check if the target node is now in the open list.

If the target node is not in the open list continue with the following:

> For each adjacent node now added to the open list use the value in the look-up table to determine the distance of travel from the parent node.
> If options exist calculate a value for each route and choose the smallest value.
> For each adjacent node update the value stored for distance travelled in the record.
> For each adjacent node update the path sequence stored in the record.
> For each adjacent node use the heuristic function to calculate a value for estimated distance to travel.
> For each adjacent node calculate a value for estimated total distance from start node to target node by adding the value for distance travelled to the value for estimated distance to travel.
> Identify the adjacent node with the lowest value for total distance from start mode to target node. Leave the record for this node in the open list as a parent node. Copy the records for the remaining nodes into the closed list. Delete these records from the open list.
> Continue with the next iteration of the recursion.

If the target node is in the open list, do the following for this node only:

> Use the value in the look-up table to find the actual distance from the parent node to the target node.
> Calculate the total distance travelled.
> If the record for the target node has zero for the value for distance travelled enter the value now calculated and enter the path sequence into the record.
> If the record already has a value for distance travelled compare the new value and if this indicates a shorter route update the value in the record for distance travelled and enter the new path sequence.

Copy the record for the target node into the initial list.
Delete the record for the target node from the open list.
If there is another adjacent node in the open list continue with the actions listed above for when the target node is not in the open list.
If the open list is now empty rewind to check nodes in the closed list that have not been included in previous path sequences. For each of these calculate a value for estimated total distance from start node to target node. If this estimate is greater than the value stored for distance travelled in the target mode record continue to rewind. If the estimate is less than this stored value copy the record for the node into the open list as a parent node and continue with the actions listed above for when the target node is not in the open list.

We can consider how the algorithm will progress if the start node is chosen to be town A and the target node to be town D.

With A the parent node, B, C and F become the adjacent nodes whose records are brought into the open list. The calculations are:
For B distance travelled = 13, estimated further distance is 100, total is 113
For C distance travelled = 45, estimated further distance is 51, total 96
For F distance travelled = 20, estimated further distance is 89, total 109
(Note that there are two calculations that have to be made for C and F but in each case the direct route is the shortest.)
B and F are now moved to the closed list and C becomes the new parent. The only adjacent node in the initial list is G which is brought into the open list.
G becomes the parent for the adjacent nodes D and E.
D is identified as the target node. The actual distance of travel from A to D using this route is calculated as 110. When the data has been stored in the record for D this is moved to the initial list.

E is the only adjacent node left in the open list so this automatically becomes the new parent node. The only node in the initial list is D which is again brought into the open list as an adjacent node. The actual distance of travel from A to D is now calculated as 105. This is a lower vale than the one previously calculated so this data is stored in the record for node D. This record is then moved once again into the initial list.

The rewinding will now consider nodes B and F. The calculations for these were carried out earlier. The estimated distances for A to D via B and for A to D via F were found to be 109 and 113 respectively. Both of these values are greater than the actual distance found for a different route so these nodes can be ignored. The problem has been solved. The shortest route has been found to be A-C-G-E-D with a total distance of 105.

Discussion Point:

Do you know what a heuristic function is?

Discussion Point:

The techniques discussed here can be used for path-finding when there are obstacles. Can you find some information about how this is approached?

22.03 Machine learning

The requirements for **machine learning** can be summarised as:

- a computer-based system has a defined task or tasks to perform
- knowledge is acquired through the experience of performing the tasks
- as a result of this experience and the knowledge gained the performance of future tasks is improved.

The ability to learn from experience is an indication of intelligence. Machine learning is therefore one of the many individual approaches defined under the umbrella of artificial intelligence.

There are a number of ways to describe how the learning can take place. Three of these will be considered here.

In **unsupervised learning** the system has to draw its own conclusions from its experience of the results of the tasks it has performed. For this, algorithms are needed that can organise or categorise the knowledge acquired. An example is where 'conceptual clusters' are identified which are based on a hierarchical framework. In this approach the knowledge is initially all placed in the root of a tree structure. Then, depending on attributes of the knowledge, selected groups are moved into branches of the tree.

Nowadays unsupervised learning is a dominant activity. Powerful computer systems having access to massive data banks are regularly used to make decisions based on previous actions recorded. We all have our activity on the world wide web recorded and stored. This stored data is then used to make decisions about what products or services should be recommended to us in future Internet use. There is no theoretical framework for this; it is rather as though the intelligence is built-in to the data.

In **supervised learning** the system is fed knowledge with associated classification. For example, an AI program might be under development for marking exam paper questions. In the supervised learning, answers to examination questions could be provided together with a grade for each one or with categorised comments.

A special case of supervised learning is where an expert system is being developed. An expert system always has a focus on a narrowly defined domain of knowledge. In this case human experts are given samples of data requiring analysis. The experts provide the conclusions to be drawn from the data. The data and conclusions are input to the knowledge base. The effectiveness of the system can be tested by a human expert providing sample data and checking the accuracy of the conclusions provided by the expert system. If performance is poor, then further data and conclusions are input to the system. Although an expert system is an example of AI it is not an example of machine learning because there is no expectation that the system will improve its performance unaided.

Reinforcement learning has some features similar to unsupervised learning and other features similar to supervised learning. The method has its own specific vocabulary. The following statements use this vocabulary in describing aspects of how a reinforcement learning algorithm works.

- An agent is learning how best to perform in an environment.
- The environment has many defined states.
- At each step the agent takes an action.
- An agent has a policy that guides its actions.
- The policy is influenced by the recorded history and the knowledge of the current state of the environment.
- An action changes the environment to a new state.
- The agent receives a reward following an action which is a measure of how effective the action was in relation to the achievement of the overall goal.

- The policy will guide the agent in deciding whether the next action should be exploiting knowledge already known or exploring a new avenue.

In summary, the aim is to maximise the reward values by improving the quality of the policy. It is a trial-and-error search for optimum performance. It requires many repeated attempts at the same problem.

This description is an abstraction. The concept becomes clearer if some instances of the application of the approach are considered. One area of application is playing logic games such as backgammon. Another is robotics where a robot has to learn how to become effective at a task. Another option is where the machine has to learn how to navigate a maze. In this last case when the agent chooses a left turn which will eventually lead to the destination the reward is given a positive numeric value. If instead it chooses a right turn the reward is given a negative value.

Regression analysis methods

In some applications the aim of the AI is to predict and provide, as output numerical values for some defined quantity, on the basis of data values for different quantities that have been input to the AI algorithm. If **regression analysis** is to be used, the first step is for the system to be provided with some actual values for both the input data and for what will become output data when the AI system is operational. This data can be used to investigate if there is any correlation between these sets of values. If a correlation is established which can be represented by a mathematical formula, then this formula can be used to output predicted values when new data is input.

The simplest application of regression analysis is when values for only one quantity are to be input and when a linear relationship is expected between these values and the values to be predicted. An example could be an AI system being used to predict marks for candidates in an A Level Computer Science exam. There may be an expectation that there would be a correlation between a candidate’s marks in an A level Computer Science exam and their marks in an IGCSE Mathematics exam. Figure 22.04 shows what might be found when some historic data is input and analysed.

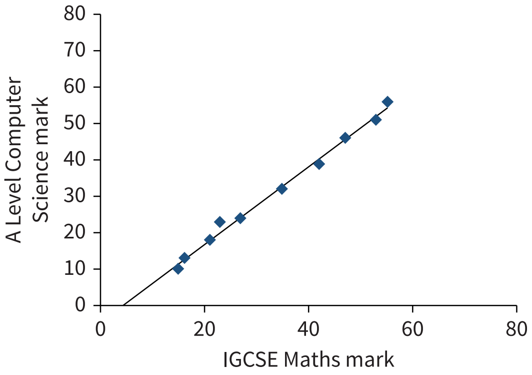


Figure 22.04 An example of a linear regression analysis

There is good correlation between the two sets of marks. The straight line in Figure 22.04 is the one that has been calculated as the best fit to the data. The formula for the line can sensibly be used to predict future marks for the A level Computer Science paper from the marks scored in an IGCSE Mathematics paper.

The regression analysis could be more complicated. For example, a similar fitting to a mathematical formula can be carried out if marks for three different exams are used as input. In other cases, a non-linear relationship might be appropriate as might happen when a prediction of future sales of a new product was needed where the growth in sales was expected to be exponential.

22.04 Artificial neural networks

The neural networks in our brains provide our intelligence. It therefore seems obvious that artificial neural networks should be considered as a foundation for artificial intelligence systems. Figure 22.05 shows a representation of two nerve cells.

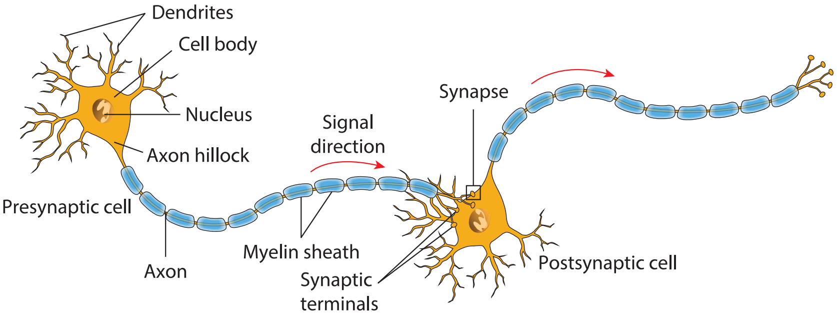


Figure 22.05 Two nerve cells showing how a signal is transmitted

At the one end of a nerve cell there are many dendrites which can receive signals. At the other end of the cell there are many axon terminal buttons that can transmit signals. The synapse is the region between an axon terminal button and a dendrite which contains neurotransmitters. When a nerve cell receives input signals the voltage in the axon increases. At some threshold value of this voltage neurotransmitters are activated and signals are sent to the dendrites of adjacent cells.

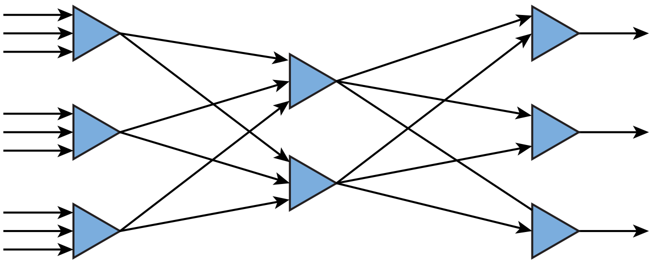


Figure 22.06 A schematic representation of a simple artificial neural network

An artificial neural network could be created in software or hardware. The components of the network can be represented by a diagram as illustrated in Figure 22.06. The triangles are the nodes in the network which represent artificial neurons. (Sometimes these are represented as circles). In general, a node can receive one or more inputs and can provide an output to one or more of the other nodes. The modelling of the action of the node involves applying a weighting factor to each input. The weighted input values are summed and then an activation function is used to compute a value for the output of the node. If the input is not a numerical value it must be converted to one so that the weighted values can be summed.

Figure 22.06 shows a very simple network structure consisting of three layers. The column of three nodes on the left receive input. The column on the right provides output. The two nodes in between form what is referred to as a hidden layer. Some artificial neural networks will contain several hidden layers.

An example of an AI system using an artificial neural network is one which estimates the cost-effectiveness of batteries based on the initial price paid and the lifetime of use. Each input node would represent one example of a battery. The inputs would be specific data that identified the battery and the price. One of the nodes in the hidden layer could be concerned with the type of device the battery was to be used in. The other node in the hidden layer could relate to the type of user of the device. Each node in the output layer would compute an estimated value for the cost per unit of time for a specific battery.

In this system there are adjustable factors for each node. These are the weighting factors for each input and the activation function. When the initial learning is taking place, these adjustable factors need to be tuned to achieve the optimum predictive capability of the system. The method that can be used for this is **back propagation of errors**. There is a need for some battery lifetime data from some real use of the batteries. The AI system is created with a set of trial values for all of the adjustable factors. The system is run with the input data that matches the real use. There will then be an error identified for each output node. The error is the difference between the output value and the real-use value for the battery lifetime. The system is then re-run with different values for the weighting factors and activation functions applying to the output nodes. This will determine the dependency of the accuracy of the output value on the adjustable factors associated with the performance of the nodes in the output layer.

The next step is to carry out a similar investigation for the nodes in the hidden layer. Finally, the adjustable factors for the nodes in the input layer are tackled. This process of learning should now be sufficient for the system to be applied to some new input data to predict the cost-effectiveness of some different batteries. When more real-use data becomes available then the back propagation of errors learning can be applied again to achieve improved performance of the system.

Extension Question 22.01

An early successful application of an artificial neural network was NETtalk. This was able to take a text as input and output a synthesised sound reading of the text. The network had 7 times 29 input nodes and 26 output nodes. Can you investigate why that number of input nodes was chosen and what was the function of some of the 80 nodes in the hidden layer?

Deep Learning

It is understood that in the brain there is a layer structure of neurons where lower layers have readily understandable functions but where higher layers are concerned with more abstract processing of information. With the increasing computing power now available, artificial neural networks are being introduced with large numbers of hidden layers which are attempting to achieve something similar. These are known as **Deep Learning** systems.

Reflection Point:

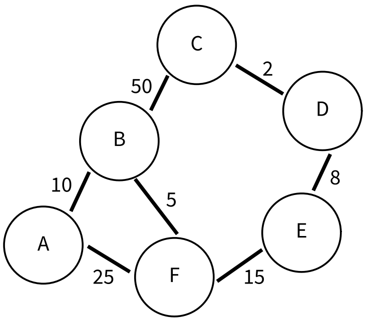
Could you create a hierarchical chart to show how the various approaches discussed in this chapter are related to each other?

Summary

- A graph can be constructed from nodes and edges where the edges carry numerical value labels.
- Algorithms are available to find the shortest path between two nodes in a graph.
- Machine learning can be supervised or unsupervised.
- Regression analysis involves finding a mathematical equation which is a best fit to sample data.
- Artificial neural networks are modelled using nodes which receive input and provide output.
- Back propagation of errors can be used for machine learning using artificial neural networks.

Exam-style Questions

1 The diagram below shows a graph representing the cost of journeys between railway stations identified by A, B, C, D, E and F.



Dijkstra's algorithm is to be used to find the total cost for journeys from station A to each of the other stations. A record structure is to be used to store for each station the cost for the travel so far and the list of stations so far visited in the order visited. Complete the table below to record the progress of the algorithm by identifying which nodes are in the ShortestPath set and what would be stored in the record at each step of the algorithm.

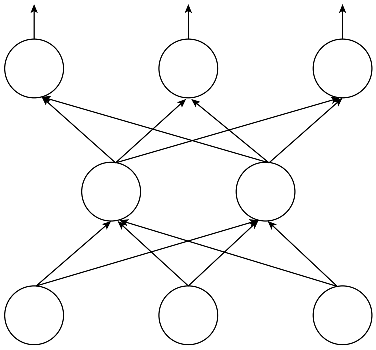
The first two rows of the table have been completed for you.

Content of the ShortestPath set	**Content of the record**					
{}	A	B	C	D	E	F
	0	∞	∞	∞	∞	∞
{A}	A	B	C	D	E	F
	0	10	∞	∞	∞	25
	A	A-B				A-F
	A	B	C	D	E	F
	A	B	C	D	E	F
	A	B	C	D	E	F
	A	B	C	D	E	F
	A	B	C	D	E	F

[7]

2 The diagram below represents an artificial neural network.

a Give as full a description as you can of what the parts of the diagram represent. If you wish you can label the diagram then use the labels in your answer. [5]



b Identify the steps involved when a backward propagation of errors algorithm is used. [4]

3 **a** Give a brief explanation of each of the following terms:

Machine learning

Artificial neural network

Deep learning [9]

b Explain which approach uses back propagation of errors. [2]

Part 4
Further problem-solving and programming skills

Chapter 23: Algorithms

Learning objectives

By the end of this chapter you should be able to:

- show understanding of linear and binary searching methods
- write a linear search algorithm
- write a binary search algorithm
- show understanding of the conditions necessary for the use of a binary search
- show understanding of how the performance of a binary search varies according to the number of data items
- show understanding of insertion sort and bubble sort methods
- show understanding that performance of a sorting routine may depend on the initial order of the data and the number of data items
- show understanding of and use Abstract Data Types (ADT)
- show how it is possible for ADTs to be implemented from another ADT
- describe the following ADTs and demonstrate how they can be implemented from appropriate built-in types or other ADTs: stack, queue, linked list, dictionary, binary tree
- write algorithms to:
  - implement an insertion sort
  - implement a bubble sort
  - find an item in each of the following: linked list, binary tree
  - insert an item into each of the following: stack, queue, linked list, binary tree
  - delete an item from each of the following: stack, queue, linked list
- show understanding that a graph is an example of an ADT
- describe the key features of a graph and justify its use for a given situation
- show understanding that different algorithms which perform the same task can be compared by using criteria such as time taken to complete the task and memory used
- show understanding of Big O notation for time and space complexity.

23.01 Linear search

In Chapter 13, we developed the algorithm for a linear search (Worked Example 13.02).

Discussion Point:

What were the essential features of a linear search?

TASK 23.01

Write program code for the linear search algorithm. Assume that the items to be searched are stored in a 1D array with n elements.

23.02 Bubble sort

In Chapter 13, we developed the algorithm for a bubble sort (Worked Example 13.03).

Discussion Point:

What were the essential features of a bubble sort?

TASK 23.02

Write program code for the most efficient bubble sort algorithm. Assume that the items to be sorted are stored in a 1D array with n elements.

23.03 Insertion sort

Imagine you have a number of cards with a different value printed on each card. How would you sort these cards into order of increasing value?

You can consider the pile of cards as consisting of a sorted part and an unsorted part. Place the unsorted cards in a pile on the table. Hold the sorted cards as a pack in your hand. To start with only the first (top) card is sorted. The card on the top of the pile on the table is the next card to be inserted. The last (bottom) card in your hand is your current card.

Figure 23.01 shows the sorted cards in your hand as blue and the pile of unsorted cards as white. The next card to be inserted is shown in red. Each column shows the state of the pile as the cards are sorted.

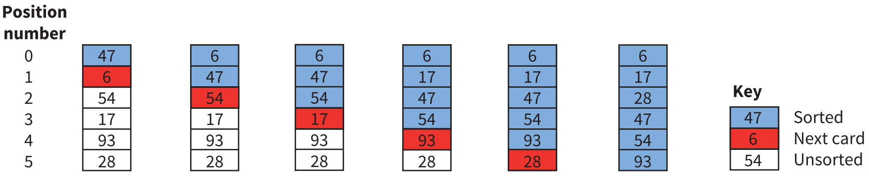


Figure 23.01 Sorting cards

Repeat the following steps until all cards in the unsorted pile have been inserted into the correct position.

1 Repeat until the card to be inserted has been placed in its correct position.

1.1 Compare the current card with the card to be inserted.

1.2 If the card to be inserted is greater than the current card, insert it below the current card.

1.3 Otherwise, if there is a card above the current card in your hand, make this your new current card.

1.4 If there is no new current card, place the card to be inserted at the top of the sorted pile.

What happens when you work through the sorted cards to find the correct position for the card to be inserted? In effect, as you consider the cards in your hand, you move the current card down a position. If the value of the card to be inserted is smaller than the last card you considered, then the card is inserted at the top of the pile (position 1).

This method is known as an insertion sort. It is a standard sort method.

We can write this algorithm using pseudocode. Assume the values to be sorted are stored in a 1D array, `List`:

```
FOR Pointer ← 1 TO NumberOfitems – 1
    ItemToBeInserted ← List[Pointer]
    CurrentItem ← Pointer – 1 // pointer to last item in sorted part of list
    WHILE (List[CurrentItem] > ItemToBeInserted) AND (CurrentItem > –1) DO
        List[CurrentItem + 1] ← List[CurrentItem] // move current item down
        CurrentItem ← CurrentItem – 1 // look at the item above
    ENDWHILE
    List[CurrentItem + 1] ← ItemToBeInserted // insert item
NEXT Pointer
```

TASK 23.03

1 Dry-run the insertion sort algorithm using a trace table. Assume the list consists of the following six items in the order given: 53, 21, 60, 18, 42, 19.

2 Write program code for the insertion sort algorithm. Assume that the items to be sorted are stored in a 1D array with n elements.

Extension Question 23.01

Investigate the performances of the insertion sort and the bubble sort by:

- varying the initial order of the items
- increasing the number of items to be sorted.

23.04 Binary search

In Section 23.01 we considered the algorithm for a linear search. This is the only way we can systematically search an unordered list. However, if the list is ordered, then we can use a different technique.

Consider the following real-world example.

If you want to look up a word in a dictionary, you are unlikely to start searching for the word from the beginning of the dictionary. Suppose you are looking for the word ‘quicksort’. You look at the middle entry of the dictionary (approximately) and find the word ‘magnetic’. ‘quicksort’ comes after ‘magnetic’, so you look in the second half of the dictionary. Again, you look at the entry in the middle of this second half of the dictionary (approximately) and find the word ‘report’. ‘quicksort’ comes before ‘report’, so you look in the third quarter. You can keep looking at the middle entry of the part which must contain your word, until you find the word. If the word does not exist in the dictionary, you will have no entries in the dictionary left to find the middle of.

This method is known as a **binary search**. It is a standard method.

We can write this algorithm using pseudocode. Assume the values are sorted in ascending order and stored in a 1D array, `List` of size `MaxItems`.

```
Found ← FALSE
SearchFailed ← FALSE
First ← 0
Last ← MaxItems – 1 // set boundaries of search area
WHILE NOT Found AND NOT SearchFailed DO
    Middle ← (First + Last) DIV 2 // find middle of current search area
    IF List[Middle] = SearchItem
      THEN
        Found ← TRUE
      ELSE
        IF First >= Last // no search area left
          THEN
            SearchFailed ← TRUE
          ELSE
            IF List[Middle] > SearchItem
              THEN  // must be in first half
                Last ← Middle - 1  // move upper boundary
              ELSE  // must be in second half
                First ← Middle + 1  // move lower boundary
            ENDIF
        ENDIF
    ENDIF
ENDWHILE
IF Found = TRUE
  THEN
    OUTPUT Middle // output position where item was found
  ELSE
    OUTPUT "Item not present in array"
ENDIF
```

TASK 23.04

1 Dry-run the binary search algorithm using a trace table. Assume the list consists of the following 20 items in the order given: 7, 12, 19, 23, 27, 33, 37, 41, 45, 56, 59, 60, 62, 71, 75, 80, 84, 88, 92, 99.

2. Search for the value 60. How many times did you have to execute the `While` loop?
3. Dry-run the algorithm again, this time searching for the value 34. How many times did you have to execute the While loop?

Discussion Point:

Compare the binary-search algorithm with the linear-search algorithm. If the array contains n items, how many times on average do you need to test a value when using a binary search and how many times on average do you need to test a value when using a linear search? Can you describe how the search time varies with increasing n?

23.05 Abstract Data Types (ADTs)

In Chapter 13 (Section 13.07 to 13.10), we introduced ADTs using conceptual diagrams and how ADTs can be implemented using arrays.

23.06 Linked lists

Look back at Chapter 13 Section 13.10. Figure 13.25 shows an empty linked list. The `StartPointer` variable contains the Null pointer. The free list links all empty nodes.

We now code the basic operations discussed using the conceptual diagrams in Figures 13.11 to 13.16.

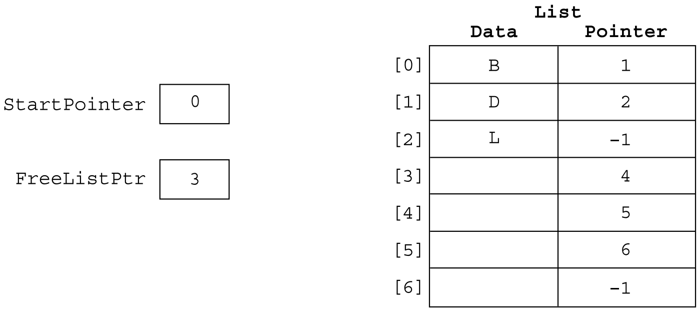
Create a new linked list

```
// NullPointer should be set to -1 if using array element with index 0
CONSTANT NullPointer = –1
// Declare record type to store data and pointer
TYPE ListNode
    DECLARE Data    : STRING
    DECLARE Pointer : INTEGER
ENDTYPE
DECLARE StartPointer : INTEGER
DECLARE FreeListPtr  : INTEGER
DECLARE List : ARRAY[0 : 6] OF ListNode

PROCEDURE InitialiseList
    StartPointer ← NullPointer       // set start pointer
    FreeListPtr ← 0                 // set starting position of free list
    FOR Index ← 0 TO 5              // link all nodes to make free list
        List[Index].Pointer ← Index + 1
    NEXT Index
    List [6].Pointer ← NullPointer   // last node of free list
ENDPROCEDURE
```

Insert a new node into an ordered linked list

```
PROCEDURE InsertNode(NewItem)
    IF FreeListPtr <> NullPointer
      THEN  // there is space in the array
        // take node from free list and store data item
        NewNodePtr ← FreeListPtr
        List[NewNodePtr].Data ← NewItem
        FreeListPtr ← List[FreeListPtr].Pointer
        // find insertion point
        ThisNodePtr ← StartPointer  // start at beginning of list
        PreviousNodePtr ← NullPointer
        WHILE ThisNodePtr <> NullPointer  // while not end of list
          AND List[ThisNodePtr].Data < NewItem DO
            PreviousNodePtr ← ThisNodePtr  // remember this node
            // follow the pointer to the next node
            ThisNodePtr ← List[ThisNodePtr].Pointer
        ENDWHILE
        IF PreviousNodePtr = StartPointer
          THEN  // insert new node at start of list
            List[NewNodePtr].Pointer ← StartPointer
            StartPointer ← NewNodePtr
          ELSE  // insert new node between previous node and this node
            List[NewNodePtr].Pointer ← List[PreviousNodePtr].Pointer
            List[PreviousNodePtr].Pointer ← NewNodePtr
        ENDIF
    ENDIF
ENDPROCEDURE
```

After three data items have been added to the linked list, the array contents are as shown in Figure 23.02.



StartPointer 0

FreeListPtr 3

List

	Data	Pointer
[0]	B	1
[1]	D	2
[2]	L	-1
[3]		4
[4]		5
[5]		6
[6]		-1

Figure 23.02 Linked list of three nodes and free list of four nodes

Find an element in an ordered linked list

```
FUNCTION FindNode(DataItem) RETURNS INTEGER  // returns pointer to node
    CurrentNodePtr ← StartPointer  // start at beginning of list
    WHILE CurrentNodePtr <> NullPointer  // not end of list
      AND List[CurrentNodePtr].Data <> DataItem DO // item not found
        // follow the pointer to the next node
        CurrentNodePtr ← List[CurrentNodePtr].Pointer
    ENDWHILE
    RETURN CurrentNodePtr  // returns NullPointer if item not found
ENDFUNCTION
```

Delete a node from an ordered linked list

```
PROCEDURE DeleteNode(DataItem)
    ThisNodePtr ← StartPointer             // start at beginning of list
    WHILE ThisNodePtr <> NullPointer       // while not end of list
      AND List[ThisNodePtr].Data <> DataItem DO // and item not found
        PreviousNodePtr ← ThisNodePtr  // remember this node
        // follow the pointer to the next node
        ThisNodePtr ← List[ThisNodePtr].Pointer
    ENDWHILE
    IF ThisNodePtr <> NullPointer  // node exists in list
      THEN
        IF ThisNodePtr = StartPointer // first node to be deleted
          THEN
            // move start pointer to the next node in list
            StartPointer ← List[StartPointer].Pointer
          ELSE
            // it is not the start node;
            // so make the previous node’s pointer point to
            // the current node’s 'next' pointer; thereby removing all
            // references to the current pointer from the list
            List[PreviousNodePtr].Pointer ← List[ThisNodePtr].Pointer
        ENDIF
        List[ThisNodePtr].Pointer ← FreeListPtr
        FreeListPtr ← ThisNodePtr
    ENDIF
```

```
ENDPROCEDURE
```

Access all nodes stored in the linked list

```
PROCEDURE OutputAllNodes
    CurrentNodePtr ← StartPointer // start at beginning of list
    WHILE CurrentNodePtr <> NullPointer DO // while not end of list
        OUTPUT List[CurrentNodePtr].Data
        // follow the pointer to the next node
        CurrentNodePtr ← List[CurrentNodePtr].Pointer
    ENDWHILE
ENDPROCEDURE
```

TASK 23.05

Convert the pseudocode for the linked-list handling subroutines to program code. Incorporate the subroutines into a program and test them.

Note that a stack ADT and a queue ADT can be treated as special cases of linked lists. The linked list stack only needs to add and remove nodes from the front of the linked list (see Section 23.08). The linked list queue only needs to add nodes to the end of the linked list and remove nodes from the front of the linked list (see Section 23.09).

23.07 Binary trees

In the real world, we draw tree structures to represent hierarchies. For example, we can draw a family tree showing ancestors and their children. A binary tree is different to a family tree because each node can have at most two ‘children’.

In computer science binary trees are used for different purposes. In Chapter 20 (Section 20.06), you saw the use of a binary tree as a syntax tree. In this chapter, you will use an ordered binary tree ADT (such as the one shown in Figure 23.03) as a binary search tree.

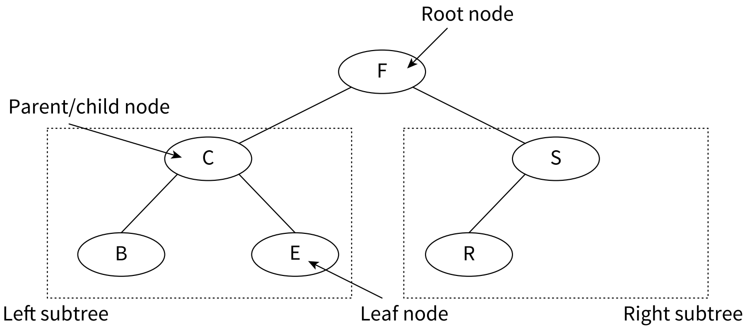


Figure 23.03 Conceptual diagram of an ordered binary tree

Nodes are added to an ordered binary tree in a specific way:

```
Start at the root node as the current node.
Repeat
  If the data value is greater than the current node’s data value, follow the right branch.
  If the data value is smaller than the current node’s data value, follow the left branch.
Until the current node has no branch to follow.
Add the new node in this position.
```

For example, if we want to add a new node with data value D to the binary tree in Figure 23.03, we execute the following steps.

1 Start at the root node.

2 D is smaller than F, so turn left.

3 D is greater than C, so turn right.

4 D is smaller than E, so turn left.

5 There is no branch going left from E, so we add D as a left child from E (see Figure 23.04).

This type of tree has a special use as a search tree. Just like the binary search applied to an ordered linear list, the binary search tree gives the benefit of a faster search than a linear search or searching a linked list. The ordered binary tree also has a benefit when adding a new node: other nodes do not need to be moved, only a left or right pointer needs to be added to link the new node into the existing tree.

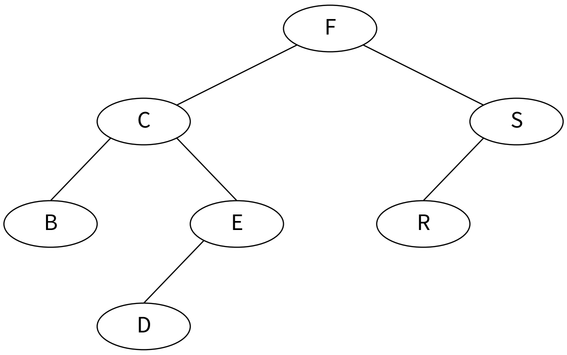


Figure 23.04 Conceptual diagram of adding a node to an ordered binary tree

We can store the binary tree in an array of records (see Figure 23.05). One record represents a node and consists of the data and a left pointer and a right pointer. Unused nodes are linked together to form a free list.

RootPointer -1

FreePtr 0

	Tree		
	LeftPointer	Data	RightPointer
[0]	1		
[1]	2		
[2]	3		
[3]	4		
[4]	5		
[5]	6		
[6]	-1		

Figure 23.05 Binary tree before any nodes are inserted

Create a new binary tree

```
// NullPointer should be set to -1 if using array element with index 0
CONSTANT NullPointer = –1
// Declare record type to store data and pointers
TYPE TreeNode
    DECLARE Data : STRING
    DECLARE LeftPointer : INTEGER
    DECLARE RightPointer : INTEGER
ENDTYPE
DECLARE RootPointer : INTEGER
DECLARE FreePtr : INTEGER
DECLARE Tree : ARRAY[0 : 6] OF TreeNode
PROCEDURE InitialiseTree
    RootPointer ← NullPointer  // set start pointer
    FreePtr ← 0                // set starting position of free list
    FOR Index ← 0 TO 5         // link all nodes to make free list
        Tree[Index].LeftPointer ← Index + 1
    NEXT Index
    Tree [6].LeftPointer ← NullPointer // last node of free list
ENDPROCEDURE
```

Insert a new node into a binary tree

```
PROCEDURE InsertNode(NewItem)
    IF FreePtr <> NullPointer
      THEN  // there is space in the array
        // take node from free list, store data item, set null pointers
        NewNodePtr ← FreePtr
        FreePtr ← Tree[FreePtr].LeftPointer
        Tree[NewNodePtr].Data ← NewItem
        Tree[NewNodePtr].LeftPointer ← NullPointer
        Tree[NewNodePtr].RightPointer ← NullPointer
        // check if empty tree
        IF RootPointer = NullPointer
          THEN  // insert new node at root
            RootPointer ← NewNodePtr
          ELSE  // find insertion point
```

```
            ThisNodePtr ← RootPointer   // start at the root of the tree
            WHILE ThisNodePtr <> NullPointer DO  // while not a leaf node
                PreviousNodePtr ← ThisNodePtr  // remember this node
                IF Tree[ThisNodePtr].Data > NewItem
                  THEN  // follow left pointer
                    TurnedLeft ← TRUE
                    ThisNodePtr ← Tree[ThisNodePtr].LeftPointer
                  ELSE // follow right pointer
                    TurnedLeft ← FALSE
                    ThisNodePtr ← Tree[ThisNodePtr].RightPointer
                ENDIF
            ENDWHILE
            IF TurnedLeft = TRUE
              THEN
                Tree[PreviousNodePtr].LeftPointer ← NewNodePtr
              ELSE
                Tree[PreviousNodePtr].RightPointer ← NewNodePtr
            ENDIF
        ENDIF
    ENDIF
ENDPROCEDURE
```

Find a node in a binary tree

```
FUNCTION FindNode(SearchItem) RETURNS INTEGER  // returns pointer to node
    ThisNodePtr ← RootPointer     // start at the root of the tree
    WHILE ThisNodePtr <> NullPointer   // while a pointer to follow
      AND Tree[ThisNodePtr].Data <> SearchItem DO // and search item not found
        IF Tree[ThisNodePtr].Data > SearchItem
          THEN  // follow left pointer
            ThisNodePtr ← Tree[ThisNodePtr].LeftPointer
          ELSE // follow right pointer
            ThisNodePtr ← Tree[ThisNodePtr].RightPointer
        ENDIF
    ENDWHILE
    RETURN ThisNodePtr // will return null pointer if search item not found
ENDFUNCTION
```

TASK 23.06

Write program code to implement a binary search tree.

23.08 Stacks

In Chapter 13 (Section 13.08) we looked at the conceptual data structure of a stack. A stack can be implemented using a 1D array. Figure 23.06 shows a stack containing four data items. For conceptual reasons the array elements are numbered from the bottom up.

Note that `BaseOfStackPointer` will always point to element 0 of the array. `TopOfStackPointer` will vary. It will increase when an item is pushed onto the stack and it will decrease when an item is popped off the stack. When the stack is empty, `TopOfStackPointer` will have the value–1.

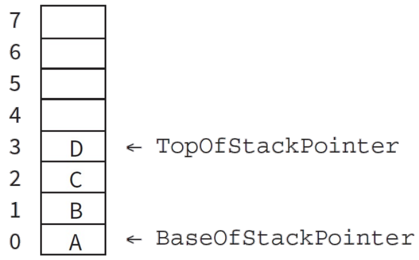


Figure 23.06 A stack

Create a new stack

```
// NullPointer should be set to -1 if using array element with index 0
CONSTANT EMPTYSTRING = ""
CONSTANT NullPointer = –1
CONSTANT MaxStackSize = 8
DECLARE BaseOfStackPointer : INTEGER
DECLARE TopOfStackPointer : INTEGER
DECLARE Stack : ARRAY[1 : MaxStackSize – 1] OF STRING

PROCEDURE InitialiseStack
    BaseOfStackPointer ← 0             // set base of stack pointer
    TopOfStackPointer ← NullPointer    // set top of stack pointer
ENDPROCEDURE
```

Push an item onto the stack

```
PROCEDURE Push(NewItem)
    IF TopOfStackPointer < MaxStackSize – 1
      THEN     // there is space on the stack
               // increment top of stack pointer
        TopOfStackPointer ← TopOfStackPointer + 1
               // add item to top of stack
        Stack[TopOfStackPointer] ← NewItem
    ENDIF
ENDPROCEDURE
```

Pop an item off the stack

```
FUNCTION Pop()
    DECLARE Item : STRING
    Item ← EMPTYSTRING
    IF TopOfStackPointer > NullPointer
      THEN     // there is at least one item on the stack
               // pop item off the top of the stack
        Item ← Stack[TopOfStackPointer]
               // decrement top of stack pointer
        TopOfStackPointer ← TopOfStackPointer – 1
    ENDIF
    RETURN Item
ENDFUNCTION
```

TASK 23.07

1. Write a program to implement the above pseudocode subroutines. Add a menu to test the subroutines.
2. Write a program to implement a stack as a linked list. Note that the adding and removing of nodes is much simpler than for an ordered linked list.

23.09 Queues

In Chapter 13 (Section 13.09) we looked at the conceptual data structure of a queue. A queue can be implemented using a 1D array. Figure 23.07 shows a queue containing five data items.

`FrontOfQueuePointer` always points to the first element in the queue, that is the next element to be taken from the queue. `EndOfQueuePointer` always points to the last element in the queue. Before another element joins the queue, the `EndOfQueuePointer` is incremented. Note that when adjusting either pointer the possibility of wrap-round has to be tested.

To make it easier to test whether the queue is empty or full, a counter variable can be used.

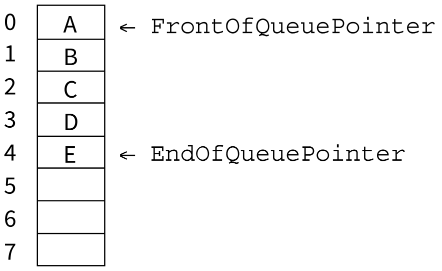


Figure 23.07 A queue before wrap-round

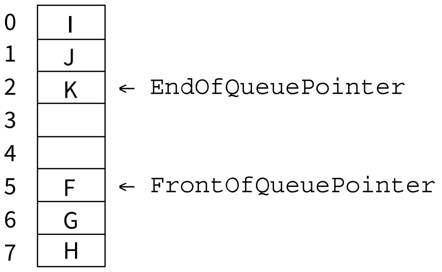


Figure 23.08 A queue with wrap-round

Create a new queue

```
// NullPointer should be set to -1 if using array element with index 0
CONSTANT EMPTYSTRING = ""
CONSTANT NullPointer = -1
CONSTANT MaxQueueSize = 8
DECLARE FrontOfQueuePointer : INTEGER
DECLARE EndOfQueuePointer : INTEGER
DECLARE NumberInQueue : INTEGER
DECLARE Queue : ARRAY[0 : MaxQueueSize – 1] OF STRING
PROCEDURE InitialiseQueue
    FrontOfQueuePointer ← NullPointer   // set front of queue pointer
    EndOfQueuePointer ← NullPointer     // set end of queue pointer
    NumberInQueue ← 0                   // no elements in queue
ENDPROCEDURE
```

Add an item to the queue

```
PROCEDURE AddToQueue(NewItem)
    IF NumberInQueue < MaxQueueSize
      THEN     // there is space in the queue
               // increment end of queue pointer
        EndOfQueuePointer ← EndOfQueuePointer + 1
               // check for wrap-round
        IF EndOfQueuePointer > MaxQueueSize – 1
```

```
          THEN     // wrap to beginning of array
            EndOfQueuePointer ← 0
                    // add item to end of queue
        ENDIF
        Queue[EndOfQueuePointer] ← NewItem
         // increment counter
         NumberInQueue ← NumberInQueue + 1
    ENDIF
ENDPROCEDURE
```

Remove an item from the queue

```
FUNCTION RemoveFromQueue()
    DECLARE Item : STRING
    Item ← EMPTYSTRING
    IF NumberInQueue > 0
      THEN // there is at least one item in the queue
           // remove item from the front of the queue
        Item ← Queue[FrontOfQueuePointer]
        NumberInQueue ← NumberInQueue – 1
        IF NumberInQueue = 0
          THEN // if queue empty, reset pointers
            CALL InitialiseQueue
          ELSE
            // increment front of queue pointer
            FrontOfQueuePointer ← FrontOfQueuePointer + 1
            // check for wrap-round
            IF FrontOfQueuePointer > MaxQueueSize – 1
              THEN // wrap to beginning of array
              FrontOfQueuePointer ← 0
            ENDIF
        ENDIF
    ENDIF
    RETURN Item
ENDFUNCTION
```

TASK 23.08

1 Write a program to implement the above pseudocode subroutines. Add a menu to test the subroutines.

2 Write program code to implement a queue as a linked list. You may find it helpful to introduce another pointer that always points to the end of the queue. You will need to update it when you add a new node to the queue.

23.10 Graphs

In Computer Science a graph is an ADT consisting of vertices (nodes) and edges. Graphs are used to record relationships between things. For uses of graphs in AI see Chapter 22 Section 22.02. A simple graph is shown in Figure 23.09. It represents a small part of the London Underground map shown in Figure 12.02 in Chapter 12.

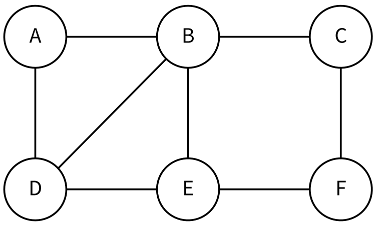


Figure 23.09 Graph showing part of the London Underground map

The vertices labelled A to F are the underground stations and the edges represent train lines connecting the stations. For example, you can take a train directly from B to D. To get from B to F, you have to travel via C or E. Two vertices connected by an edge are known as neighbours.

A labelled (weighted) graph has edges with values representing something. In our example, we can add weights to Figure 23.10 to show the time it takes to travel between stations:

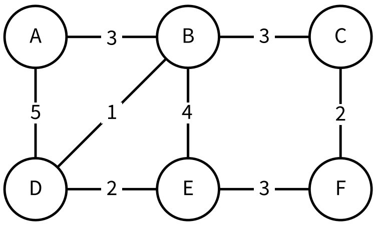


Figure 23.10 Weighted graph showing travelling times between stations

Graphs can be directed or undirected (as in Figure 23.10). Travel times may vary depending on the direction of travel. This can be shown in a directed graph (see Figure 23.11).

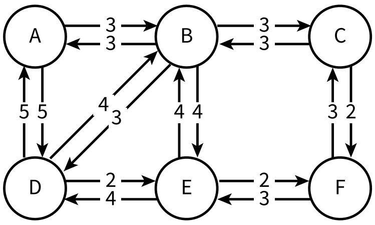


Figure 23.11 Directed graph

Sometimes one direction may not be available. For example, if the line from B to D is blocked, this could be represented as shown in Figure 23.12.

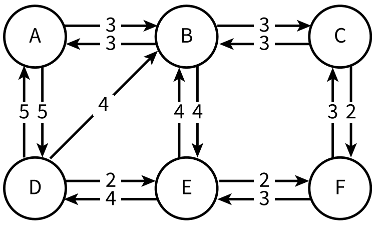


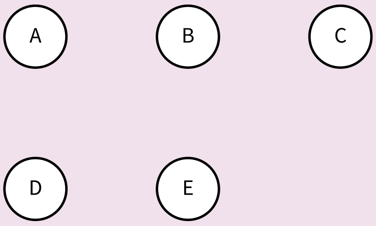
Figure 23.12 Directed graph

We can use a graph to plan a journey. Using Figure 23.12, we want to travel from C to D, so we can either use the route C B A D (11 minutes) or the route C F E D (9 minutes). For the return journey we can use D B C (7 minutes).

TASK 23.09

Draw the labelled edges for a directed graph of a road system where:

- A to B is 2 km, and it is a one-way system
- D to A is 1 km, and is one-way
- B to D is 3 km and is one-way
- B to C is 4 km
- C to E is 7 km
- D to E is 5 km.



To implement a graph, we can use an adjacency matrix or an adjacency list.

An adjacency matrix stores the relationship between every vertex to all other vertices. For an unweighted graph, a 1 represents an edge, a 0 no edge. When weights are to be recorded, the weight replaces the 1. Instead of a 0, we use the infinity symbol ∞.

	A	B	C	D	E	F
A	0	1	0	1	0	0
B	1	0	1	1	1	0
C	0	1	0	0	0	1
D	1	1	0	0	1	0
E	0	1	0	1	0	1
F	0	0	1	0	1	0

Table 23.01

Adjacency matrix for Figure 23.09

	A	B	C	D	E	F
A	∞	3	∞	5	∞	∞
B	3	∞	3	∞	4	∞
C	∞	3	∞	∞	∞	2
D	5	4	∞	∞	2	∞
E	∞	4	∞	4	∞	2
F	∞	∞	3	∞	3	∞

Table 23.02

Adjacency matrix for Figure 23.12

An adjacency list stores the relationship between every vertex to all relevant vertices. An entry is made only when there is an edge between two vertices. For a weighted graph, the connection as well as the weight is stored in the list.

	connected to
A	B, D
B	A, C, D, E
C	B, F
D	A, B, E
E	B, D, F
F	C, E

Table 23.03

Adjacency list for Figure 23.09

	connected to
A	B, 3; D, 5
B	A, 3; C, 3; E, 4
C	B, 3; F, 2
D	A, 5; B, 4; E, 2
E	B, 4; D, 4; F, 2
F	C, 3; E, 3

Table 23.04

Adjacency list for Figure 23.12

TASK 23.10

Construct an adjacency matrix and an adjacency list to represent your graph from Task 23.09.

23.11 Hash tables

If we want to store records in an array and have direct access to records, we can use the concept of a hash table.

The idea behind a hash table is that we calculate an address (the array index) from the key value of the record and store the record at this address. When we search for a record, we calculate the address from the key and go to the calculated address to find the record. Calculating an address from a key is called ‘hashing’.

Finding a hashing function that will give a unique address from a unique key value is very difficult. If two different key values hash to the same address this is called a ‘collision’. There are different ways to handle collisions:

- chaining: create a linked list for collisions with start pointer at the hashed address
- using overflow areas: all collisions are stored in a separate overflow area, known as ‘closed hashing’
- using neighbouring slots: perform a linear search from the hashed address to find an empty slot, known as ‘open hashing’.

WORKED EXAMPLE 23.01

Calculating addresses in a hash table

Assume we want to store customer records in a 1D array `HashTable[0 : n]`. Each customer has a unique customer ID, an integer in the range 10001 to 99999.

We need to design a suitable hashing function. The result of the hashing function should be such that every index of the array can be addressed directly. The simplest hashing function gives us addresses between 0 and n:

```
FUNCTION Hash(Key) RETURNS INTEGER
    Address ← Key MOD(n + 1)
    RETURN Address
ENDFUNCTION
```

For illustrative purposes, we choose n to be 9. Our hashing function is:

```
Index ← CustomerID MOD 10
```

We want to store records with customer IDs: 45876, 32390, 95312, 64636, 23467. We can store the first three records in their correct slots, as shown in Figure 23.13.

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
32390		95312				45876			

Figure 23.13 A hash table without collisions

The fourth record key (64636) also hashes to index 6. This slot is already taken; we have a collision. If we store our record here, we lose the previous record. To resolve the collision, we can choose to store our record in the next available space, as shown in Figure 23.14.

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
32390		95312				45876	64636		

Figure 23.14 A hash table with a collision resolved by open hashing

The fifth record key (23467) hashes to index 7. This slot has been taken up by the previous record, so again we need to use the next available space (Figure 23.15).

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
32390		95312				45876	64636	23467	

Figure 23.15 A hash table with two collisions resolved by open hashing

When searching for a record, we need to allow for these out-of-place records. We know if the record we are searching for does not exist in the hash table when we come across an unoccupied slot.

We will now develop algorithms to insert a record into a hash table and to search for a record in the hash table using its record key.

The hash table is a 1D array `HashTable[0 : Max] OF Record`.

The records stored in the hash table have a unique key stored in field `Key`.

Insert a record into a hash table

```
PROCEDURE Insert(NewRecord)
    Index ← Hash(NewRecord.Key)
    WHILE HashTable[Index] NOT empty DO
        Index ← Index + 1 // go to next slot to check if empty
        IF Index > Max // beyond table boundary?
          THEN // wrap around to beginning of table
            Index ← 0
        ENDIF
    ENDWHILE
    HashTable[Index] ← NewRecord
ENDPROCEDURE
```

Find a record in a hash table

```
FUNCTION FindRecord(SearchKey) RETURNS Record
    Index ← Hash(SearchKey)
    WHILE (HashTable[Index].Key <> SearchKey)
      AND (HashTable[Index] NOT empty) DO
        Index ← Index + 1 // go to next slot
        IF Index > Max // beyond table boundary?
          THEN // wrap around to beginning of table
            Index ← 0
        ENDIF
    ENDWHILE
    IF HashTable[Index] NOT empty // if record found
      THEN
        RETURN HashTable[Index] // return the record
    ENDIF
ENDFUNCTION
```

23.12 Dictionaries

A real-world dictionary is a collection of key–value pairs. The key is the term you use to look up the required value. For example, if you use an English–French dictionary to look up the English word ‘book’, you will find the French equivalent word ‘livre’. A real-world dictionary is organised in alphabetical order of keys.

An ADT dictionary in computer science is implemented using a hash table (see Section 23.11), so that a value can be looked up using a direct-access method.

Python, VB.NET and Java have a built-in ADT dictionary class.

Here are some examples of Python dictionaries:

```
EnglishFrench = {} # empty dictionary
EnglishFrench["book"] = "livre" # add a key-value pair to the dictionary
EnglishFrench["pen"] = "stylo"
print(EnglishFrench["book"]) # access a value in the dictionary
# alternative method of setting up a dictionary
ComputingTerms = {"Boolean" : "can be TRUE or FALSE", "Bit" : "0 or 1"}
print(ComputingTerms["Bit"])
```

Here are some examples of VB dictionaries:

```
Dim EnglishFrench As New Dictionary(Of String, String)
EnglishFrench.Add("book", "livre")
EnglishFrench.Add("pen", "stylo")
Console.WriteLine(EnglishFrench.Item("book"))
Dim ComputingTerms As New Dictionary(Of String, String)
ComputingTerms.Add("Boolean", "can be TRUE or FALSE")
ComputingTerms.Add("Bit", "0 or 1")
Console.WriteLine(ComputingTerms.Item("Bit"))
Console.ReadLine()
```

Here are some examples of Java dictionaries (the Dictionary class is obsolete, use HashMap instead):

```
import java.util.Map;
import java.util.HashMap;
    Map<String, String> englishFrench = new HashMap<String, String>();
    englishFrench.put("book", "livre");
    englishFrench.put("pen", "stylo");
    System.out.println(englishFrench.get("book"));
    Map<String, String>  computingTerms = new HashMap<String, String>();
    computingTerms.put("Boolean", "can be TRUE or FALSE");
    computingTerms.put("Bit", "0 or 1");
    System.out.println(computingTerms.get("Bit"));
```

There are many built-in functions for Python, VB and Java dictionaries. These are beyond the scope of this book. However, we need to understand how dictionaries are implemented. The following pseudocode shows how to create a new dictionary.

```
TYPE DictionaryEntry
    DECLARE Key   : STRING
    DECLARE Value : STRING
ENDTYPE
DECLARE EnglishFrench : ARRAY[0 : 999] OF DictionaryEntry  // empty dictionary
```

TASK 23.11

Write pseudocode to:

- insert a key–value pair into a dictionary
- look up a value in a dictionary.

Use the hashing function from Worked Example 23.01.

23.13 Big O notation

A problem can be solved in different ways, with different algorithms. Clearly, we want to use time and memory efficiently. A way of comparing the efficiency of algorithms has been devised using order of growth as a function of the size of the input. Big O notation is used to classify algorithms according to how their running time (or space requirements) grows as the input size grows. The letter O is used because the growth rate of a function is also referred to as 'order of the function'. The worst-case scenario is used when calculating the order of growth for very large data sets.

Consider the linear search algorithm in Chapter 13, Worked Example 13.02. The worst case scenario is that the item searched for is the last item in the list. The longer the list, the more comparisons have to be made. If the list is twice as long, twice as many comparisons have to be made. Generally, we can say the order of growth is linear. We write this as O(n), where n is the size of the data set.

Consider the bubble sort algorithm for the worst case scenario.

```
Unsorted ← n – 1
FOR i ← 0 TO n – 2
    FOR j ← 0 TO Unsorted - 1
        IF MyList[j] > MyList[j + 1]
          THEN
            Temp ← MyList[j]
            MyList[j] ← MyList[j + 1]
            MyList[j + 1] ← Temp
        ENDIF
    NEXT j
    Unsorted ← Unsorted - 1
NEXT i
```

The basic operation for this algorithm is the comparison `IF MyList[j] > MyList[j + 1]`

n	Number of comparisons	
1	0	
2	1	
3	3	= 1 + 2
4	6	= 1 + 2 + 3
5	10	= 1 + 2 + 3 + 4
6	15	= 1 + 2 + 3 + 4 + 5

Table 23.05 Number of comparisons in bubble sort

We can see that the total number of comparisons is the sum of the first (n – 1) whole numbers. This leads us to the formula:

½ * n * (n – 1) = ½ * (n^2 – n)

For very large n we can disregard all factors except the largest and we get n^2. So the order of growth is n^2.

Consider the binary search algorithm in Section 23.04. With each iteration this algorithm halves the number of values in the data set. This iterative halving of data sets produces a growth curve that peaks at the beginning and slowly flattens out as the size of the data sets increase. This type of algorithm is described as O($\log_2$n)

Table 23.06 shows a summary of standard algorithms and their order of growth (time complexity).

Order of growth	Example	Explanation

$O(1)$	`FUNCTION GetFirstItem(List : ARRAY)`
	`RETURN List[1]`
		The complexity of the algorithm does not change regardless of data set size

$O(n)$	Linear search Bubble sort performed on an already sorted list	Linear growth
$O(\log_2 n)$	Binary search	The total time taken increases as the data set size increases, but each comparison halves the data set. So the time taken increases by smaller amounts and approaches constant time.
$O(n^2)$	Bubble sort Insertion sort	Polynomial growth Common with algorithms that involve nested iterations over the data set
$O(n^3)$		Polynomial growth Deeper nested iterations will result in $O(n^3)$, $O(n^4)$, …
$O(2^n)$	Recursive calculation of Fibonacci numbers	Exponential growth

Table 23.06 Order of growth (time complexity) for data input set of size n

Space complexity refers to the amount of memory required for a growing number of values n in the dataset.

The ADT operations discussed in this chapter have space complexity O(n). This means that they only take the space required for the data set. Bubble sort and insertion sort have space complexity of O(1). This means they do not require extra memory for sorting larger lists.

Reflection Point:

List the standard algorithms you have met in this chapter. Can you give the essential features of each of these?

Summary

- Standard algorithms include bubble sort, insertion sort, linear search and binary search.
- Abstract data types (ADTs) include stacks, queues, linked lists, binary trees, hash tables, dictionaries and graphs.
- Basic operations required for an ADT include creating an ADT and inserting, finding or deleting an element of an ADT.
- Time taken and space used can be measured using Big O notation.

Exam-style Questions

1 **a** Complete the algorithm for a binary search function `FindName`.

The data being searched is stored in the array `Names[0 : 50]`.

The name to be searched for is passed as a parameter.

```
FUNCTION FindName(s : STRING) RETURNS INTEGER
    Index ← -1
    First ← 0
    Last ← 50
    WHILE (Last >= First) AND ........................................ DO
        Middle ← (First + Last) DIV 2
        IF Names[Middle] = s
          THEN
            Index ← Middle
          ELSE
            IF .....................................................
             THEN
               Last ← Middle + 1
             ELSE
                ..................................................
            ENDIF
        ENDIF
    ENDWHILE
ENDFUNCTION
```

[3]

b The binary search does not work if the data in the array being searched is .... [1]

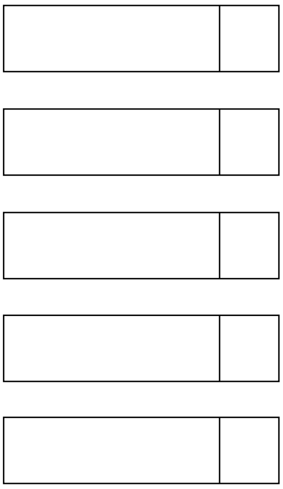
c What does the function `FindName` return when:

i the name searched for exists in the array?

ii the name searched for does not exist in the array? [2]

2 A queue Abstract Data Type (ADT) is to be implemented as a linked list of nodes. Each node is a record, consisting of a data field and a pointer field. The queue ADT also has a `FrontOfQueue` pointer and an `EndOfQueue` pointer associated with it. The possible queue operations are: `JoinQueue` and `LeaveQueue`.

a **i** Add labels to the diagram to show the state of the queue after three data items have been added to the queue in the given order: Apple, Pear, Banana.



[5]

ii Add labels to the diagram to show how the unused nodes are linked to form a list of free nodes. This list has a StartOfFreeList pointer associated with it. [2]

b **i** Write program code to declare the record type Node. [3]

ii Write program code to create an array Queue with 50 records of type Node. Your solution should link all nodes and initialise the pointers FrontOfQueue, EndOfQueue and StartOfFreeList. [7]

c The pseudocode algorithm for the queue operation JoinQueue is written as a procedure with the header:

```
PROCEDURE JoinQueue(NewItem)
```

where NewItem is the new value to be added to the queue. The procedure uses the variables shown in the following identifier table:

Identifier	**Data type**	**Description**
NullPointer	INTEGER	Constant set to –1
		Array to store queue data
	STRING	Value to be added
		Pointer to next free node in array
		Pointer to first node in queue
		Pointer to last node in queue
		Pointer to node to be added

i Complete the identifier table. [7]

ii Complete the pseudocode using the identifiers from the table in **part (c) (i)**. [6]

```
PROCEDURE JoinQueue(NewItem : STRING)
    // Report error if no free nodes remaining
    IF StartOfFreeList = ..............................
      THEN
        Report Error
      ELSE
        // new data item placed in node at start of free list
        NewNodePointer ← StartOfFreeList
        Queue[NewNodePointer].Data ← NewItem
        // adjust free list pointer
        StartOfFreeList ← Queue[NewNodePointer].Pointer
        Queue[NewNodePointer].Pointer ← NullPointer
        // if first item in queue then adjust front of queue pointer
        IF FrontOfQueue = NullPointer
          THEN
            ..............................
            ..............................
        ENDIF
        // new node is new end of queue
        Queue[.....................].Pointer ← .....................
        EndOfQueue ← ...............................................
    ENDIF
ENDPROCEDURE
```

3 A program is required that sorts a list of words into alphabetical order. The list of words is supplied as a text file.

a Write a program to declare a string array, WordList, that can hold 500 elements. Initialise the array so all elements contain the empty string. [3]

b Write a procedure, `OutputList`, to output all elements in index order. [3]

c Write a procedure, `LoadWords`, that asks the user for a filename and reads the contents of the text file, storing each line of text (word) in a separate array element. The procedure should output a relevant error message if:

- the file doesn't exist
- the array is full. [7]

d Write a procedure, `SortWords`, to perform a bubble sort on all non-empty array elements, so that the words are in alphabetical order. [7]

e Write program code to call `LoadWords`, then `OutputList`, followed by `SortWords` and then `OutputList` again. [3]

f Test your program by running it first with a non-existing file, and then with a text file containing 20 words in random order.

Make screenshots of your test runs that show your code works correctly. [2]

Chapter 24:
Recursion

Learning objectives

By the end of this chapter you should be able to:

- show understanding of the essential features of recursion
- show understanding of how recursion is expressed in a programming language
- trace recursive algorithms
- write recursive algorithms
- show understanding of when the use of recursion is beneficial
- show awareness of what a compiler has to do to translate recursive programming code, including the use of stacks and unwinding.

24.01 Concept of recursion

In mathematical logic and computer science, a function or procedure is said to be a **recursive routine** if it is defined in terms of itself.

The classic mathematical example is the factorial function, n!, which is defined in Figure 24.01. This definition holds for all positive whole numbers.

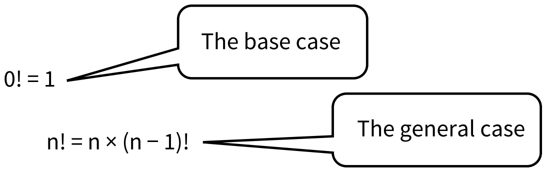


Figure 24.01 Mathematical definition of the factorial function

Figure 24.02 shows expressions of the factorial function for the first four numbers.

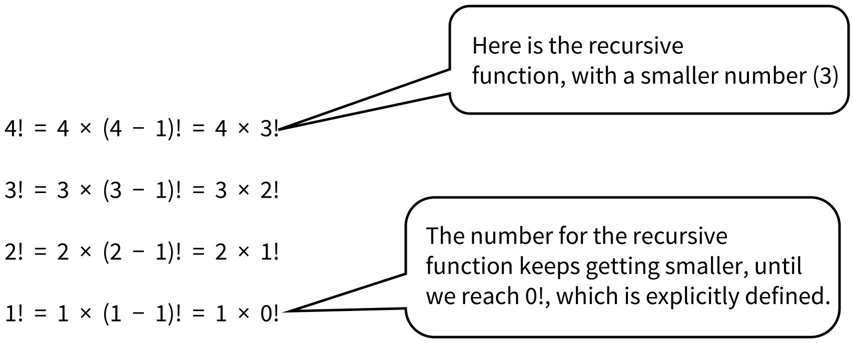

Here is the recursive function, with a smaller number (3)
4! = 4 × (4 − 1)! = 4 × 3!
3! = 3 × (3 − 1)! = 3 × 2!
2! = 2 × (2 − 1)! = 2 × 1!
1! = 1 × (1 − 1)! = 1 × 0!
The number for the recursive function keeps getting smaller, until we reach 0!, which is explicitly defined.

Figure 24.02 Expressions of the factorial function

Because 0! = 1:

$$4! = 4 \times 3 \times 2 \times 1 \times 1 = 24$$

Recursive solutions have a **base case** and a **general case**. The base case gives a result without involving the general case. The general case is defined in terms of the definition itself. It is very important that the general case must come closer to the base case with each recursion, for any starting point.

24.02 Programming a recursive subroutine

WORKED EXAMPLE 24.01

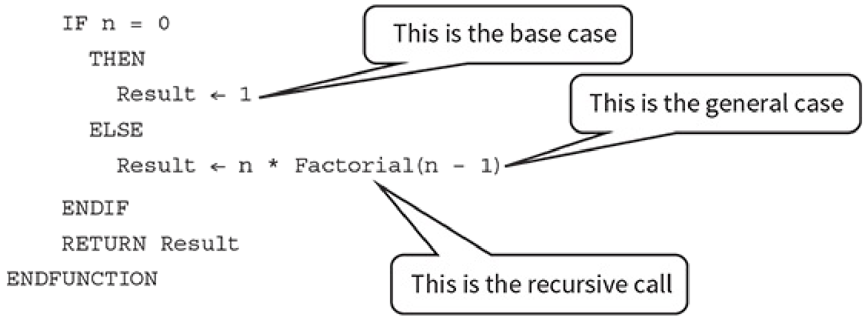
Coding the factorial function

We could program the function `Factorial` iteratively, using a loop:

```
FUNCTION Factorial(n : INTEGER) RETURNS INTEGER
    Result ← 1
    FOR i ← 1 TO n
        Result ← Result * i
    NEXT i
    RETURN Result
ENDFUNCTION
```

Alternatively, we can define the function `Factorial` recursively (Figure 24.03).

```
FUNCTION Factorial(n : INTEGER) RETURNS INTEGER
```



```
    IF n = 0
      THEN
        Result ← 1
      ELSE
        Result ← n * Factorial(n – 1)
    ENDIF
    RETURN Result
ENDFUNCTION
```

Figure 24.03 The factorial function defined recursively

The recursive pseudocode resembles the original mathematical definition of the factorial function. The dry run in Figure 24.05 (Section 24.03) shows how this works.

Discussion Point:

Carefully examine the two solutions to the factorial function. What happens if the iterative function is called with parameter 0? What happens if the recursive function is called with parameter 0? What changes would need to be made so the mathematical definition holds for all values of n?

When writing a recursive subroutine, there are three rules you must observe. A recursive subroutine must:

- have a base case
- have a general case
- reach the base case after a finite (limited) number of calls to itself.

TASK 24.01

Write program code to implement the recursive algorithm for the factorial function.

Question 24.01

What happens when the function is called with `Factorial(-2)`? Which rule is not satisfied?

WORKED EXAMPLE 24.02

Coding a recursive procedure

Consider a procedure to count down from a given integer. We can write the solution as an iterative algorithm:

```
PROCEDURE CountDownFrom(n : INTEGER)
   FOR i ← n DOWNTO 0
       OUTPUT i
   NEXT i
ENDPROCEDURE
```

We can also write the solution as a recursive algorithm. Consider what happens after the first value has been output. The remaining numbers follow the same pattern of counting down from the next smaller value. The base case is when n reaches 0. 0 will be output but no further numbers. The general case is outputting n and then counting down from (n – 1). This can be written using pseudocode:

```
PROCEDURE CountDownFrom(n : INTEGER)
    OUTPUT n
    IF n > 0
      THEN
        CALL CountDownFrom(n – 1)
    ENDIF
ENDPROCEDURE
```

24.03 Tracing a recursive subroutine

Tracing a recursive procedure

Dry-running the recursive procedure from Worked Example 24.02, we can complete a trace table as shown in Table 24.01.

Call number	Procedure call	OUTPUT	n > 0
1	`CountDownFrom(3)`	3	TRUE
2	`CountDownFrom(2)`	2	TRUE
3	`CountDownFrom(1)`	1	TRUE
4	`CountDownFrom(0)`	0	FALSE

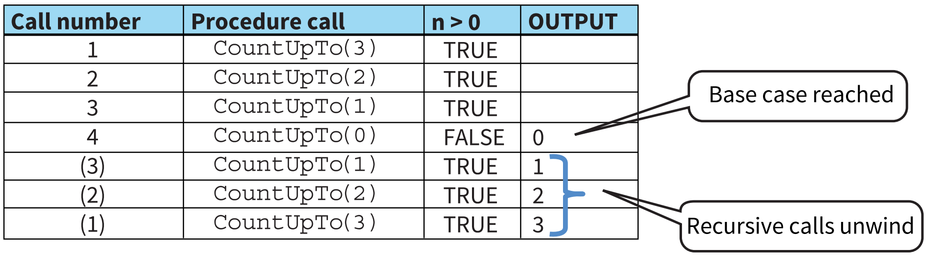
Table 24.01 Trace table for `CALL CountDownFrom(3)`

It is more complex to trace a subroutine that contains statements to execute after the recursive call. Look at the slightly modified algorithm:

```
PROCEDURE CountUpTo(n : INTEGER)
    IF n > 0
      THEN
        CALL CountUpTo(n – 1)
    ENDIF
    OUTPUT n
ENDPROCEDURE
```

Note that the statements after `CALL CountUpTo(n – 1)` are not executed until control returns to this statement as the recursive calls unwind.

What is the effect of moving the `OUTPUT` statement to the end of the procedure? Figure 24.04 traces the execution of `CALL CountUpTo(3)`



Call number	Procedure call	n > 0	OUTPUT
1	CountUpTo(3)	TRUE	
2	CountUpTo(2)	TRUE	
3	CountUpTo(1)	TRUE	
4	CountUpTo(0)	FALSE	0
(3)	CountUpTo(1)	TRUE	1
(2)	CountUpTo(2)	TRUE	2
(1)	CountUpTo(3)	TRUE	3

Base case reached
Recursive calls unwind

Figure 24.04 Trace table for `CALL CountUpTo(3)`

When the base case is reached, the fourth call of the procedure is complete and the procedure is exited. Control then passes back to the third call and so on. Note how we show the trace as the recursive calls unwind. Don’t go back up the table and fill in the OUTPUT column as this will not make it clear enough when the output occurred.

Tracing a recursive function

A recursive function has a statement after the recursive call to itself: the `RETURN` statement. Again we show what happens when the recursive calls unwind by filling in more rows in the trace table. Let’s consider the factorial function again.

```
FUNCTION Factorial(n : INTEGER) RETURNS INTEGER
   IF n = 0
     THEN
       Result ← 1
     ELSE
```

```
        Result ← n * Factorial(n – 1)
    ENDIF
    RETURN Result
ENDFUNCTION
```

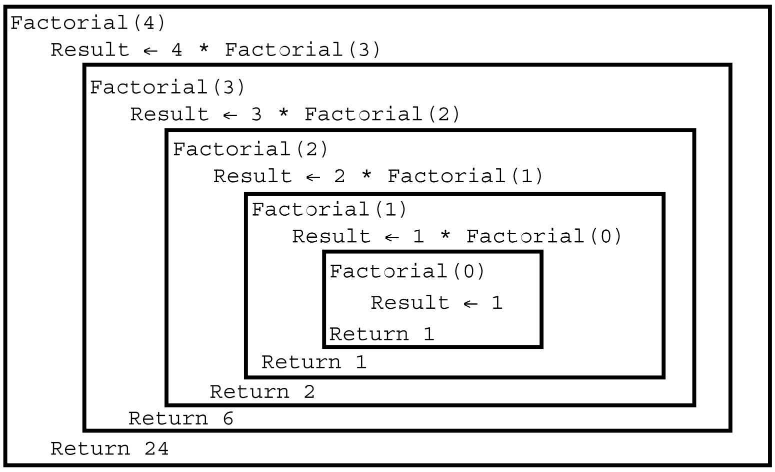
Call number	Procedure call	n = 0	Result	Return value
1	Factorial(4)	FALSE	4 * Factorial(3)	
2	Factorial(3)	FALSE	3 * Factorial(2)	
3	Factorial(2)	FALSE	2 * Factorial(1)	
4	Factorial(1)	FALSE	1 * Factorial(0)	
5	Factorial(0)	TRUE	1	1
(4)	Factorial(1)	FALSE	1 * 1	1
(3)	Factorial(2)	FALSE	2 * 1	2
(2)	Factorial(3)	FALSE	3 * 2	6
(1)	Factorial(4)	FALSE	4 * 6	24

Base case reached

Recursive calls unwind

Figure 24.05 Trace table for Trace table for `CALL Factorial(4)`

Another way to illustrate how the function calls unwind is by framing each call with a box (see Figure 24.06). When the innermost box is completed the result is fed to the next one out. And so on until the outermost box has been completed.



```
Factorial(4)
    Result ← 4 * Factorial(3)
        Factorial(3)
            Result ← 3 * Factorial(2)
                Factorial(2)
                    Result ← 2 * Factorial(1)
                        Factorial(1)
                            Result ← 1 * Factorial(0)
                                Factorial(0)
                                    Result ← 1
                                Return 1
                            Return 1
                    Return 2
            Return 6
    Return 24
```

Figure 24.06 Diagrammatic view of recursive calls of `Factorial`

TASK 24.02

Consider the following recursive algorithm:

```
PROCEDURE X(n : INTEGER)
    IF (n = 0) OR (n = 1)
      THEN
        OUTPUT n
      ELSE
        CALL X(n DIV 2)
        OUTPUT(n MOD 2)
    ENDIF
ENDPROCEDURE
```

Dry-run the procedure call `X(19)` by completing a trace table. What is the purpose of this algorithm?

24.04 Running a recursive subroutine

Recursive subroutines can only be executed if the compiler produces object code that uses a stack to push return addresses and local variables when calling a subroutine repeatedly.

WORKED EXAMPLE 24.03

Running the factorial function

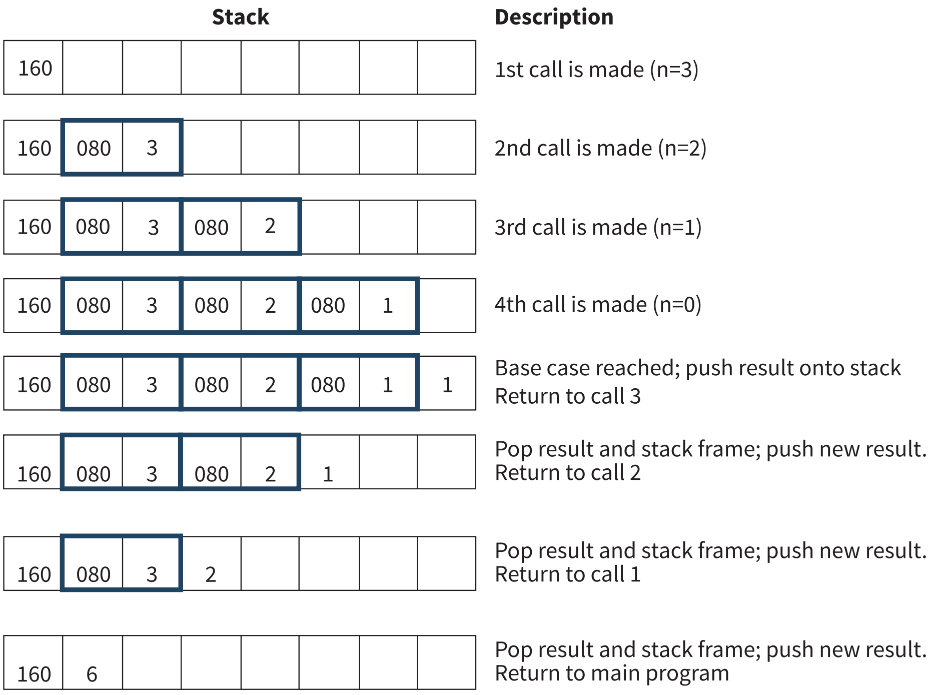
Consider the following program, written in pseudocode:

```
010  PROGRAM
020
030  FUNCTION Factorial(n : INTEGER) RETURNS INTEGER
040      IF n = 0
050        THEN
060          Result ← 1
070        ELSE
080          Result ← n * Factorial(n - 1)
090      ENDIF
100      RETURN Result
110  ENDFUNCTION
120
130  // main program
140
150  DECLARE Answer : INTEGER
160  Answer ← Factorial(3)
170  OUTPUT Answer
180
190  ENDPROGRAM
```

The first program statement to be executed is line 160. The actual parameter `n` has the value 3. The function call causes the return address to be put on the stack, as shown in Figure 24.07. Program execution jumps to line 30.

When line 80 is reached, the function call causes the return address to be stored on the stack, together with the current contents of the local variables. The locations used to store these values are referred to as a stack frame (represented by the blue borders in Figure 24.07). Each recursive call will add another stack frame to the stack until the base case is reached.

When the base case is reached, the result of the function call `Factorial(0)` is returned by pushing it onto the stack. The result is popped off the stack by the previous call of the function. With each return from a function call, the corresponding stack frame is taken off and the values of the local variables are restored. Eventually, control is returned to line 160 with the result of the function call on the top of the stack. The value of `Answer` is output in line 170.



Stack								Description
160								1st call is made (n=3)
160	080	3						2nd call is made (n=2)
160	080	3	080	2				3rd call is made (n=1)
160	080	3	080	2	080	1		4th call is made (n=0)
160	080	3	080	2	080	1	1	Base case reached; push result onto stack Return to call 3
160	080	3	080	2	1			Pop result and stack frame; push new result. Return to call 2
160	080	3	2					Pop result and stack frame; push new result. Return to call 1
160	6							Pop result and stack frame; push new result. Return to main program

Figure 24.07 Stack contents during recursive calls of `Factorial`

TASK 24.03

Use your program code from Task 24.01 and add the main program as shown in Worked Example 24.03.

Amend your code in the following ways (line numbers are relative to the pseudocode in Worked Example 24.03):

- Add a global integer variable `CallNumber`
- Initialise `CallNumber` to zero (line `155`).
- Increment `CallNumber` (line `35`).
- Add a statement to output the values of `CallNumber` and `n` (line `36`).
- Add a statement to output the value of `Result` (line `95`).

Run the program and study the output.

24.05 Benefits and drawbacks of recursion

Recursion is an important technique in different programming paradigms (See Chapter 29, Section 29.08). When designing a solution to a mathematical problem that is recursive by nature, the easiest way to write a solution is to implement the recursive definition. Some optimising compilers will change a recursive subroutine to an iterative one when producing object code.

Recursive solutions are often more elegant and use less program code than iterative solutions. However, repeated recursive calls can carry large amounts of memory usage and processor time (see Section 24.04). For example, the procedure call `CountDownFrom(100)` will require 100 stack frames before it completes.

Reflection Point:

How can you tell from a function or procedure definition whether or not it is recursive?

Summary

- A recursive subroutine is defined in terms of itself.
- A recursive subroutine must have a base case and a general case.
- A recursive subroutine must reach the base case after a finite number of calls to itself.
- Each time a subroutine is called, a stack frame is pushed onto the stack.
- A stack frame consists of the return address and the values of the local variables.
- When a subroutine completes, the corresponding stack frame is popped off the stack.

Exam-style Questions

1 **a** Outline the differences between iteration and recursion. [2]

b Give **one** advantage and **one** disadvantage of using recursive subroutines. [2]

2 The following is a recursively defined function which calculates the result of Base$^{\text{Exponent}}$. For example, 2^3 is 8.

```
FUNCTION Power(Base: INTEGER, Exponent : INTEGER) RETURNS INTEGER
    IF Exponent = 0
      THEN
        Result ← 1
      ELSE
        Result ← Base * Power(Base, Exponent – 1)
    ENDIF
    RETURN Result
ENDFUNCTION
```

a What is meant by 'recursively defined'? [1]

b Trace the execution of the function call `Power(2,4)` showing for each re-entry into the `Power` function, the values passed to the function and the results returned. [6]

c Explain the role of the stack in the execution of the Power function. [3]

d Write a pseudocode non-recursive (iterative) version of the Power function. [6]

e **i** Give **one** reason why a non-recursive Power function may be preferred to a recursive one. [1]

ii Give **one** reason why a recursive Power function may be preferred to a non-recursive one. [1]

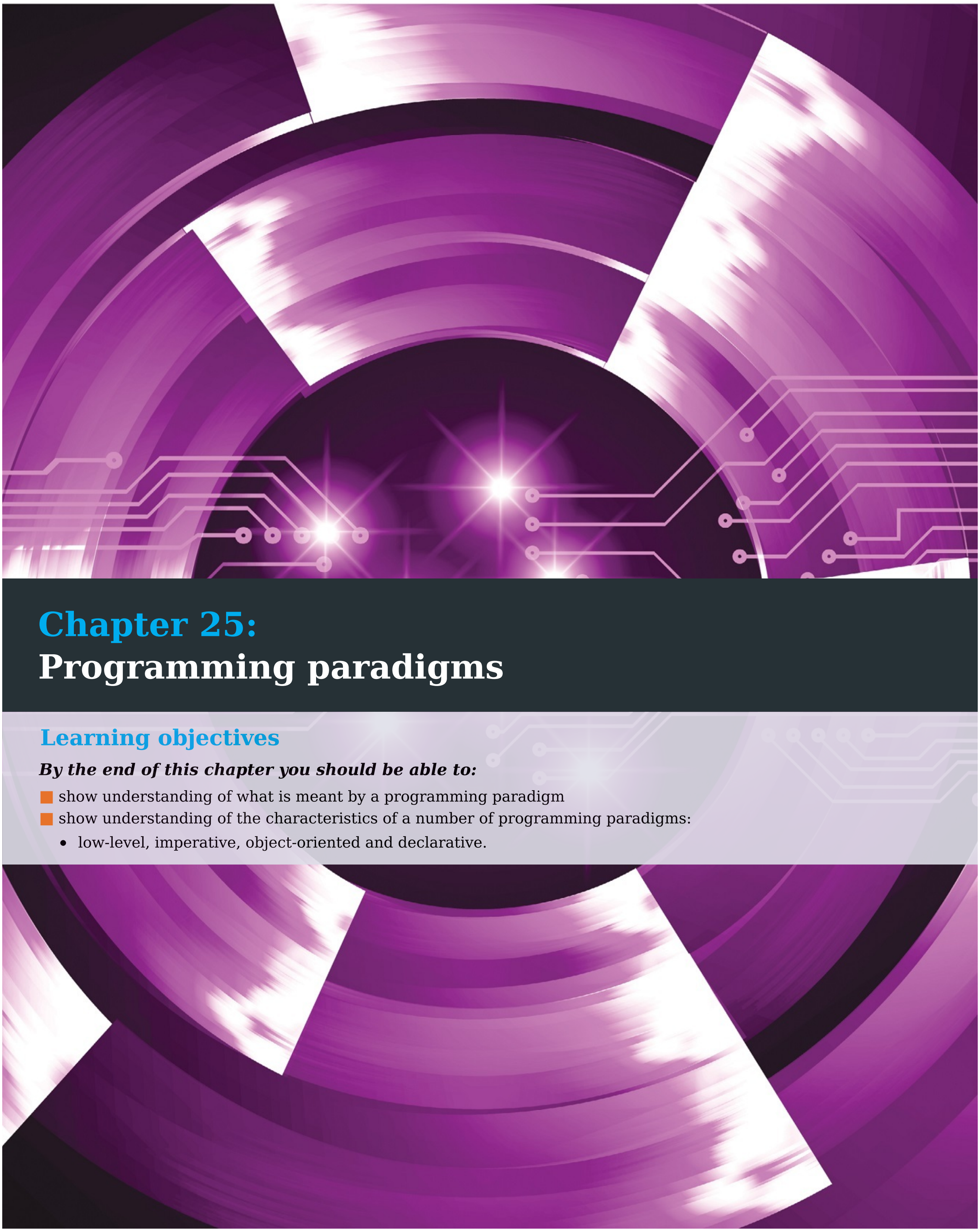
3 The following is a recursively defined function which calculates the *n*th integer in the sequence of Fibonacci numbers.

```
01  FUNCTION Fibonacci(n : INTEGER) RETURNS INTEGER
02      IF (n = 0) OR (n = 1)
03        THEN
04          Result ← 1
05        ELSE
06          Result ← Fibonacci(n – 1) + Fibonacci(n – 2)
07      ENDIF
08      RETURN Result
09  ENDFUNCTION
```

a **i** Which line is the base case? [1]

ii Which line is the general case? [1]

b Dry-run the function call `Fibonacci(4)`. [7]



Chapter 25:
Programming paradigms

Learning objectives

By the end of this chapter you should be able to:

- show understanding of what is meant by a programming paradigm
- show understanding of the characteristics of a number of programming paradigms:
  - low-level, imperative, object-oriented and declarative.

25.01 Programming paradigms

A **programming paradigm** is a fundamental style of programming. Each paradigm will support a different way of thinking and problem solving. Paradigms are supported by programming language features. Some programming languages support more than one paradigm. There are many different paradigms, and some overlap. Here are just a few different paradigms.

Low-level programming paradigm

The features of low-level programming languages give us the ability to manipulate the contents of memory addresses and registers directly and exploit the architecture of a given processor. We solve problems in a very different way when we use the low-level programming paradigm than if we use a high-level paradigm. See Chapter 6 and Chapter 28 for low-level programming examples. Note that each different type of processor has its own programming language. There are 'families' of processors that are designed with similar architectures and therefore use similar programming languages. For example, the Intel processor family (present in many PC-type computers) uses the x86 instruction set.

Imperative programming paradigm

Imperative programming involves writing a program as a sequence of explicit steps that are executed by the processor. Most of the programs in this book use imperative programming (Chapters 12 to 15 and Chapters 23, 24 and 26). An imperative program tells the computer *how* to get a desired result, in contrast to declarative programming where a program describes *what* the desired result should be. Note that the procedural programming paradigm belongs to the imperative programming paradigm. There are many imperative programming languages, Pascal, C and Basic to name just a few.

Object-oriented programming paradigm

The object-oriented paradigm is based on objects interacting with one another. These objects are data structures with associated methods (see Chapter 27). Many programming languages that were originally imperative have been developed further to support the object-oriented paradigm. Examples include Pascal (under the name Delphi or Object Pascal) and Visual Basic (the .NET version being the first fully object-oriented version). Newer languages, such as Python and Java, were designed to be object-oriented from the beginning.

Declarative programming paradigm

Programming languages such as Pascal, VB and Python are referred to as 'imperative programming languages' because the programmer writes sequences of statements that reflect how to solve the problem. When a programmer uses a declarative programming language, the programmer writes down (in the language of logic) a declarative specification that describes the situation of interest: *what* the problem is. The programmer doesn't tell the computer what to do. To get information, the programmer poses a query (sets a goal). It's up to the logic programming system to work out how to get the answer.

Declarative programs are expressed as formal logic and computations are deductions from the formal logic statements (see Chapter 29). Declarative programming languages include SQL (see Chapter 11) and Prolog (Chapter 29).

Reflection Point:

Why are there different paradigms?

After you have studied Chapters 26 to 29 do Tasks 25.01 and 25.02

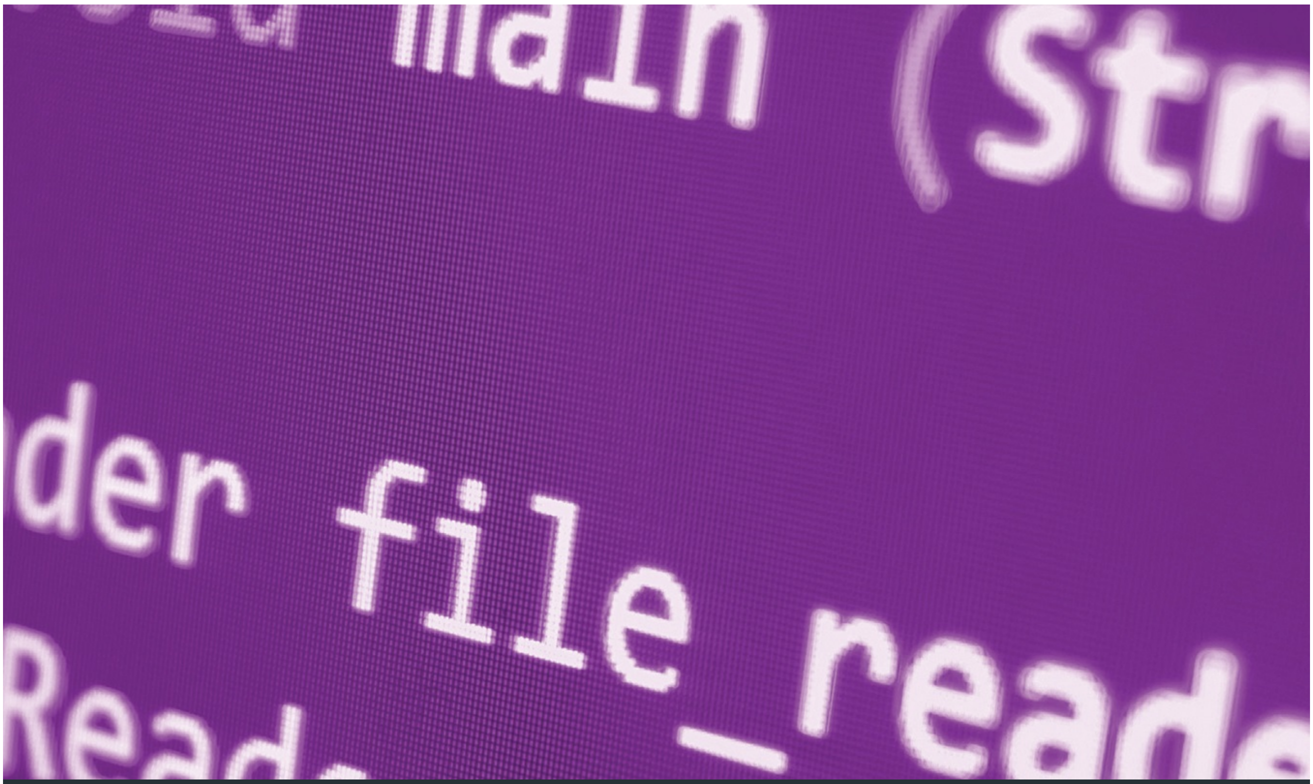
TASK 25.01

Draw a line between a statement on the left and its matching programming paradigm on the right.

Commands available depend on the type of processor	Declarative
Information hiding is used to protect internal properties of an object	
Statements operate on data	Imperative
Answer a question via a search for a solution	
Data is combined with procedures	Low-level
First do this and then do that	
The basic concept is a relation	Object-oriented

TASK 25.02

For each of the four programming paradigms in this chapter, give one programming statement example that is characteristic for this paradigm. State the programming language you used for each example.



Chapter 26:
File processing and exception handling

Learning objectives

By the end of this chapter you should be able to:

- write code to perform file-processing operations:
  - open a file (in read, write or append mode)
  - read a record from a file
  - write a record to a file
  - on serial, sequential and random files
- show understanding of an exception and the importance of exception handling
- show understanding of when it is appropriate to use exception handling
- write code to use exception handling.

26.01 Records

Records are user-defined types (discussed in Chapter 16, Section 16.01). See also Section 13.02.

WORKED EXAMPLE 26.01

Using records

A car manufacturer and seller wants to store details about cars. These details can be stored in a record structure:

```
TYPE CarRecord
    DECLARE VehicleID           : STRING  // unique identifier and record key
    DECLARE Registration        : STRING
    DECLARE DateOfRegistration  : DATE
    DECLARE EngineSize          : INTEGER
    DECLARE PurchasePrice       : CURRENCY
ENDTYPE
```

To declare a variable of that type we write:

```
DECLARE ThisCar : CarRecord
```

Note that we can declare arrays of records. If we want to store the details of 100 cars, we declare an array of type `CarRecord`

```
DECLARE Car : ARRAY[1:100] OF CarRecord
```

Python

Python does not have a record type. However, we can use a class definition with only a constructor to assign initial values. (See Chapter 27 for more about classes).
The pseudocode example of a car record described in Worked Example 26.01 can be programmed as follows:

```
class CarRecord:        # declaring a class without other methods
  def __init__(self):  # constructor
     self.VehicleID = ""
     self.Registration = ""
     self.DateOfRegistration = None
     self.EngineSize = 0
     self.PurchasePrice = 0.00
ThisCar = CarRecord() # instantiates a car record
ThisCar.EngineSize = 2500 # assign a value to a field
Car = [CarRecord() for i in range(100)] # make a list of 100 car records
Car[1].EngineSize = 2500 # assign value to a field of the 2nd car in list
```

VB.NET

```
Structure CarRecord
   Dim VehicleID As String
   Dim Registration As String
   Dim DateOfRegistration As Date
   Dim EngineSize As Integer
   Dim PurchasePrice As Decimal
End Structure
Dim ThisCar As CarRecord   ' declare a variable of CarRecord type
Dim Car(100) As CarRecord  ' declare an array of CarRecord type
ThisCar.EngineSize = 2500 ' assign value to a field
Car(2).EngineSize = 2500  ' assign value to a field of 2nd car in array
```

Java

Java does not have a record type. However, we can use a data structure. A class definition without methods is a data structure that can be used like a record. (See Chapter 27 for more about classes).

The pseudocode example of a car record described in Worked Example 26.01 can be programmed as follows:

```
class CarRecord

{
   String vehicleID;
   String registration;
   String dateOfRegistration;
   int engineSize;
   double purchasePrice;
   public CarRecord()   // declare a constructor without other methods
   {
      vehicleID = "XX";
      registration = "";
      dateOfRegistration = "01/01/2010";
      engineSize = 0;
      purchasePrice = 0.00;
   }
}

CarRecord thisCar = new CarRecord(); // instantiates a car record
thisCar.engineSize = 2500; // assign a value to a field

CarRecord[] car = new CarRecord[100]; // declare an array of car record type

car[2] = new CarRecord(); // instantiate a car
car[2].engineSize = 2500; // assign a value to a field of 2nd car in array
```

26.02 File processing

In Chapter 13 (Section 13.06) we used text files to store and read lines of text. Text files only allow us to write strings in a serial or sequential manner. We can append strings to the end of the file.

When we want to store records in a file, we create a binary file (see Chapter 16, Section 16.02). We can store records serially or sequentially. We can also store records using direct access to a **random file**. Table 26.01 lists the operations we use for processing files.

Structured English	Pseudocode
Create a file and open it for writing	`OPENFILE <filename> FOR WRITE`
Open a file in append mode	`OPENFILE <filename> FOR APPEND`
Open a file for reading	`OPENFILE <filename> FOR READ`
Open a file for random access	`OPENFILE <filename> FOR RANDOM`
Close a file	`CLOSEFILE <filename>`
Write a record to a file	`PUTRECORD <filename>, <identifier>`
Read a record from a file	`GETRECORD <filename>, <identifier>`
Move to a specific disk address within the file	`SEEK <filename>, <address>`
Test for end of file	`EOF(<filename>)`

Table 26.01 Operations for file processing

Sequential file processing

If we have an array of records, we may want to store the records on disk before the program quits, so that we don't lose the data. We can open a binary file and write one record after another to the file. We can then read the records back into the array when the program is run again.

WORKED EXAMPLE 26.02

Processing records in a sequential file

Table 26.02 shows the pseudocode for storing the car records from Worked Example 26.01 in a sequential file and accessing them.

Saving contents of array	Restoring contents of array
OPENFILE CarFile FOR WRITE FOR i ← 1 TO MaxRecords PUTRECORD CarFile, Car[i] NEXT i CLOSEFILE CarFile	OPENFILE CarFile FOR READ FOR i ← 1 TO MaxRecords GETRECORD CarFile, Car[i] NEXT i CLOSEFILE CarFile

Table 26.02 Pseudocode for processing records

Processing records sequentially in Python, VB.NET and Java

Python

```python
import pickle # this library is required to create binary files
Car = [CarRecord() for i in range(100)]

CarFile = open('CarFile.DAT', 'wb') # open file for binary write

for i in range(100):  # loop for each array element
   pickle.dump(Car[i], CarFile) # write a whole record to the binary file

CarFile.close() # close file

CarFile = open('CarFile.DAT', 'rb') # open file for binary read

Car = [] # start with empty list
while True: # check for end of file
   Car.append(pickle.load(CarFile)) # append record from file to end of list
```

```
CarFile.close()
```

VB.NET

```
Option Explicit On
Imports System.IO

Dim CarFileWriter As BinaryWriter
Dim CarFileReader As BinaryReader
Dim CarFile As FileStream
Dim Car(100) As CarRecord ' declare an array of CarRecord type
Dim i As Integer

'link file to the filename
CarFile = New FileStream("CarFile.DAT", FileMode.Create)
' create a new file and open it for writing
CarFileWriter = New BinaryWriter(CarFile)
For i = 1 To 100 ' loop for each array element
   CarFileWriter.Write(Car(i).VehicleID) ' write a field to the binary file
   CarFileWriter.Write(Car(i).Registration)
   CarFileWriter.Write(Car(i).DateOfRegistration)
   CarFileWriter.Write(Car(i).EngineSize)
   CarFileWriter.Write(Car(i).PurchasePrice)
Next

CarFileWriter.Close()   ' close file channel
CarFile.Close()

'link file to the filename
CarFile = New FileStream("CarFile.DAT", FileMode.Open)
' create a new file and open it for reading
CarFileReader = New BinaryReader(CarFile)
i = 0

' loop until end of binary file reached
i = 0
Do While CarFile.Position < CarFile.Length
   ' read fields from the binary file
   Car(i).VehicleID = CarFileReader.ReadString()
   Car(i).Registration = CarFileReader.ReadString()
   Car(i).DateOfRegistration = CarFileReader.ReadString()
   Car(i).EngineSize = CarFileReader.ReadInt32()
   Car(i).PurchasePrice = CarFileReader.ReadDecimal()
   i = i + 1
Loop

CarFileReader.Close()   'close file channel

CarFile.Close()
```

Java

writing records to a file:

```
import java.io.DataOutputStream;
import java.io.FileOutputStream;
import java.io.EOFException;
import java.io.FileInputStream;
import java.io.FileOutputStream;
   try
   {  // set up file stream and link to file name
     FileOutputStream fos = new FileOutputStream("CarFile.DAT");
     // link file stream to data stream
     DataOutputStream dos = new DataOutputStream(fos);
     for (int i = 1; i < 100; i++) // loop for each array element
     {
        dos.writeUTF(car[i].vehicleID); // write a field to the file
        dos.writeUTF(car[i].registration);
        dos.writeInt(car[i].engineSize);
        dos.writeUTF(car[i].dateOfRegistration);
        dos.writeDouble(car[i].purchasePrice);
     }
     dos.close(); // close data stream
   }
   catch (Exception x)
   {
     System.out.println("IO error");
   }
```

reading records back into array:

```
   CarRecord[] car = new CarRecord[100];
   try
   {  // set up file stream and link to file name
      FileInputStream fis = new FileInputStream("CarFile.DAT");
      // link file stream to data stream
      DataInputStream dis = new DataInputStream(fis);
      int i = 1;
      while (true) // loop for each array element
      {
         thisCar = new CarRecord();
         thisCar.vehicleID = dis.readUTF(); // read fields from the file
         thisCar.registration = dis.readUTF();
         thisCar.engineSize = dis.readInt();
         thisCar.dateOfRegistration = dis.readUTF();
         thisCar.purchasePrice = dis.readDouble();
         car[i] = thisCar; // assign record to next array element
         i += 1;
      }
   }
   catch (EOFException x)
   {
      System.out.println("End of File reached");
   }
   catch (Exception x)
   {
      System.out.println(x);  // output error message
   }
```

Alternative method using Date data type (which is an object in Java): write to file

```
import java.io.*;
import java.util.Date;
class CarRecord implements java.io.Serializable // to allow writing and reading
                                               // objects
{
   String vehicleID;
   String registration;
   Date dateOfRegistration;
   int engineSize;
   double purchasePrice;

   public CarRecord() // declare constructor
   {
      vehicleID = "XX";
      registration = "";
      dateOfRegistration = new Date();
      engineSize = 0;
      purchasePrice = 0.00;
   }
}

class Program
{
public static void main(String[] args)
{
   CarRecord[] car = new CarRecord[100];
   CarRecord thisCar;
   try
   {  //Write array to file.
      FileOutputStream fos = new FileOutputStream("cars.ser");
      ObjectOutputStream oos = new ObjectOutputStream(fos);
      for (int i = 1; i < 100; i++)  // loop for each array element
      {
         oos.writeObject(car[i].vehicleID); // write a field to the file
         oos.writeObject(car[i].registration);
         oos.writeObject(car[i].engineSize);
         oos.writeObject(car[i].dateOfRegistration);
         oos.writeObject(car[i].purchasePrice);
      }
      oos.close();
   }
   catch (EOFException x)
   {
      System.out.println("End of File reached");
   }
      catch (Exception x)
   {
      System.out.println(x);
   }
}
}
```

reading file into array:

```
CarRecord[] car = new CarRecord[100];
try
{
   //Read array from file.
   FileInputStream fis = new FileInputStream("cars.ser");
   ObjectInputStream ois = new ObjectInputStream(fis);
   int i = 1;
   while (true)
   {
      car[i].vehicleID = (String) ois.readObject(); // read the first item from the
                                      // file to the field
      car[i].registration= (String) ois.readObject();
      car[i].engineSize = (int) ois.readObject();
      car[i].dateOfRegistration = (Date) ois.readObject();
      car[i].purchasePrice = (double) ois.readObject();
      i += 1;
   }
}
catch (EOFException x)
{
   System.out.println("End of File reached");
}
catch (Exception x)
{
   System.out.println(x);
}
```

TASK 26.01

1 Write a complete program to save several car records to a sequential file.

2 Write another program to read the file and display the contents on screen.

Random-access file processing

Instead of storing records in an array, we may want to store each record in a binary file as the record is created. We can then update the record in situ (read it, change it and save it back in the same place). Note that this only works for fixed-length records. We can use a hashing function to calculate an address from the record key and store the record at the calculated address in the file (this is similar to using a hash table, see Chapter 23, Section 23.11). Just as with a hash table, collisions may occur and records need to be stored in the next free record space.

WORKED EXAMPLE 26.03

Processing records in a random-access file

Table 26.03 shows the pseudocode for storing a car record from Worked Example 26.01 in a random-access file and accessing it.

Saving a record	Retrieving a record

`OPENFILE CarFile FOR RANDOM`
`Address ← Hash(ThisCar.VehicleID)`
`SEEK CarFile, Address`
`PUTRECORD CarFile, ThisCar`
`CLOSEFILE CarFile`
	`OPENFILE CarFile FOR RANDOM`
	`Address ← Hash(ThisCar.VehicleID)`
	`SEEK CarFile, Address`
	`GETRECORD CarFile, ThisCar`
	`CLOSEFILE CarFile`

Table 26.03 Pseudocode for random-access file operations

`SEEK` moves a pointer to the given record address. The `PUTRECORD` and `GETRECORD` commands access the record at that address. After the command has been executed the pointer points to the next record in the file.

Processing random-access records in Python, VB.NET and Java

Python

```
import pickle  # this library is required to create binary files
ThisCar = CarRecord()
CarFile = open('CarFile.DAT','rb+')  # open file for binary read and write
Address = hash(ThisCar.VehicleID)
CarFile.seek(Address)
pickle.dump(ThisCar, CarFile)  # write a whole record to the binary file
CarFile.close()  # close file
```

to find a record from a given VehicleID:

```
CarFile = open('CarFile.DAT','rb')  # open file for binary read
Address = hash(VehicleID)
CarFile.seek(Address)
ThisCar = pickle.load(CarFile)  # load record from file
CarFile.close()
```

In Python, the hash function needs to allow for the record size in bytes. For example, if the record size is 58 bytes, then the second record slot starts at position 59. The *n*th record slot starts at position $(n - 1) \times 58 + 1$.

VB.NET

```
Dim CarFileWriter As BinaryWriter
Dim CarFileReader As BinaryReader
Dim CarFile As FileStream
Dim ThisCar, MyCar As CarRecord
' link the file to the filename
CarFile = New FileStream("CarFile.DAT", FileMode.Open)
' create a new file and open it for writing
CarFileWriter = New BinaryWriter(CarFile)
' get starting address for record
CarFile.Position = Hash(ThisCar.VehicleID)
' write fields to the binary file

CarFileWriter.Write(ThisCar.VehicleID)
CarFileWriter.Write(ThisCar.Registration)
CarFileWriter.Write(ThisCar.DateOfRegistration)
CarFileWriter.Write(ThisCar.EngineSize)
CarFileWriter.Write(ThisCar.PurchasePrice)
CarFileWriter.Close() 'close file channel
CarFile.Close()
```

to find a record from a given VehicleID:

```
CarFile = New FileStream("CarFile.DAT", FileMode.Open)
CarFileReader = New BinaryReader(CarFile)
' get starting address for record
CarFile.Position = Hash(VehicleID)
' read fields from the binary file
MyCar.VehicleID = CarFileReader.ReadString()
MyCar.Registration = CarFileReader.ReadString()
MyCar.DateOfRegistration = CarFileReader.ReadString()
MyCar.EngineSize = CarFileReader.ReadInt32()
MyCar.PurchasePrice = CarFileReader.ReadDecimal()
CarFileReader.Close() 'close file channel
CarFile.Close()
```

In VB.NET, the hash function needs to allow for the record size in bytes. For example, if the record size is 58 bytes, then the second record slot starts at position 59. The *n*th record slot starts at position $(n - 1) \times 58 + 1$.

Java

```
CarRecord thisCar;
int recordSize = 50;
try // set up a file with 100 dummy records
{
      RandomAccessFile writer = new RandomAccessFile("CarFile.DAT", "rw");
      for (int i = 1; i < 100; i++) // loop for each array element
      {
      thisCar = new CarRecord();
      thisCar.vehicleID = "A" + i;
      writer.seek(i * recordSize);
      writer.writeUTF(thisCar.vehicleID);
      writer.writeUTF(thisCar.registration);
      writer.writeUTF(thisCar.dateOfRegistration);
      writer.writeInt(thisCar.engineSize);
      writer.writeDouble(thisCar.purchasePrice);
      }
      writer.close();
}
catch (IOException x)
{
}
```

to find a record from a given VehicleID:

```
try
{
      RandomAccessFile writer = new RandomAccessFile("CarFile.DAT", "rw");
      RandomAccessFile reader = new RandomAccessFile("CarFile.DAT", "r");
      reader.seek(hash(vehicleID));
      thisCar = new CarRecord();
      thisCar.vehicleID = reader.readUTF();
      thisCar.registration = reader.readUTF();
      thisCar.engineSize = reader.readInt();
      thisCar.dateOfRegistration = reader.readUTF();
      thisCar.purchasePrice = reader.readDouble();
      reader.close();
}
catch (IOException x)
{
}
```

TASK 26.02

Write a complete program to save several car records to a random-access file. Write another program to find a record in the random-access file using the record key. Display the record data on screen.

26.03 Exception handling

Run-time errors can occur for many reasons. Some examples are division by zero, invalid array index or trying to open a non-existent file. Run-time errors are called ‘exceptions’. They can be handled (resolved) with an error subroutine (known as an ‘exception handler’), rather than let the program crash.

Using pseudocode, the error-handling structure is:

```
TRY
   <statementsA>
EXCEPT
   <statementsB>
ENDTRY
```

Any run-time error that occurs during the execution of `<statementsA>` is caught and handled by executing `<statementsB>`. There can be more than one `EXCEPT` block, each handling a different type of exception. Sometimes a `FINALLY` block follows the exception handlers. The statements in this block will be executed regardless of whether there was an exception or not.

VB.NET is designed to treat exceptions as abnormal and unpredictable erroneous situations. Python is designed to use exception handling as flow-control structures. You may find you need to include exception handling in the code for Worked Example 26.02. Otherwise the end of file is encountered and the program crashes.

Python distinguishes between different types of exception, such as:

- `IOError`: for example, a file cannot be opened
- `ImportError`: Python cannot find the module
- `ValueError`: an argument has an inappropriate value
- `KeyboardInterrupt`: the user presses Ctrl+C or Ctrl+Del
- `EOFError`: a file-read meets an end-of-file condition
- `ZeroDivisionError`: a division by zero has been attempted

Java distinguishes between different types of exception, such as:

- `IOException`: for example, a file cannot be opened
- `ArithmeticException`: an arithmetic error occurred, such as division by zero
- `ArrayIndexOutOfBoundsException`: the program tries to access an array element outside the boundary

WORKED EXAMPLE 26.04

Here is a simple example of exception handling. Asking the user to key in an integer could result in a non-integer input. This should not crash the program.

Python

```
NumberString = input("Enter an integer: ")
try:
   n = int(NumberString)
   print(n)
except:
   print("This was not an integer")
```

VB.NET

```
Dim NumberString As String
Dim n As Integer
Console.WriteLine("Enter an integer")
NumberString = Console.ReadLine()
Try
```

```
   n = Int(NumberString)
   Console.WriteLine(n)
Catch
   Console.WriteLine("This was not an integer")
End Try
```

Java

```
import java.util.Scanner;
{
   public static void main(String[] args)
   {
      Scanner console = new Scanner(System.in);
      try
      {
         System.out.print("Enter an integer: ");
         int n = console.nextInt();
         System.out.println(n);
      }
      catch(Exception e) // catches any exception
      {
         System.out.println("This was not an integer");
      }
   }
}
```

TASK 26.03

Add exception-handling code to your programs for Task 26.01 or Task 26.02. Test your code handles exceptions without the program crashing.

Reflection Point:

Can you explain the difference between serial, sequential and random access files? Give an example of when each is appropriate.

Summary

- Records are user-defined types.
- Records can be stored in files in a serial, sequential or random (direct access) manner.
- Exception handling is advisable to avoid program crashes due to run-time errors.

Exam-style Questions

1 A company stores details about their customers in a binary file of records.

- The key field of a customer record is the customer ID (a number between 100001 and 999999).
- The name of the customer is stored in a 30-character field.
- The customer's telephone number is stored in a 14-character field.
- The total value of orders so far is stored in a currency (decimal) field.

a **i** Write the declaration statement for the record data type `CustomerRecord` required to store the data. Write program code. [6]

ii Write the declaration statement for an array `CustomerData[0 : 999]` to store customer records. [2]

b The array `CustomerData` is to be used as a hash table to store customer records. The function `Hash` is used to calculate the address where a record is to be stored.

```
FUNCTION Hash(CustomerID : INTEGER) RETURNS INTEGER
    Address ← CustomerID MOD 1000
    RETURN Address
ENDFUNCTION
```

i Write program code to implement the function `Hash`. [3]

ii Write a procedure `AddRecord(Customer : CustomerRecord)` to add a customer record to the hash table `CustomerData`. Your solution should handle collisions by using the next available slot in the hash table. [7]

iii Write a function `FindRecord(CustomerID : INTEGER)` that returns the index of the hash table slot where the record for the customer with `CustomerID` is stored. [7]

c Before the program stops, the hash table records must be stored in a sequential file, so that the records can be restored to the array when the program is re-entered.

Write program code to store the records of the array CustomerData sequentially into a binary file CustomerData.DAT [6]

d Instead of using a hash table, the company decide they want to store customer records in a direct-access binary file.

Explain what changes need to be made to your program to do this. [6]

2 A program allows a user to enter a filename for accessing a data file. If the user types in a filename that does not exist, the program crashes. Write program code that includes exception handling to replace the following pseudocode: [5]

```
OUTPUT "Which file do you want to use? "
INPUT FileName
OPENFILE FileName FOR RANDOM
```

Chapter 27:
Object-Oriented Programming (OOP)

Learning objectives

By the end of this chapter you should be able to:

- show understanding of the terminology associated with OOP: objects, properties, methods, classes, inheritance, polymorphism, containment/aggregation, encapsulation, getters, setters and instances
- show understanding of how to solve a problem by designing appropriate classes
- show understanding of and write code that demonstrates the use of OOP.

27.01 Concept of OOP

Chapters 14 and 26 covered programming using the procedural aspect of our programming languages. Procedural programming groups related programming statements into subroutines. Related data items are grouped together into record data structures. To use a record variable, we first define a record type. Then we declare variables of that record type.

Example of using a record

A car manufacturer and seller wants to store details about cars. These details can be stored in a record structure (see Chapter 16, Section 16.01 and Chapter 26, Section 26.02):

```
TYPE CarRecord
    DECLARE VehicleID            : STRING
    DECLARE Registration         : STRING
    DECLARE DateOfRegistration   : DATE
    DECLARE EngineSize           : INTEGER
    DECLARE PurchasePrice        : CURRENCY
ENDTYPE
```

We can write program code to access and assign values to the fields of this record. For example:

```
PROCEDURE UpdateRegistration(BYREF ThisCar : CarRecord, BYVALUE NewRegistration)
    ThisCar.Registration ← NewRegistratio
ENDPROCEDURE
```

We can call this procedure from anywhere in our program. This seems a well-regulated way of operating on the data record. However, we can also access the record fields directly from anywhere within the scope of `ThisCar`:

```
ThisCar.EngineSize ← 2500
```

Classes in OOP

OOP goes one step further and groups together the data structure and the subroutines that operate on the data items in this data structure. Such a group is called an **object**. The data of an object are called **attributes** and the subroutines acting on the attributes are called **methods**. The idea behind OOP is that attributes can only be accessed through methods. The direct path to the data is unavailable. Attributes are referred to as ‘private’. The methods to access the data are made available to programmers, so these are ‘public’. The feature of data being combined with the subroutines acting on this data is known as **encapsulation**. To use an object, we first define an object type. An object type is called a **class**.

Classes are templates for objects. When a class type has been defined it can be used to create one or more objects of this class type. Therefore, an object is an instance of a class.

The first stage of writing an object-oriented program to solve a problem is to design the classes. This is part of object-oriented design. From this design, a program can be written using an object-oriented programming (OOP) language.

The programming languages the syllabus prescribes can be used for OOP: Python 3, VB.NET and Java.

Advantages of OOP over procedural languages

The advantage of OOP is that it produces robust, more reliable code. The attributes can only be manipulated using methods provided by the class definition. This means the attributes are protected from accidental changes. Classes provided in module libraries are thoroughly tested. If you use tried and tested building blocks to construct your program, you are less likely to introduce bugs than when you write code from scratch.

27.02 Designing classes and objects

When designing a class, we need to think about the attributes we want to store. We also need to think about the methods we need to access the data and assign values to the data of an object. A data type is a blueprint when declaring a variable of that data type. A class definition is a blueprint when declaring an object of that class. Creating a new object is known as ‘instantiation’.

Any data that is held about an object must be accessible, otherwise there is no point in storing it. We therefore need methods to access each one of these attributes. These methods are usually referred to as **getters**. They get an attribute of the object.

When we first set up an object of a particular class, we use a constructor. A **constructor** instantiates the object and assigns initial values to the attributes.

Any attributes that might be updated after instantiation will need subroutines to update their values. These are referred to as **setters**. Some attributes get set only at instantiation. These don’t need setters. This makes an object more robust, because you cannot change attributes that were not designed to be changed.

Python and VB.NET allow properties to be declared. A **property** combines the attribute with its associated setter and/or getter (see Figures 27.03 and 27.05).

WORKED EXAMPLE 27.01

Creating a class

Consider the car data from Section 27.01.

When a car is manufactured it is given a unique vehicle ID that will remain the same throughout the car’s existence. The engine size of the car is fixed at the time of manufacture. The registration ID will be given to the car when the car is sold.

In our program, when a car is manufactured, we want to create a new car object. We need to instantiate it using the constructor. Any attributes that are already known at the time of instantiation can be set with the constructor. In our example, `VehicleID` and `EngineSize` can be set by the constructor. The other attributes are assigned values at the time of purchase and registration. So, we need setters for them. The identifier table for the `Car` class is shown in Table 27.01.

Identifier	Data Type	Description
`Car`	`Class`	Class identifier
`VehicleID`	`STRING`	Unique ID assigned at time of manufacture
`Registration`	`STRING`	Unique ID assigned after time of purchase
`DateOfRegistration`	`DATE`	Date of registration
`EngineSize`	`INTEGER`	Engine size assigned at time of manufacture
`PurchasePrice`	`CURRENCY`	Purchase price assigned at time of purchase
`Constructor()`		Method to create a `Car` object and set properties assigned at manufacture
`SetPurchasePrice()`		Method to assign purchase price at time of purchase
`SetRegistration()`		Method to assign registration ID
`SetDateOfRegistration()`		Method to assign date of registration
`GetVehicleID()`		Method to access vehicle ID
`GetRegistration()`		Method to access registration ID

`GetDateOfRegistration()`		Method to access date of registration
`GetEngineSize()`		Method to access engine size
`GetPurchasePrice()`		Method to access purchase price

Table 27.01 Identifier table for `Car` class

We can represent this information as a class diagram in Figure 27.01.

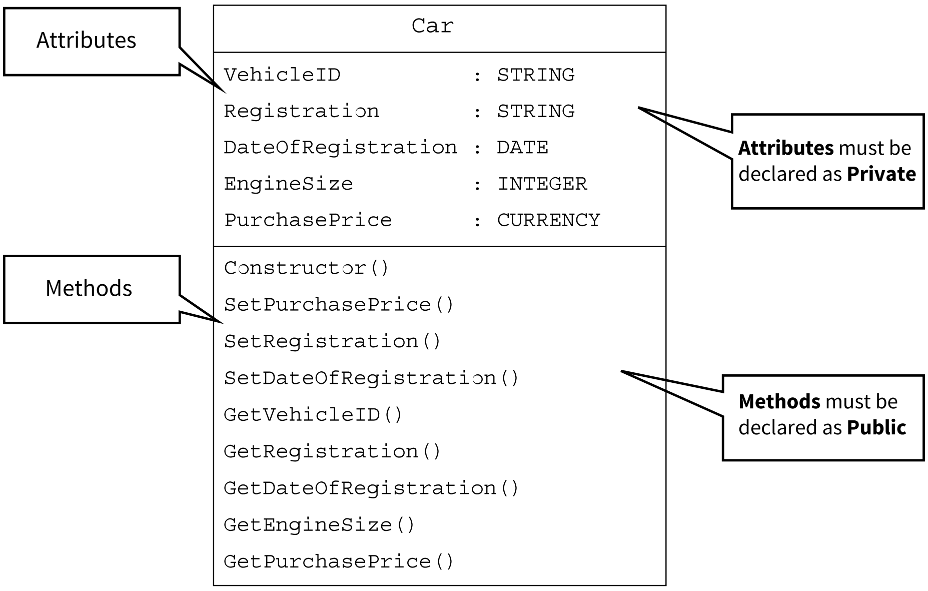


Figure 27.01 Car class diagram

27.03 Writing object-oriented code

Declaring a class

Attributes should always be declared as ‘Private’. This means they can only be accessed through the class methods. So that the methods can be called from the main program, they have to be declared as ‘Public’. There are other modifiers (such as ‘Protected’), but they are beyond the scope of this book.

The syntax for declaring classes is quite different for the different programming languages. We will look at the three chosen languages. You are expected to write programs in one of these.

Declaring a class in Python

The code below shows how a constructor, getters and setters can be declared in Python.

Two underscore characters are required before and after **init** to define the constructor

Self is the first parameter in the parameter list for every method

Python

```
class Car:
    def __init__(self, n, e):       # constructor
       self.__VehicleID = n
       self.__Registration = “”
       self.__DateOfRegistration = None
       self.__EngineSize = e
       self.__PurchasePrice = 0.00

    def SetPurchasePrice(self, p):
       self.__PurchasePrice = p
    def SetRegistration(self, r):
       self.__Registration = r
    def SetDateOfRegistration(self, d):
       self.__DateOfRegistration = d

    def GetVehicleID ID(self) :
       return(self.__VehicleID)

    def GetRegistration(self):
       return(self.__Registration)

    def GetDateOfRegistration(self):
       return(self.__DateOfRegistration)

    def GetEngineSize(self):
       return(self.__EngineSize)

    def GetPurchasePrice(self):
       return(self.__PurchasePrice)
```

Two underscore characters before an attribute name signify it is private

Python also supports properties:

Python using properties

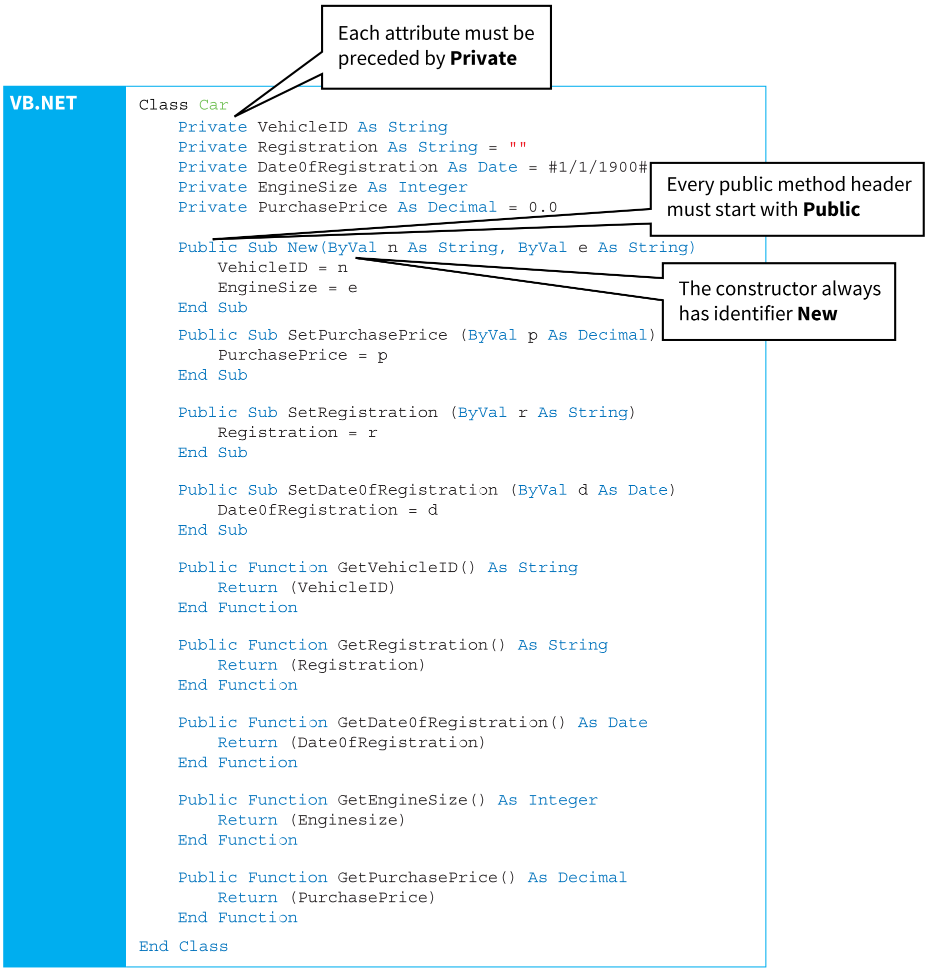
```
class Car:
    def __init__(self, n, e):    # constructor
       self.__VehicleID = n
       self.__Registration = ""
       self.__DateOfRegistration = None
       self.__EngineSize = e
       self.__PurchasePrice = 0.00

    @property
    def VehicleID(self):
       return(self.__VehicleID)

    @property
    def EngineSize(self):
       return(self.__EngineSize)

    @property
```

```
def Registration(self):
   return(self.__Registration)

@Registration.setter
def Registration(self, r):
   self.__Registration = r

@property
def DateOfRegistration(self):
   return(self.__DateOfRegistration)

@DataOfRegistration.setter
def DateOfRegistration(self, d):
   self.__DateOfRegistration = d

@property
def PurchasePrice(self):
   return(self.__PurchasePrice)

@PurchasePrice.setter
def PurchasePrice(self, p):
   self.__PurchasePrice = p
```

Note that not all properties provide a setter, only where attribute values are to be changed.

Declaring a class in VB.NET

The code below shows how attributes, the constructor, getters and setters can be declared in VB.NET.



Each attribute must be preceded by **Private**

Every public method header must start with **Public**

The constructor always has identifier **New**

VB.NET

```
Class Car
    Private VehicleID As String
    Private Registration As String = ""
    Private DateOfRegistration As Date = #1/1/1900#
    Private EngineSize As Integer
    Private PurchasePrice As Decimal = 0.0

    Public Sub New(ByVal n As String, ByVal e As String)
        VehicleID = n
        EngineSize = e
    End Sub

    Public Sub SetPurchasePrice (ByVal p As Decimal)
        PurchasePrice = p
    End Sub

    Public Sub SetRegistration (ByVal r As String)
        Registration = r
    End Sub

    Public Sub SetDateOfRegistration (ByVal d As Date)
        DateOfRegistration = d
    End Sub

    Public Function GetVehicleID() As String
        Return (VehicleID)
    End Function

    Public Function GetRegistration() As String
        Return (Registration)
    End Function

    Public Function GetDateOfRegistration() As Date
        Return (DateOfRegistration)
    End Function

    Public Function GetEngineSize() As Integer
        Return (Enginesize)
    End Function

    Public Function GetPurchasePrice() As Decimal
        Return (PurchasePrice)
    End Function

End Class
```

VB.NET also supports properties:

VB using properties

```
Module Module1

Class Car
    Public Property VehicleID() As String
    Public Property Registration() As String = ""
    Public Property DateOfRegistration() As Date = #1/1/1900#
```

```
   Public Property EngineSize() As Integer
   Public Property PurchasePrice() As Decimal = 0.0

   Public Sub New(ByVal n As String, ByVal e As String)
       VehicleID = n
       EngineSize = e
   End Sub

End Class
```

Note that getters and setters are automatically available for any property.
If some properties should be read-only, getters and setters have to be declared explicitly:

```
Class Car
  Private _VehicleID As String
  Public ReadOnly Property VehicleID() As String
    Get
      Return _VehicleID
    End Get
  End Property

  Public Property Registration() As String
  Public Property DateOfRegistration() As Date

  Private _EngineSize As Integer
  Public ReadOnly Property EngineSize() As Integer
    Get
      Return _EngineSize
    End Get
  End Property

  Public Property PurchasePrice() As Decimal
  Public Sub New(ByVal n As String, ByVal e As String)
    _VehicleID = n
    _Registration = ""
    _DateOfRegistration = #1/1/1900#
    _EngineSize = e
    _PurchasePrice = 0.0
  End Sub

End Class
```

Declaring a class in Java

The code below shows how attributes, the constructor and methods are declared in Java.

Java

```
import java.util.Date;

class Car

{
   private String vehicleID;
   private String registration;
   private Date dateOfRegistration;
   private int engineSize;
   private float purchasePrice;

   public Car(String n, int e)  // constructor
   {
      vehicleID = n;
      registration = "";
      dateOfRegistration = new Date();
      engineSize = e;
      purchasePrice = 0;
   }
   public void setPurchasePrice(float p)
   {
      purchasePrice = p;
   }
   public void setRegistration(String r)
   {
      registration = r;
   }
   public void setDateOfRegistration(Date d)
   {
      dateOfRegistration = d;
   }
   public String getVehicleID()
   {
      return(vehicleID);
   }
   public String getRegistration()
   {
      return(registration);
   }
   public Date getDateOfRegistration()
   {
      return(dateOfRegistration);
   }
   public int getEngineSize()
   {
      return(engineSize);
   }
   public float getPurchasePrice()
   {
      return(purchasePrice);
   }
}
```

Instantiating a class

To use an object of a class type in a program the object must first be instantiated. This means the memory space must be reserved to store the attributes.

The following code instantiates an object `ThisCar` of class `Car`.

Python	`ThisCar = Car("ABC1234", 2500)`
VB.NET	`Dim ThisCar As New Car("ABC1234", 2500)`
Java	`Car thisCar = new Car("ABC1234", 2500);`

Using a method

To call a method in program code, the object identifier is followed by the method identifier and the parameter list.

The following code sets the purchase price for an object `ThisCar` of class `Car`.

Python	`ThisCar.SetPurchasePrice(12000)`
	`ThisCar.PurchasePrice = 12000 # using properties`

VB.NET	`ThisCar.SetPurchasePrice(12000)`
	`ThisCar.PurchasePrice = 12000 ' using properties`

Java	`thisCar.setPurchasePrice(12000);`

The following code gets and prints the vehicle ID for an object `ThisCar` of class `Car`.

Python	`print(ThisCar.GetVehicleID())`
	`print(ThisCar.VehicleID) # using properties`

VB.NET	`Console.WriteLine(ThisCar.GetVehicleID())`
	`Console.WriteLine(ThisCar.VehicleID) 'using properties`

Java	`System.out.print(thisCar.getVehicleID());`

TASK 27.01

1 Copy the `Car` class definition into your program editor and write a simple program to test that each method works.

2 A business wants to store data about companies they supply. The data to be stored includes: company name, email address, date of last contact.

   a Design a class `Company` and draw a class diagram.

   b Write program code to declare the class. Company name and email address are to be set by the constructor and will never be changed.

   c Instantiate one object of this class and test your class code works.

27.04 Inheritance

The advantage of OOP is that we can design a class (a base class or a superclass) and then derive further classes (subclasses) from this base class. This means that we write the code for the base class only once and the subclasses make use of the attributes and methods of the base class, as well as having their own attributes and methods. This is known as **inheritance** and can be represented by an inheritance diagram (Figure 27.02).

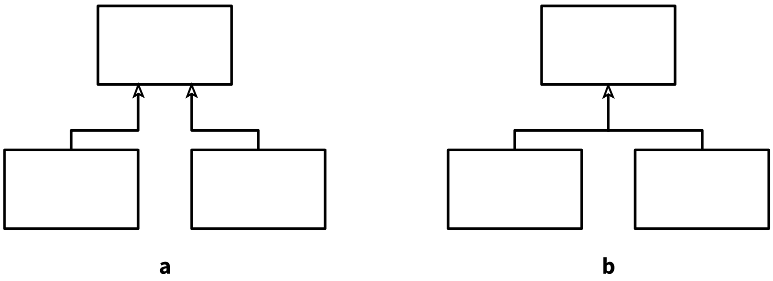


Figure 27.02 Inheritance diagram (a) standard and (b) alternative

WORKED EXAMPLE 27.02

Implementing a library system

Consider the following problem.

- A college library has items for loan.
- The items are currently books and CDs.
- Items can be borrowed for three weeks.
- If a book is on loan, it can be requested by another borrower.

Table 27.02 shows the information to be stored.

Library item	
Book	**CD**
Title of book*	Title of CD*
Author of book*	Artist of CD*
Unique library reference number*	Unique library reference number*
Whether it is on loan*	Whether it is on loan*
The date the book is due for return*	The date the CD is due for return*
Whether the book is requested by another borrower	The type of music on the CD (genre)

Table 27.02 Library system information

The information to be stored about books and CDs needs further analysis. Note that we could have a variable `Title`, which stores the book title or the CD title, depending on which type of library item we are working with. There are further similarities (marked * in Table 27.02).

There are some items of data that are different for books and CDs. Books can be requested by a borrower. For CDs, the genre is to be stored.

We can define a class `LibraryItem` and derive a `Book` class and a `CD` class from it. We can draw the inheritance diagrams for the `LibraryItem`, `Book` and `CD` classes as in Figure 27.03.

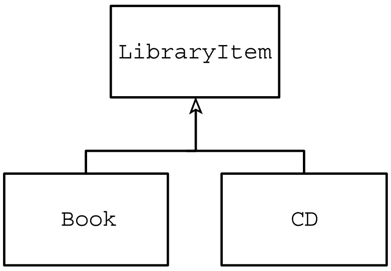


Figure 27.03 Inheritance diagram for LibraryItem, Book and CD classes

Analysing the attributes and methods required for all library items and those only required for books and only for CDs, we arrive at the class diagram in Figure 27.04.

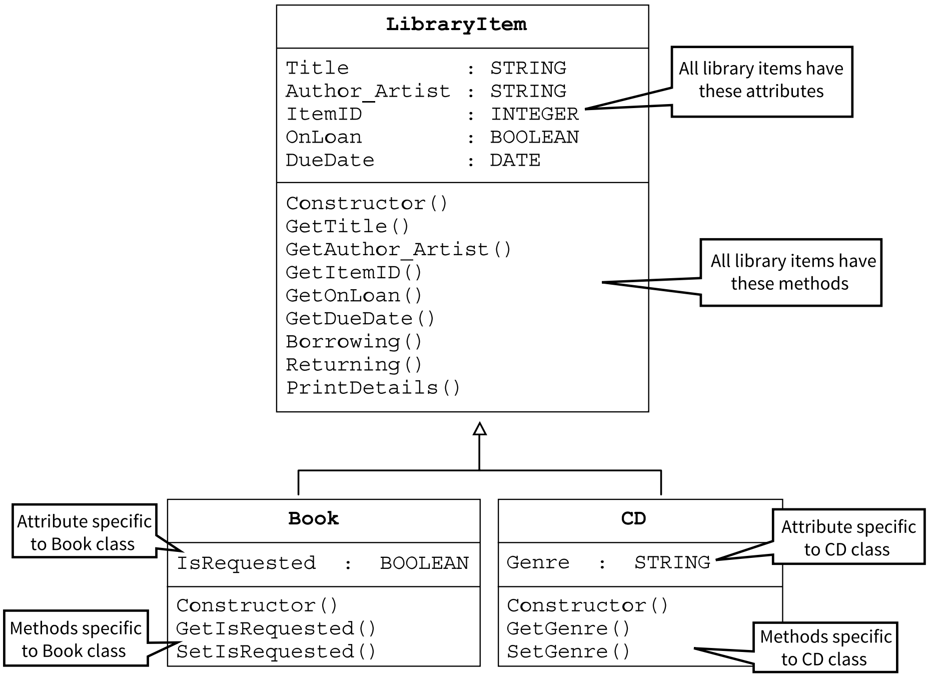


Figure 27.04 Class diagram for LibraryItem, Book and CD

A base class that is never used to create objects directly is known as an **abstract class**. `LibraryItem` is an abstract class.

Declaring a base class and derived classes (subclasses) in Python

The code below shows how a base class and its subclasses are declared in Python.

Python

The base class definition

```
import datetime
class LibraryItem:

    def __ init__(self, t, a, i):          # initialiser method
       self.__Title = t
       self.__Author__Artist = a
       self.__ItemID = i
       self.__OnLoan = False
       self.__DueDate = datetime.date.today()

    def GetTitle(self):
       return(melf.__Title)

# other Get methods go here

    def Borrowing(self):
       self.__OnLoan = True
       self.__DueDate = self.__DueDate + datetime.timedelta(weeks-3)

    def Returning(self):
       self.__OnLoan - False

    def PrintDetails(self):
       print(self.__Title,',', self.__Author__Artist,',')
       print(self.__ItemID',', self.__OnLoan,',',Self,__DueDate)
class Book(Libraryltem):
```

A subclass definition

```
    def __init__(self, t, a, i):           # initialiser method
       Libraryltem.__init__(self, t, a, i)
       self.__IsRequested = False
       self.__RequestedBy = 0
```

This statement calls the constructor for the base class

```
    def GetIsRequested(self):
       return(self.__IsRequested)

    def SetLiRequested(self):
       self.__IsRequested = True

class CD(LibraryItem):

    def __init__(self, t, a, i):           # initialiser method
       Libraryltem.__init__(self, t, a, i)
       self.__Genre = "

    def GetGestre(self):
       return(self.__Genre)

    def SetGenre(self, g):
       self.__Genre = g
```

Python using properties

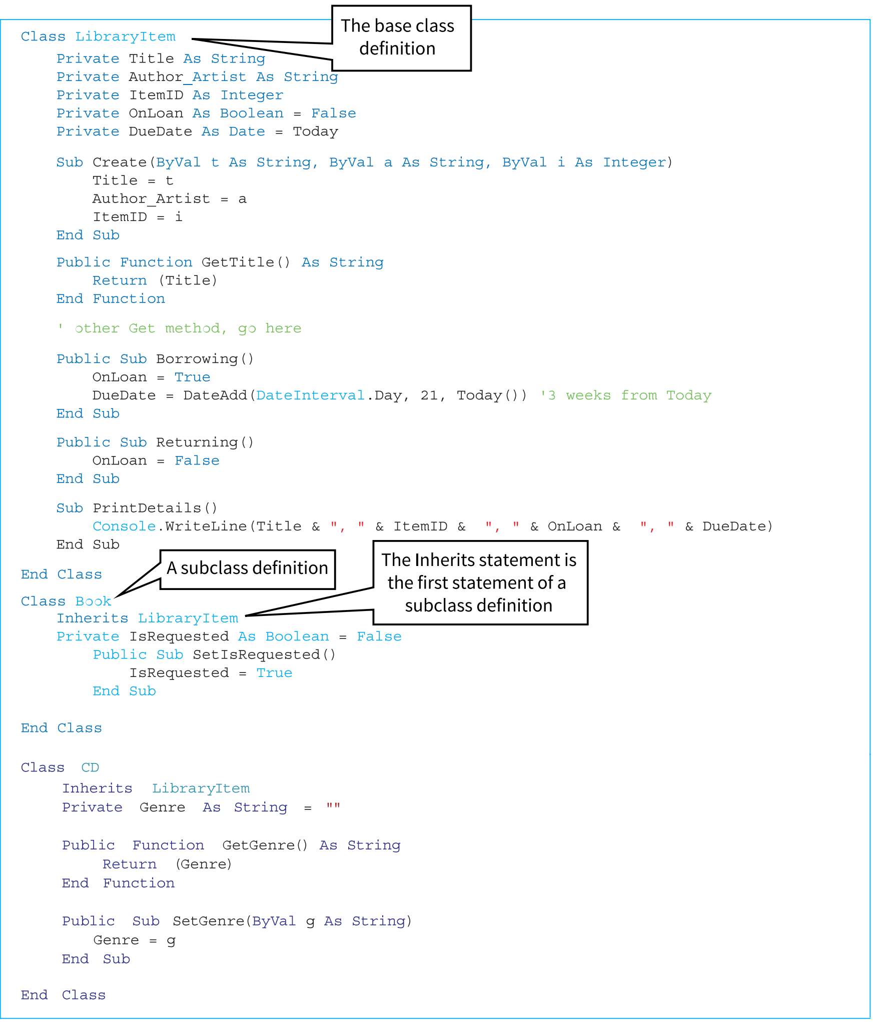
```
class LibraryItem:

  def __init__(self, t, a, i):    # constructor / initialiser method
      self.__Title = t
      self.__Author__Artist = a
      self.__ItemID = i
      self.__OnLoan = False
      self.__DueDate = datetime.date.today()

  @property
  def Title(self):
      return(self.__Title)

# other getters go here

  def Borrowing(self):
      self.__OnLoan = True
      self.__DueDate = self.__DueDate + datetime.timedelta(weeks=3)

  def Returning(self):
      self.__OnLoan = False

  def PrintDetails(self):
      print(self.__Title,'; ', self.__Author__Artist,'; ', end='')
      print(self.__ItemID,'; ', self.__OnLoan,'; ', self.__DueDate)

class Book(LibraryItem):
  def __init__(self, t, a, i):    # initialiser method
      LibraryItem.__init__(self, t, a, i)
      self.__IsRequested = False
      self.__RequestedBy = 0
```

```
    @property
    def IsRequested(self):
        return(self.__IsRequested)

    @IsRequested.setter
    def IsRequested(self, b):
        self.__IsRequested = b

    # print details method for Book

    def PrintDetails(self):
        print("Book Details")
        LibraryItem.PrintDetails(self)
        print(self.__IsRequested)

class CD(LibraryItem):

    def __init __(self, t, a, i):     # initialiser method
        LibraryItem.__init__(self, t, a, i)
        self.__Genre = ""

    @property
     def Genre(self):
        return(self.__Genre)

     @Genre.setter
     def Genre(self, g):
        self.__Genre = g
```

Declaring a base class and derived classes (subclasses) in VB.NET

The code below shows how a base class and its subclasses are declared in VB.NET.

VB.NET



```
Class LibraryItem
    Private Title As String
    Private Author_Artist As String
    Private ItemID As Integer
    Private OnLoan As Boolean = False
    Private DueDate As Date = Today

    Sub Create(ByVal t As String, ByVal a As String, ByVal i As Integer)
        Title = t
        Author_Artist = a
        ItemID = i
    End Sub

    Public Function GetTitle() As String
        Return (Title)
    End Function

    ' other Get method, go here

    Public Sub Borrowing()
        OnLoan = True
        DueDate = DateAdd(DateInterval.Day, 21, Today()) '3 weeks from Today
    End Sub

    Public Sub Returning()
        OnLoan = False
    End Sub

    Sub PrintDetails()
        Console.WriteLine(Title & ", " & ItemID &  ", " & OnLoan &  ", " & DueDate)
    End Sub

End Class

Class Book
    Inherits LibraryItem
    Private IsRequested As Boolean = False
        Public Sub SetIsRequested()
            IsRequested = True
        End Sub

End Class

Class  CD
     Inherits  LibraryItem
     Private  Genre  As  String  =  ""

     Public  Function  GetGenre() As String
          Return  (Genre)
     End  Function

     Public  Sub  SetGenre(ByVal g As String)
         Genre = g
     End  Sub

End  Class
```

The base class definition
A subclass definition
The Inherits statement is the first statement of a subclass definition

VB.NET using properties

```
class LibraryItem
    Public Property Title()
    Public Property Author_Artist As String
    Public Property ItemID As Integer
    Public Property OnLoan As Boolean = False
    Public Property DueDate As Date = Today

   Sub Create (ByVal t As String, ByVal a As String, ByVal i As Integer)
      Title = t
      Author_Artist = a
      ItemID = i
    End Sub

   Public Sub Borrowing()
      OnLoan = True
      DueDate = DateAdd(DateInterval.Day, 21, Today()) '3 weeks from today
    End Sub

   Public Sub Returning()
      _OnLoan = False
    End Sub

   Public Sub PrintDetails()
      Console.WriteLine(Title & "; " & ItemID & "; " & OnLoan & "; " &
      DueDate)
    End Sub
  End Class

  Class Book
    Inherits LibraryItem
    Public Property IsRequested() = False
  End Class

  Class CD
    Inherits LibraryItem
    Public Property Genre() As String
  End Class
```

Declaring a base class and derived classes (subclasses) in Java

The code below shows how a base class and its subclasses are declared in Java.

Java

```
import java.util.Date;

class LibraryItem

{
   private String title;
   private String author_Artist;
   private int itemID;
   private Boolean onLoan;
   private Date dueDate;
   public LibraryItem(String t, String a, int i)  // constructor
   {
      title = t;
      author_Artist = a;
      itemID = i;
      onLoan = false;
      dueDate = new Date();
   }
   public String GetTitle()
   {
      return(title);
   }
// other Get methods go here
   public void borrowing()
   {
      onLoan = true;
      //dueDate = dueDate + 21;
   }
   public void returning()
   {
      onLoan = false;
   }
   public void printDetails()
   {
      System.out.print(title + " ; " + author_Artist + " ; ");
      System.out.println(itemID + " ; " + onLoan + " ; " + dueDate);
   }
}
class Book extends LibraryItem
{
   private Boolean isRequested;
   private int requestedBy;
   public Book(String t, String a, int i)  // constructor
   {
      super(t, a, i);
      isRequested = false;
      requestedBy = 0;
   }
   public Boolean getIsRequested()
   {
      return(isRequested);
   }
   public void setIsRequested()
   {
      isRequested = true;
   }
}
class CD extends LibraryItem
{
   private String genre;
   public CD(String t, String a, int i)  // constructor
   {
      super(t, a, i);
      genre = "";
   }
   public String getGenre()
   {
      return(genre);
   }
   public void setGenre(String g)
   {
      genre = g;
   }
}
```

Instantiating a subclass

Creating an object of a subclass is done in the same way as with any class (See Section 27.03).

Python	`ThisBook = Book(Title, Author, ItemID)`
	`ThisCD = CD(Title, Artist, ItemID)`

VB.NET	`Dim ThisBook As New Book()`
	`Dim ThisCD As New CD()`
	`ThisBook.Create(Title, Author, ItemID)`
	`ThisCD.Create(Title, Artist, ItemID)`

Java	`Book thisBook = new Book("Computing", "Sylvia", 1234);`
	`CD thisCD = new CD("Let it be", "Beatles", 2345);`

Using a method

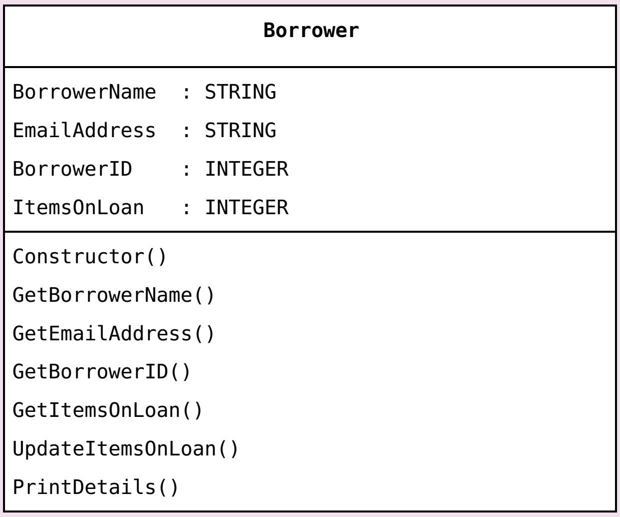
Using an object created from a subclass is exactly the same as an object created from any class.

TASK 27.02

Copy the class definitions for `LibraryItem`, `Book` and `CD` into your program editor. Write the additional get methods. Write a simple program to test that each method works.

TASK 27.03

Write code to define a Borrower class as shown in the class diagram in Figure 27.05.



```
Borrower

BorrowerName  : STRING
EmailAddress  : STRING
BorrowerID    : INTEGER
ItemsOnLoan   : INTEGER

Constructor()
GetBorrowerName()
GetEmailAddress()
GetBorrowerID()
GetItemsOnLoan()
UpdateItemsOnLoan()
PrintDetails()
```

Figure 27.05 Borrower class diagram

The constructor should initialise `ItemsOnLoan` to `0`.

`UpdateItemsOnLoan()` should increment `ItemsOnLoan` by an integer passed as parameter.

Write a simple program to test the methods.

27.05 Polymorphism

Look at Worked Example 27.02 and the code that implements it in Section 27.04. The constructor method of the base class is redefined in the subclasses. The constructor for the `Book` class calls the constructor of the `LibraryItem` class and also initialises the `IsRequested` attribute. The constructor for the `CD` class calls the constructor of the `LibraryItem` class and also initialises the `Genre` attribute.

The `PrintDetails` method is currently only defined in the base class. This means we can only get information on the attributes that are part of the base class. To include the additional attributes from the subclass, we need to declare the method again. Although the method in the subclass will have the same identifier as in the base class, the method will actually behave differently. This is known as **polymorphism**.

The way the programming languages re-define a method varies.

The code shown here includes a call to the base class method with the same name. You can completely re-write the method if required.

Python

```
# define the print details method for Book
def PrintDetails(self):
   print("Book Details")
   LibraryItem.PrintDetails(self)
   print(self.__IsRequested)
```

This line calls the base class method with the same name.

VB.NET

```
' in base class, add the keyword Overridable
' to the method to be redefined
Overridable Sub PrintDetails()

' in subclass, add the redefined method:
Public Overrides Sub PrintDetails()
   Console.WriteLine("Book Details")
   MyBase.PrintDetails()
   Console.WriteLine(IsRequested)

End Sub
```

This line calls the base class method with the same name.

Java

```
@Override
public void printDetails()
{
   System.out.println("Book Details");
   super.printDetails();
   System.out.println(isRequested);
}
```

This line calls the base class method with the same name.

TASK 27.04

Use your program code from Task 27.02. Re-define the `PrintDetail` methods for the `Book` class and the `CD` class. Test your code.

TASK 27.05

Use your program code from Task 27.03. Add another attribute, `BorrowerID`, to the `LibraryItem` class so that the item being loaned can have the borrower recorded.

Change the `LibraryItem.Borrowing` and `LibraryItem.Returning methods, so that LoanItem.BorrowerID` and `Borrower.ItemsOnLoan` are updated when a library item is borrowed or returned.

TASK 27.06

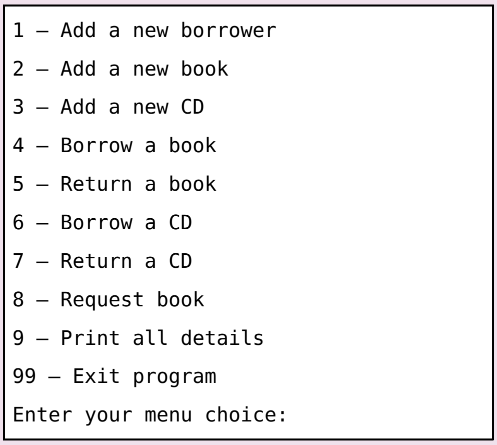
Use your code from Task 27.02 or Task 27.04. Add another attribute, `RequestedBy`, to the `Book` class so that the borrower making the request can be recorded.

Change the method `Book.SetIsRequested`, so that `Book.RequestedBy` is updated when a book is requested.

TASK 27.07

Use your code from Task 27.06 to write the complete program to implement a simplified library system.

Write code to provide the user with a menu to choose an option. An example of a menu that would be suitable is shown in Figure 27.06.



```
1 – Add a new borrower
2 – Add a new book
3 – Add a new CD
4 – Borrow a book
5 – Return a book
6 – Borrow a CD
7 – Return a CD
8 – Request book
9 – Print all details
99 – Exit program
Enter your menu choice:
```

Figure 27.06 Library system menu

27.06 Garbage collection

When objects are created they occupy memory. When they are no longer needed, they should be made to release that memory, so it can be re-used. If objects do not let go of memory, we eventually end up with no free memory when we try and run a program. This is known as ‘memory leakage’.

How do our programming languages handle this?

Python	Memory management involves a private heap containing all Python objects and data structures. The management of the Python heap is performed by the interpreter itself. The programmer does not need to do any housekeeping.
VB.NET	A garbage collector automatically reclaims memory from objects that are no longer referred to by the running program.
Java	The Java runtime environment deletes objects when it determines that they are no longer being used.

Table 27.03 Garbage collection strategies

27.07 Containment (aggregation)

In Section 27.04 we covered how a subclass inherits from a base class. This can be seen as generalisation and specialisation. The base class is the most general class, subclasses derived from this base class are more specialised.

We have other kinds of relationships between classes. **Containment** means that one class contains other classes. For example, a car is made up of different parts and each part will be an object based on a class. The wheels are objects of a different class to the engine object. The engine is also made up of different parts. Together, all these parts make up one big object.

The containment relationship is shown in Figure 27.07.

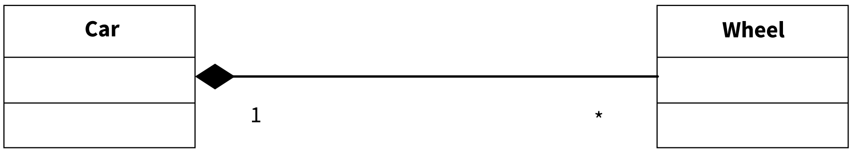


Figure 27.07 Containment (aggregation) class diagram

WORKED EXAMPLE 27.03

Using containment

A college runs courses of up to 50 lessons. A course may end with an assessment. Object-oriented programming is to be used to set up courses. The classes required are shown in Figure 27.08.

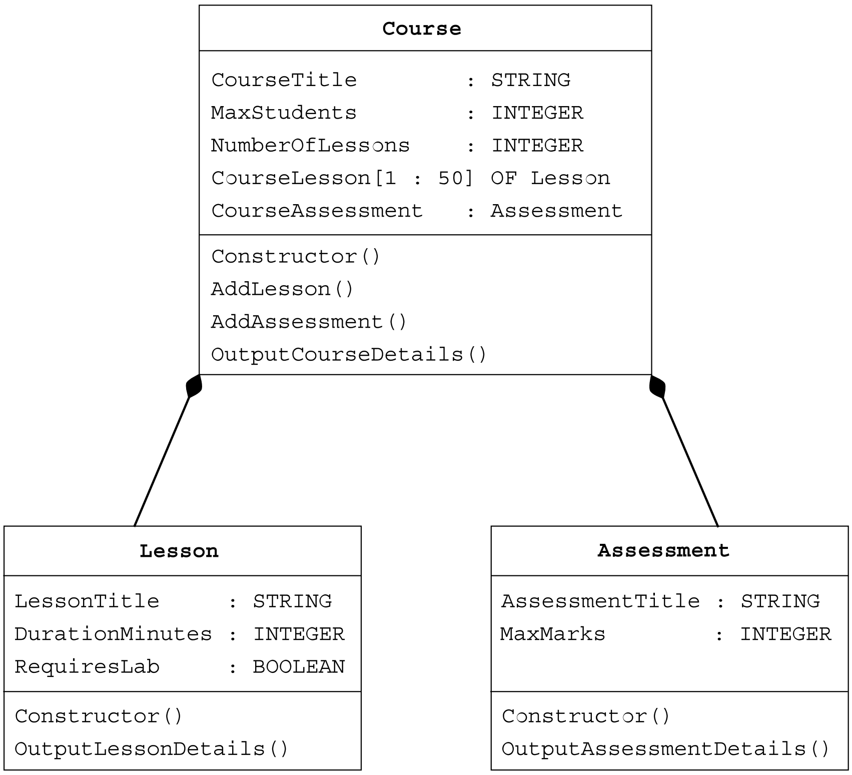


Figure 27.08 Containment class diagram

Assuming that all attributes for the `Lesson` and `Assessment` classes are set by values passed as parameters to the constructor, the code for declaring the `Lesson` and `Assessment` classes is straightforward.

The code below shows how the `Course` class is declared.

Python Course class declaration

```
class Course:
    def __init__(self, t, m): # sets up a new course
        self.__CourseTitle = t
        self.__MaxStudents = m
        self.__NumberOfLessons = 0
        self.__CourseLesson = []
        self.__CourseAssessment = Assessment

    def AddLesson(self, t, d, r):
        self.__NumberOfLessons = self.__NumberOfLessons + 1
        self.__CourseLesson.append(Lesson(t, d, r))

    def AddAssessment(self, t, m):
        self.__CourseAssessment = Assessment(t, m)

    def OutputCourseDetails(self):
        print(self.__CourseTitle, "Maximum number: ", self.__MaxStudents)
        for i in range(self.__NumberOfLessons):
            print(self.__CourseLesson[i].OutputLessonDetails())
```

VB.NET Course class declaration

```
Class Course
    Private CourseTitle As String
    Private MaxStudents As Integer
    Private NumberOfLessons As Integer = 0
    Private CourseLesson(50) As Lesson
    Private CourseAssessment As Assessment

    Public Sub Create(ByVal t As String, ByVal m As Integer)
        CourseTitle = t
        MaxStudents = m
    End Sub

    Sub AddLesson(ByVal t As String, ByVal d As Integer, ByVal r As Boolean)
        NumberOfLessons = NumberOfLessons + 1
        CourseLesson(NumberOfLessons) = New Lesson
        CourseLesson(NumberOfLessons).Create(t, d, r)
    End Sub
    Public Sub AddAssessment(ByVal t As String, ByVal m As Integer)
        CourseAssessment = New Assessment
        CourseAssessment.Create(t, m)
    End Sub

    Public Sub OutputCourseDetails()
        Console.WriteLine(CourseTitle & "Maximum number: " & MaxStudents)
        For i = 1 To NumberOfLessons
            CourseLesson(i).OutputLessonDetails()
        Next
    End Sub
End Class
```

Java Course class declaration

```
class Course
{
    private String courseTitle;
    private int maxStudents;
    private int numberOfLessons;
    private Lesson[] courseLesson;
    private Assessment courseAssessment;

    public Course(String t, int m)   // sets up a new course
    {
        courseTitle = t;
        maxStudents = m;
        numberOfLessons = 0;
        courseLesson = new Lesson[50];
        //courseAssessment = new Assessment();
    }
    public void addLesson(String t, int d, Boolean r)
    {
        numberOfLessons = numberOfLessons + 1;
        courseLesson[numberOfLessons] = new Lesson(t, d, r);
    }

    public void addAssessment(String t, int m)
    {
        courseAssessment = new Assessment(t, m);
    }
    public void outputCourseDetails()
    {
        System.out.println(courseTitle + " - Maximum number of students: " + maxStudents);
        for (int i = 1; i < numberOfLessons; i++)
        {
            courseLesson[i].outputLessonDetails();
        }
    }
}
```

Here are simple test programs to check it works.

Python test program

```
def Main():
    MyCourse = Course("Computing", 10) # sets up a new course

    MyCourse.AddAssessment("Programming", 100) # adds an assignment
    # add 3 lessons

    MyCourse.AddLesson("Problem Solving", 60, False)
    MyCourse.AddLesson("Programming", 120, True)
    MyCourse.AddLesson("Theory", 60, False)

    # check it all works
    MyCourse.OutputCourseDetails()
```

VB.NET test program

```
Dim MyCourse As New Course
MyCourse.Create("Computing", 10) ' sets up a new course

MyCourse.AddAssessment("Programming", 100) ' adds an assessment
' add 3 lessons
MyCourse.AddLesson("Problem Solving", 60, False)
MyCourse.AddLesson("Programming", 120, True)
MyCourse.AddLesson("Theory", 60, False)
```

```
'check it all works
MyCourse.OutputCourseDetails()
Console.ReadLine()
```

Java test program

```
Course myCourse = new Course("Computing", 10); // sets up a new course
myCourse.addAssessment("Programming", 100); // adds an assessment
// add 3 lessons
myCourse.addLesson("Problem Solving", 60, false);
myCourse.addLesson("Programming", 120, true);
myCourse.addLesson("Theory", 60, false);
// check it all works
myCourse.outputCourseDetails();
```

TASK 27.08

Write the code required for the `Lesson` and `Assessment` classes. Add the code for the `Course` class and test your program with the appropriate simple program from Worked Example 27.03.

Reflection Point:

Explain why a class can be regarded as a use-defined type.

Summary

- A class has attributes (declared as private) and methods (declared as public) that operate on the attributes. This is known as encapsulation.
- Python and VB.NET support properties: attributes that also include getters and setters.
- A class is a blueprint for creating objects.
- An object is an instance of a class.
- A constructor is a method that instantiates a new object.
- A class and its attributes and methods can be represented by a class diagram.
- Classes (subclasses) can inherit from another class (the base class or superclass). This relationship between a base class and its subclasses can be represented using an inheritance diagram.
- A subclass has all the attributes and methods of its base class. It also has additional attributes and/or methods.
- Polymorphism describes the different behaviour of a subclass method with the same name as the base class method.
- Containment is a relationship between two classes where one class has a component that is of the other class type. This can be represented using a containment diagram.

Exam-style Questions

1 A program is to be written using an object-oriented programming language. A bank account class is designed. Two subclasses have been identified:

- **Personal Account**: the account holder pays a monthly fee and may overdraw the account up to an agreed overdraft limit.
- **Savings Account**: the account holder must maintain a positive balance and gets paid interest on the balance at an agreed interest rate.

a Draw an inheritance diagram for these classes. [3]

The design for the `BankAccount` class consists of:

- attributes:
  - `AccountHolderName`
  - `IBAN`: International Bank Account Number
- methods:
  - `CreateNewAccount`
  - `SetAccountHolderName`
  - `GetAccountHolderName`
  - `GetIBAN`

b Write **program code** for the class definition of the superclass `BankAccount`. [5]

c **i** State the attributes and/or methods required for the subclass `PersonalAccount`. [4]

ii State the attributes and/or methods required for the subclass `SavingsAccount`. [4]

iii Identify the feature of object-oriented program design that combines the attributes and methods into a class. [1]

2 A bus company in a town has two types of season ticket for their regular customers: pay-as-you go and contract. All season ticket holders have their name and email address recorded.

A pay-as-you-go ticket holder pays a chosen amount into their account. Each time the ticket holder makes a journey on the bus, the price of the fare is deducted from the amount held in the account. They can top up the amount at any time.

A contract ticket holder pays a fixed fee per month. They can then make unlimited journeys on the bus.

The bus company wants a program to process the season ticket data. The program will be written using an object-oriented programming language.

a Complete the class diagram showing the appropriate attributes and methods. [7]

```
SeasonTicketHolder
--------------------------------------------
PRIVATE
TicketHolderName: STRING

……………………………………………………………

……………………………………………………………
--------------------------------------------
PUBLIC
Constructor ()


……………………………………………………………
```

....
....
....
....

Pay–As–You–Go–TicketHolder
....
....

ContractTicketHolder
....

`Constructor (Name: STRING,`
 `email : STRING, Fee : CURRENCY)`
 ....
 ....

b Attributes and methods can be declared as either public or private.

i Explain why the `SeasonTicketHolder` attributes are declared as private. [2]

ii Explain why the `SeasonTicketHolder` methods have been declared as public. [2]

c Write program code to create a new instance of `ContractTicketHolder` with:

- Identifier: `NewCustomer`
- name: A. Smith
- email address: xyz@abc.xx
- monthly fee: $10 [3]

3 A queue abstract data type (ADT) is to be implemented using object-oriented programming. Two classes have been identified: `Queue` and `Node`. The class diagrams are as follows:

QueueClass

`Queue : ARRAY[0 : 50] OF NodeClass`
 `Head  : INTEGER`
 `Tail  : INTEGER`

`Constructor()`
 `JoinQueue(NewItem : NodeClass)`
 `LeaveQueue() : STRING`

NodeClass

`Data    : STRING`
 `Pointer : INTEGER`

`Constructor()`
 `SetData(d : STRING)`
 `SetPointer(x : INTEGER)`
 `GetData()    : STRING`
 `GetPointer() : INTEGER`

a State the relationship between these two classes. [1]

b The `NodeClass` constructor is to:

- create a new node
- initialise the `Data` attribute to the empty string
- initialise the `Pointer` attribute to -1.

Write program code to define `NodeClass`, including the get and set methods. [10]

c The `QueueClass` constructor is to:

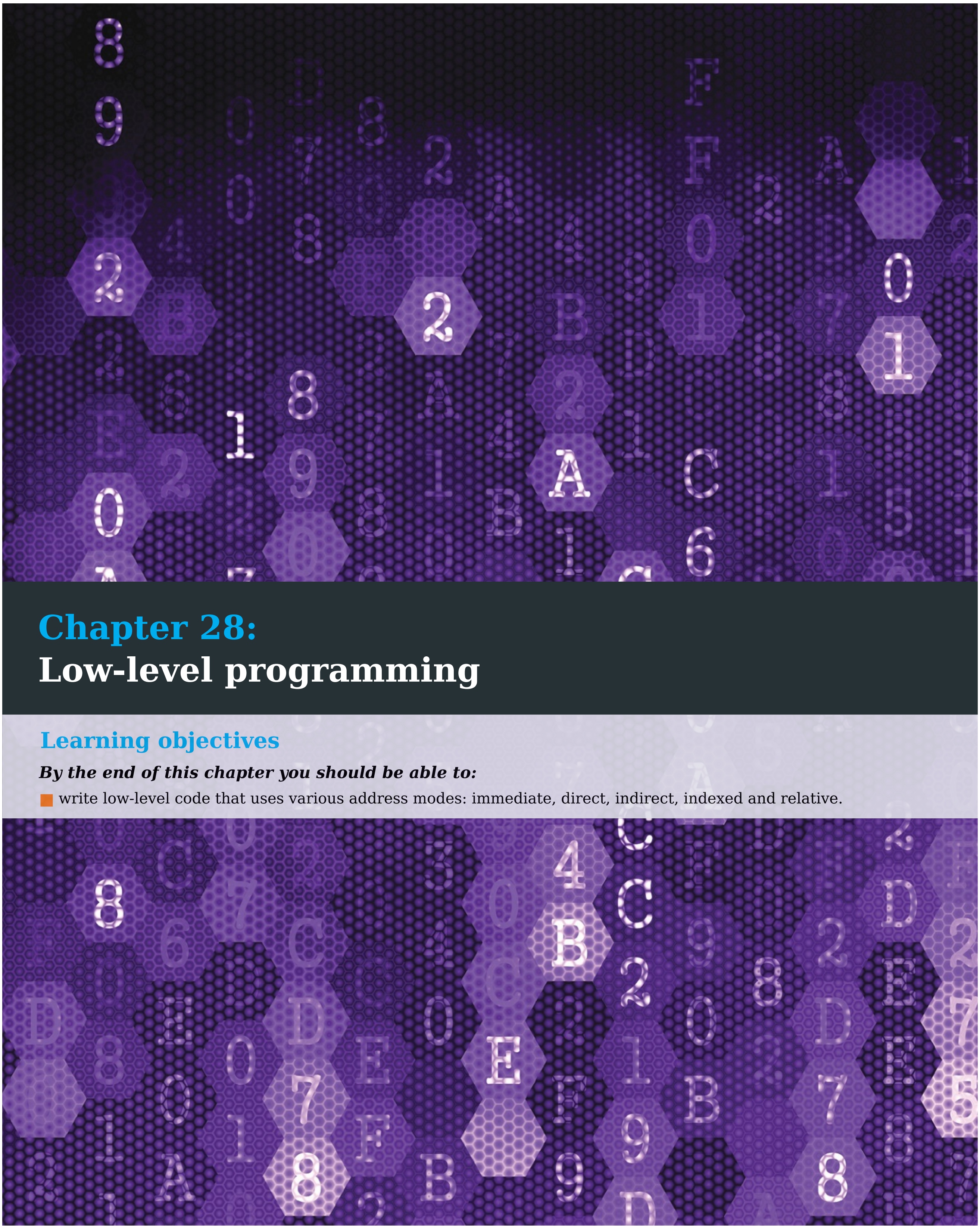
- create a new queue
- initialise the `Head` and `Tail` attributes to -1.

Write program code to define the constructor for `QueueClass`. [3]

d The `JoinQueue` method is to:

- create a new object, `Node`, of `NodeClass`
- assign the value passed as parameter to the `Data` attribute of `Node`
- assign `Node` to the end of `Queue`.

Write program code to define the `JoinQueue` method. [5]



Chapter 28:
Low-level programming

Learning objectives

By the end of this chapter you should be able to:

■ write low-level code that uses various address modes: immediate, direct, indirect, indexed and relative.

28.01 Processor instruction set

The instruction set we will use in this chapter is shown in Table 28.01.

Instruction			Explanation
Label	**Op code**	**Operand**	
Data movement instructions			
	LDM	#n	Immediate addressing. Load the denary number *n* to ACC
	LDD	<address>	Direct addressing. Load the contents of the location at the given address to ACC
	LDI	<address>	Indirect addressing. The address to be used is at the given address. Load the contents of this second address to ACC
	LDX	<address>	Indexed addressing. Form the address from <address> + the contents of the index register. Copy the contents of this calculated address to ACC
	LDR	#n	Immediate addressing. Load the denary number *n* to IX
	MOV	<register>	Move the contents of ACC to the given register (IX)
	STO	<address>	Store the contents of ACC at the given address
	STX	<address>	Indexed addressing. Form the address from <address> + the contents of the index register. Copy the contents from ACC to this calculated address*
	STI	<address>	Indirect addressing. The address to be used is at the given address. Store the contents of ACC at this second address*
*Note: The STX and STI instructions are not given as part of the instruction set in the syllabus for 9618			
Arithmetic operations			
	ADD	<address>	Add the contents of the given address to the ACC
	ADD	#n	Add the denary number *n* to the ACC
	SUB	<address>	Subtract the contents of the given address from the ACC
	SUB	#n	Subtract the denary number *n* from the ACC
	INC	<register>	Add 1 to the contents of the register (ACC or IX)
	DEC	<register>	Subtract 1 from the contents of the register (ACC or IX)
Comparison and jump instructions			
	JMP	<address>	Jump to the given address
	CMP	<address>	Compare the contents of ACC with the contents of <address>
	CMP	#n	Compare the contents of ACC with the denary number *n*

	CMI	<address>	Indirect addressing. The address to be used is at the given address. Compare the contents of ACC with the contents of this second address
	JPE	<address>	Following a compare instruction, jump to <address> if the compare was True
	JPN	<address>	Following a compare instruction, jump to <address> if the compare was False
Input/output instructions			
	IN		Key in a character and store its ASCII value in ACC
	OUT		Output to the screen the character whose ASCII value is stored in ACC
Bit manipulation instructions			
	AND	#n	Bitwise AND operation of the contents of ACC with the operand
	AND	<address>	Bitwise AND operation of the contents of ACC with the contents of <address>
	XOR	#n	Bitwise XOR operation of the contents of ACC with the operand
	XOR	<address>	Bitwise XOR operation of the contents of ACC with the contents of <address>
	OR	#n	Bitwise OR operation of the contents of ACC with the operand
	OR	<address>	Bitwise OR operation of the contents of ACC with the contents of <address>
	LSL	#n	Bits in ACC are shifted logically n places to the left. Zeros are introduced on the right-hand end
	LSR	#n	Bits in ACC are shifted logically n places to the right. Zeros are introduced on the left-hand end
Other			
	END		Return control to the operating system
<label>:	<opcode>	<operand>	Labels an instruction
<label>:	<data>		Gives a symbolic address <label> to the memory location with contents <data>

Table 28.01 Processor instruction set

In the assembly code in this chapter:

- ACC denotes the Accumulator
- IX denotes the Index Register
- # denotes immediate addressing
- B denotes a binary number, e.g. B01001010
- & denotes a hexadecimal number, e.g. &4A
- <address> can be an absolute address or a symbolic address.

An instruction set was introduced in Chapter 6.

28.02 Symbolic addresses

A label is a symbolic name for the memory location that it represents. You can treat it like a variable name. When writing low-level programs, we can give absolute addresses of memory locations. This is very restrictive, especially if we want to change the program by adding extra instructions. Writing low-level instructions using symbolic addresses (labels), allows us to think at a higher level. The assembler will allocate absolute addresses during the assembly process (see Chapter 6, Section 6.03).

28.03 Problem-solving and assembly-language programs

When writing a solution to a problem using low-level programming, we need to break down the problem into simple steps that can be programmed using the instruction set available.

One approach is to think in terms of the basic constructs we discussed for high-level languages. You can use the following examples as design patterns.

Assignment

Table 28.02 shows some examples of assembly language assignments that match the pseudocode.

Pseudocode examples	Assembly code examples	Explanation

`A ← 34`	`LDM #34`
	`STO A`
		To store a value in a memory location, the value must first be generated in the accumulator

`B ← B + 1`	`LDD B`
	`INC ACC`
	`STO B`
		To increment the value stored at a memory location: first load the value into the accumulator, increment the value and then store the contents of the accumulator back to the memory location

`B ← B + A`	`LDD B`
	`ADD A`
	`STO B`
		To calculate a value: load the first value from a memory location into the accumulator, then add the value stored at the second memory location to the accumulator and then store the contents of the accumulator to the required memory location

`A ← -A`	`LDD A`
	`XOR #&FF`
	`INC ACC`
	`STO A`
	Alternative method:
	`LDM #0`
	`SUB A`
	`STO A`
		Load the value (assuming eight bits),
		`XOR` with 11111111 to produce the one’s complement. Add 1 to get the two’s complement.
		Alternatively, set the ACC to zero and then subtract the value from the ACC and store it back in the original address

Table 28.02 Using assignment instructions

TASK 28.01

Write assembly code instructions for this sequence of pseudocode statements:

```
A ← 2
B ← 10
C ← A + B
D ← A - B
```

Selection

Table 28.03 shows some examples of assembly language selections that match the pseudocode.

Pseudocode examples	Assembly code examples	Explanation

`IF A = 0`
`  THEN`
`    B ← B + 1`
	`LDD A`
	`CMP #0`
	`JPN ENDIF`
		Load the contents of the memory location to be tested. Compare it with the required value (in this example 0). If the comparison result

ENDIF	THEN: LDD B
	INC ACC
	STO B
	ENDIF: ...
		is false (A does not equal 0), a jump over the THEN part is required; if the comparison result is true (A = 0) then the following instructions are executed. For ease of understanding, the labels THEN and ENDIF are used.

IF A = B
THEN
OUTPUT "Y"
ELSE
OUTPUT "N"
ENDIF
	LDD A
	SUB B
	CMP #0
	JPN ELSE
	THEN: LDM #89
	OUT
	JMP ENDIF
	ELSE: LDM #78
	OUT
	ENDIF: ...
		Load the contents of A.
		Subtract B.
		If the result is zero, A = B.
		If the comparison result is false (A does not equal B), jump to the ELSE part; if the comparison result is true
		(A = B) the instructions following the THEN label are executed.
		Note that a jump over the ELSE part is required.

Table 28.03 Using selection instructions

TASK 28.02

Write assembly code instructions for this sequence of pseudocode statements:

```
IF A <> 0
  THEN
    B ← A
  ELSE
    B ← B - 1
```

Repetition

Table 28.04 shows an example of repetition in assembly language that matches the pseudocode.

Pseudocode example	Assembly code example	Explanation
A = 0 REPEAT OUTPUT "*" A ← A + 1 UNTIL A = 5	LDM #0 STO A LOOP: LDM #42 OUT LDD A INC ACC STO A CMP #5 JPN LOOP	Store the initial value of the counter in A. Generate the ASCII code for the character "*" and output it. Load the counter value, increment the counter, save it, test for final value. If final value has not been reached, jump back to beginning of loop.

Table 28.04 Using repetition instructions

TASK 28.03

Write assembly code instructions for this sequence of pseudocode statements:

```
Number ← 1
Total ← 0
```

```
Max ← 5
REPEAT
    Total ← Total + Number
    Number ← Number + 1
UNTIL Number = Max
```

Input/output

Table 28.05 shows some examples of input and output in assembly language that match the pseudocode.

Pseudocode examples	Assembly code examples	Explanation

`INPUT A`	`IN`
	`STO A`
		Store a character input from the keyboard at memory location A.

`OUTPUT B`	`LDM #-1`
	`MOV IX`
	`LOOP: INC IX`
	`LDX B`
	`OUT`
	`CMP #13`
	`JPN LOOP`
		To output a string of characters stored in consecutive locations, starting at location labelled B, a loop and indexed addressing are used.
		The first time round the loop the index register is 0 and the character in memory location B will be loaded into the accumulator and output to the screen. Then a check is made for the end of the string (the carriage return character with ASCII code 13). If it is not the end of the string, jump back to the beginning of the loop.
		To output a number, the number must first be changed into its equivalent string and stored in consecutive memory locations. Then the above method can be used.

`INPUT A`	`LDM #-1`
	`MOV IX`
	`LOOP: INC IX`
	`IN`
	`STX A`
	`CMP #13`
	`JPN LOOP`
		Store a string of characters input from the keyboard into consecutive memory locations starting from A.
		Note: The STX instruction is not given as part of the instruction set in the syllabus for 9618 (from now on referred to as the 9618 instruction set). This means a string can not be saved with the 9618 instruction set.

Table 28.05 Using input and output instructions

TASK 28.04

Write assembly code instructions for this sequence of pseudocode statements:

```
Count ← 0
REPEAT
    Count ← Count + 1
    INPUT Character
UNTIL Character = "N"
```

TASK 28.05

Modify your assembly code instructions from Task 28.04 to implement this sequence of pseudocode statements:

```
Count ← 0
REPEAT
    Count ← Count + 1
    INPUT Character
UNTIL Character = "N"
OUTPUT Count
```

28.04 Absolute and relative addressing

An absolute address is the numeric address of a memory location. A program using absolute addresses cannot be loaded anywhere else in memory. Some assemblers produce relative addresses, so that the program can be loaded anywhere in memory.

Relative addresses are addresses relative to a base address, for example the first instruction of the program. When the program is loaded into memory the base address is stored in a base register `BR`. Instructions that refer to addresses then use the value in the base register, modified by the offset. For example, `STO [BR] + 10` will store the contents of the accumulator at the address calculated from (contents of the base register) + 10.

Table 28.06 shows an example of instructions using symbolic, relative and absolute addressing.

Symbolic addressing	Offset from base (START)	Relative addressing (base address stored in base register)	Absolute addressing
START: LDM #0	0	LDM #0	100 LDM #0
STO A	1	STO [BR] + 10	201 STO 210
LOOP: LDM #42	2	LDM #42	202 LDM #42
OUT	3	OUT	203 OUT
LDD A	4	LDD [BR] + 10	204 LDD 210
INC ACC	5	INC ACC	205 INC ACC
STO A	6	STO [BR] + 10	206 STO 210
CMP #5	7	CMP #5	207 CMP #5
JPN LOOP	8	JPN [BR] + 2	208 JPN 202
END	9	END	209 END
A: 0	10	0	210 0

Table 28.06 Symbolic, relative and absolute addressing

It is very important that, at the end of the program, control is passed back to the operating system. Otherwise the binary pattern held in the next memory location will be interpreted as an instruction. If the content of that memory location does not correspond to a valid instruction, the processor will crash. The instruction `END` signals the end of the program instructions.

Note: the 9618 instruction set does not include relative (base register) addressing.

28.05 Indirect addressing

Indirect addressing is useful if the memory address to be used in an instruction is changed during the execution of the program.

One example is when programming subroutines to which parameters are passed by reference (this is beyond the scope of this book).

Another use of indirect addressing is for a pointer variable.

WORKED EXAMPLE 28.01

Writing a program for a simple queue

At the top level, we can write the problem using structured English:

```
Add a character to the queue:
1  Store the contents of the accumulator in the memory location
pointed to by the tail pointer.
2  Increment the tail pointer.
Remove a character from the queue:
1  Load contents of the memory location at the head of the queue.
2  Increment the head pointer.
```

Table 28.07 shows an example of instructions that implement the above queue-processing algorithms.

Instruction			Explanation
Label	Op code	Operand	
JOINQ:	STI	TAILPTR	Store contents of ACC in the memory location pointed to by the tail pointer
	LDD	TAILPTR	Increment the tail pointer
	INC	ACC	
	STO	TAILPTR	
	JMP	ENDQ	
LEAVEQ:	LDI	HEADPTR	Load contents of memory location at the head of the queue
	OUT		Output the character
	LDD	HEADPTR	Increment the head pointer
	INC	ACC	
	STO	HEADPTR	
	JMP	ENDQ	
ENDQ:			
HEADPTR:	QSTART		Pointer to start of queue
TAILPTR:	QSTART		Pointer to next free location in queue
QSTART:	""		Start of memory reserved for queue, currently empty

Table 28.07 Queue processing

Note that the value shown in Table 28.07 at the memory locations labelled HEADPTR and TAILPTR is the address of the start of the memory locations reserved for the queue. As values are added to the queue, the `TAILPTR` value will increase to point to the memory location at the end of the queue data. When a value is taken from the queue, the `HEADPTR` value will increase to point to the memory location at the head of the queue data.

Note: the 9618 instruction set does not include the STI opcode and a character can not be added to the queue using the 9618 instruction set.

TASK 28.06

Write instructions to reverse a word entered at the keyboard. This requires access to an area of memory treated as a stack.

Reflection Point:

Here are the addressing modes you should be able to use when writing low-level code:

Immediate	
Direct	
Indirect	
Indexed	
Relative	

Tick the box next to each addressing mode you can use easily and place a cross in the boxes next to the addressing modes you have difficulty with.

Summary

- A problem to be solved must be broken down into simple steps that can be programmed using the processor's given instruction set.
- A value must be copied into the accumulator before it can be processed.
- Processing includes:
  - arithmetic: adding, incrementing, decrementing
  - comparison: equal or not equal
  - bitwise operations: AND, OR, XOR, shifting
  - output to screen.
- To set a value in the accumulator it can be:
  - input from the keyboard
  - created using immediate addressing
  - loaded from a memory location using direct, relative, indirect or indexed addressing.
- An address can be absolute (a number) or symbolic (a label).

Exam-style Questions

1 The instruction set of a processor with one general-purpose register, the accumulator, includes the following instructions.

Instruction			**Explanation**
Label	**Op code**	**Operand**	
	`LDD`	`<address>`	Direct addressing. Load the contents of the given address to `ACC`
	`STO`	`<address>`	Store the contents of `ACC` at the given address
	`ADD`	`<address>`	Add the contents of the given address to the `ACC`
	`IN`		Key in a character and store its ASCII value in `ACC`
	`AND`	`<address>`	Bitwise `AND` operation of the contents of `ACC` with the contents of `<address>`
	`LSL`	`#n`	Bits in `ACC` are shifted logically *n* places to the left. Zeros are introduced on the right-hand end
	`END`		Return control to the operating system
`<label>:`	`<data>`		Gives a symbolic address `<label>` to the memory location with contents `<data>`

Key to the above table:

- `ACC` denotes the Accumulator.
- `#` denotes immediate addressing.
- `&` denotes a hexadecimal number, e.g. `&4A`.
- `<address>` can be an absolute address or a symbolic address.

a Explain the operation of the `AND` instruction. [1]

b The ASCII code for ‘0’ is the binary value 00110000. The ASCII code for ‘1’ is the binary value 00110001.

Write an `AND` instruction to convert any numeric digit stored in `ACC` in the form of an ASCII code to its eight-bit binary integer equivalent. [1]

c Write the assembly code instructions to convert a two-digit number keyed in at the keyboard to its BCD representation. Store the result in the memory location labelled Result. [7]

Instruction			**Explanation**
Label	**Op code**	**Operand**	
			Input first digit
			Convert from ASCII to its digit value
			Move to upper nibble
			Store in location Result
			Input second digit
			Convert from ASCII to its digit value
			Combine the two values
			Store result
			End of program
`Mask:`			Mask to convert from ASCII to digit equivalent
`Result:`	`&00`		Memory location for result

Result:	&00		Memory location for result

2 A given processor has one general-purpose register, the accumulator ACC, and one index register, IX. Part of the instruction set for this processor is as follows.

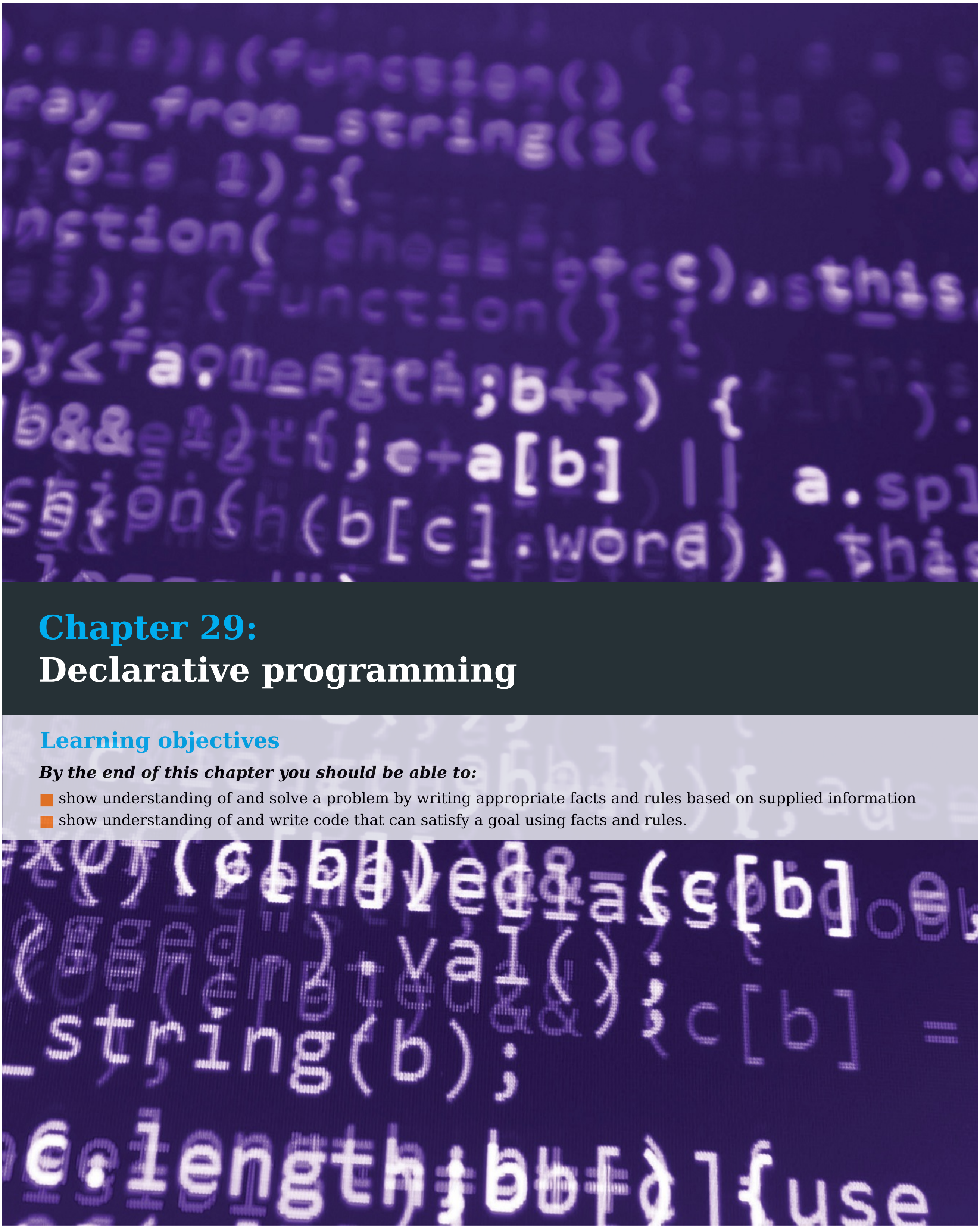
Instruction			**Explanation**
Label	**Op code**	**Operand**	
	`LDM`	`#n`	Immediate addressing. Load the number *n* to ACC
	`LDD`	`<address>`	Direct addressing. Load the contents of the given address to ACC
	`LDX`	`<address>`	Indexed addressing. Form the address from `<address>` + the contents of the Index Register. Copy the contents of this calculated address to ACC
	`LDR`	`#n`	Immediate addressing. Load the denary number *n* to IX
	`STO`	`<address>`	Store the contents of ACC at the given address
	`STX`	`<address>`	Indexed addressing. Form the address from `<address>` + the contents of the index register. Copy the contents from ACC to this calculated address
	`ADD`	`<address>`	Add the contents of the given address to the ACC
	`INC`	`<register>`	Add 1 to the contents of the register (ACC or IX)
	`JMP`	`<address>`	Jump to the given address
	`CMP`	`<address>`	Compare the contents of ACC with the contents of `<address>`
	`CMP`	`#n`	Compare the contents of ACC with the denary number *n*
	`JPE`	`<address>`	Following a compare instruction, jump to `<address>` if the compare was True
	`JPN`	`<address>`	Following a compare instruction, jump to `<address>` if the compare was False
	`IN`		Key in a character and store its ASCII value in ACC
	`END`		Return control to the operating system
`<label>:`	`<op code>`	`<operand>`	Labels an instruction
`<label>:`	`<data>`		Gives a symbolic address `<label>` to the memory location with contents `<data>`

Key to the above table:

- `#` denotes immediate addressing.
- `<address>` can be an absolute address or a symbolic address.

Write an assembly language program that outputs a sequence of characters stored in successive locations, starting at the location labelled: STRING. Output ends when the character in ACC is ‘!’ (ASCII code 33). [8]

Instruction			**Explanation**
Label	**Op code**	**Operand**	
			Set index register to zero
			Load ACC with character stored at STRING (modified by index register)
			Increment index register
			Output character
			Is this character the ! key?
			No – jump to beginning of loop
			End of program
STRING:	72		String stored from here onwards
	69		
	76		
	80		
	33		



Chapter 29:
Declarative programming

Learning objectives

By the end of this chapter you should be able to:

- show understanding of and solve a problem by writing appropriate facts and rules based on supplied information
- show understanding of and write code that can satisfy a goal using facts and rules.

29.01 Declarative programming languages

Declarative languages include database query languages (such as SQL, see Chapter 11, Section 11.07), regular expressions, logic programming and functional programming.

Prolog is a logic programming language widely used for artificial intelligence and expert systems.

The Prolog programs in this chapter have been prepared using the SWI-Prolog environment shown in Figure 29.01 (see SWI-Prolog for a free download).

Figure 29.01 SWI-Prolog environment

29.02 Prolog basics

There are three basic constructs in Prolog: facts, rules and queries.

The program logic is expressed using clauses (facts and rules). Problems are solved by running a query (goal).

A collection of clauses is called a ‘knowledge base’. Writing a Prolog program means writing a knowledge base as a collection of clauses. We use the program by writing queries.

A clause is of the form:

```
Head :- Body.
```

Note that a clause always terminates with a full stop (.)

Prolog has a single data type, called a ‘term’. A term can be:

- an atom, a general-purpose name with no inherent meaning that always starts with a lower case letter
- a number, integer or float (real)
- a variable, denoted by an identifier that starts with a capital letter or an underscore (_)
- a compound term, a predicate, consisting of an atom and a number of arguments in parentheses ().

The arguments themselves can be compound terms. A predicate has an arity (that is, the number of arguments in parentheses).

Prolog is case sensitive.

29.03 Facts in Prolog

A clause without a body is a fact, for example:

```
01  capitalCity(paris).
02  capitalCity(berlin).
03  capitalCity(cairo).
```

The meaning of clause `01` is: Paris is a capital city.
`capitalCity(paris)` is a compound term. Both `capitalCity` and `paris` are atoms.
`capitalCity` is called a predicate and `paris` is the argument.

`capitalCity` has arity 1, as it has just one argument. This can be written as `capitalCity/1`, the `/1` showing that it takes one argument.

TASK 29.01

Launch the editor (File, New ...) from the SWI-Prolog environment. Enter the three clauses, as shown in Figure 29.02. Then save the file (File, Save buffer) as Ex1.

Figure 29.02 Example facts in SWI-Prolog editor

Clauses 01 to 03 are a knowledge base. We can run a query on this knowledge base.

To ask the question whether Paris is a capital city, we write:

```
capitalCity(paris).
```

Prolog answers `true`.

This means: yes, Paris is a capital city.

To ask the question whether London is a capital city, we write:

```
capitalCity(london).
```

Prolog answers `false`.

This means: no, London is not a capital city.

This is because the fact that London is a capital city has not been included in our knowledge base.

TASK 29.02

Run your own queries. You first need to consult the knowledge base (File, Consult ...) from within the Prolog environment. Note that SWI-Prolog uses the prompt ?- (see Figure 29.03).

Figure 29.03 Example queries in SWI-Prolog

If your query does not get a response, check that you have:

- consulted your knowledge base (green text in Figure 29.03)
- used lower-case letters appropriately
- used a full stop at the end of your query.

29.04 Prolog variables

Let’s add some more facts to our knowledge base. Comments in Prolog are enclosed in /* and */.

```
cityInCountry(paris, france). /* Paris is a city in France */
cityInCountry(berlin, germany).
cityInCountry(cairo, egypt).
cityInCountry(munich, germany).
```

To find out which country Berlin is in, we can run the query (see Figure 29.04):

```
cityInCountry(berlin, Country).
```

Note that `Country` is a variable (it starts with a capital letter).

To find out which cities are in Germany, we can run the query (see Figure 29.04):

```
cityInCountry(City, germany).
```

Figure 29.04 Instantiations of a variable

Note how Prolog responds when running a query that includes a variable. When there is more than one answer, you need to type a semicolon after the first answer and Prolog will give the second answer. The semicolon has the meaning OR. First `City` is instantiated to `berlin` and then `City` is instantiated to `munich`.

WORKED EXAMPLE 29.01

Using a knowledge base

Consider the following knowledge base:

```
vegetable(aubergine). /* aubergine is a vegetable */
vegetable(potato).
vegetable(tomato).
meat(chicken). /* chicken is a type of meat */
meat(beef).
meat(lamb).
ingredient(tagine, aubergine, 250). /* tagine contains 250g aubergine */
ingredient(tagine, tomato, 100).
ingredient(stew, beef, 400).
ingredient(stew, potato, 600).
```

We can check the ingredients of a tagine by asking:

```
ingredient(tagine, Ingredient, Amount).
```

Look at the response Prolog gives in Figure 29.05.

Figure 29.05 Instantiation of variables

29.05 The anonymous variable

Consider the knowledge base from Worked Example 29.01. If we are not interested in the amount of each ingredient, we can use the anonymous variable (represented by the underscore character). The query then becomes

```
ingredient(tagine, Ingredient, _).
```

29.06 Rules in Prolog

Remember a clause is of the form `Head :- Body`.

A rule's body consists of calls to predicates, which are called the rule's goals. A predicate is either true or false, based on its terms. If the body of the rule is true, then the head of the rule is true too.

WORKED EXAMPLE 29.02

Using rules in a knowledge base

Consider the following knowledge base:

```
parent(fred, jack). /* Fred is the father of Jack */
parent(fred, alia).
parent(fred, paul).
parent(dave, fred).
```

We know that G is a grandparent of S, if G is a parent of P and P is a parent of S.

We could write this as a rule:

`grandparent(G, S) IF parent(G, P) AND parent(P, S).`

However, in Prolog the `IF` is replaced by `:-` and the `AND` is replaced with a comma:

```
grandparent(G, S) :- parent(G, P), parent(P, S).
```

A person has a sibling (brother or sister) if they have the same parent. We can write this as the Prolog rule:

```
sibling(A, B) :-
   parent(P, A),
   parent(P, B).
```

If we run the query

```
sibling(jack, X).
```

we get the answers we expect, but we also get the answer that Jack is his own sibling. To avoid this, we modify the query to ensure that A is not equal to B:

```
sibling(A, B) :-
   parent(P, A),
   parent(P, B),
   not(A=B).
```

Question 29.01

What answer do you expect to get from Prolog to the following query?

`sibling(dave, X).`

TASK 29.03

Write a knowledge base for your own family. You can include more predicates, for example:

Predicate	Meaning
`male(fred).`	`Fred is male`
`female(alia).`	`Alia is female`

Write a rule for father.

Test your program.

WORKED EXAMPLE 29.03

Adding a rule to a knowledge base

Using the knowledge base from Worked Example 29.01, we want to know which dishes contain meat. We are not interested how much meat, so we don't need to know the value of the third argument of the predicate `ingredient/3`. We can write the rule:

```
containsMeat(X) :-
   ingredient(X, Meat, _),
   meat(Meat).
```

The query `containsMeat(X).` returns `X = stew`.

29.07 Instantiation and backtracking

Prolog responds to a query with an answer, such as the one in Worked Example 29.03:

```
X = stew.
```

The = sign is not an assignment as in imperative programs. The = sign shows instantiation.

How does Prolog use the knowledge base to arrive at the answers? One way to see exactly what Prolog is doing is to use the graphical debugger.

WORKED EXAMPLE 29.04

Use the knowledge base from Worked Example 29.03. After consulting the knowledge base, start the debugger (Debug, Graphical debugger) from the Prolog environment. Then type: `trace.` and then the goal as shown in Figure 29.06.

Figure 29.06 Switching on the trace facility

The graphical debugger window opens as shown in Figure 29.07.

Figure 29.07 Graphical debugger

Using the space bar you can step through the program. When Prolog gives an answer in the Prolog Environment window, remember to input a semicolon, so that Prolog will go and check for another possible answer.

If you don’t use the graphical debugger but type `trace.` you can see the trace in the SWI-Prolog window, as shown in Figure 29.08.

Figure 29.08 SWI-Prolog trace of goal `containsMeat(X)`

The following terminology is used when discussing a trace:

- `Call` is the initial entry to a predicate
- `creep` indicates that Prolog is moving to the next predicate
- `Exit` is a successful return
- `Redo` indicates that the predicate is backed into for another answer
- `Fail` indicates that Prolog can find no more solutions.

29.08 Recursion

Recursion for imperative languages is covered in Chapter 24. Recursion for declarative languages is where a rule is defined by itself, or more precisely, a rule uses itself as a sub-goal.

Let us expand the Family knowledge base from Worked Example 29.02.

We want a rule that defines whether person A is an ancestor of person B. If A is a parent of B, then A is an ancestor of B. Similarly, if person A is the parent of P, who is the parent of B, then A is an ancestor of B. This is true for the parent of a parent of a parent of B. In general, if A is a parent of X and X is an ancestor of B, then A is an ancestor of B. We can write this information as the rules shown in Figure 29.09.

```
ancestor(A, B) :- parent(A, B).
ancestor(A, B) :- parent(A, X), ancestor(X, B).
```

The base case

The general case

Figure 29.09 Recursive rules

Note that recursion in declarative programming must follow the equivalent rules that imperative programming must follow. A recursive rule must:

- have a base case
- have a general case
- reach the base case after a finite number of calls to itself.

TASK 29.04

Add the ancestor rules to the Family knowledge base and check that the following query gives the correct results:

```
ancestor(A, jack).
```

WORKED EXAMPLE 29.05

Creating the factorial function in Prolog

In Chapter 24, Worked Example 24.01, we programmed the factorial function using recursion with imperative programming. We can also program this function using recursion in Prolog.

```
factorial(0, 1).                        /* base case: 0! = 1    */
factorial(N, Result) :-                 /* Result = N!          */
   M is N - 1,                          /* assign N-1 to M      */
   factorial(M, PartResult),            /* PartResult  = (N-1)! */
   Result is PartResult * N.            /* Result = N * (N-1)!  */
```

TASK 29.05

Enter the code from Worked Example 29.05 into the Prolog editor. Save it and consult it. Then pose the following query:

```
factorial(5, Answer).
```

Do you get the correct answer?

29.09 Lists

A list is an ordered collection of terms. It is denoted by square brackets with the terms separated by commas or in the case of the empty list, []. For example [1, 2, 3] or [red, green, blue]. An element can be any type of Prolog object. Different types can be mixed within one list. Lists are used in Prolog where arrays may be used in procedural languages.

Any non-empty list can be thought of as consisting of two parts: the head and the tail. The head is the first item in the list; the tail is the list that remains when we take the first element away. This means that the tail of a list is always a list.

Lists are manipulated by separating the head from the tail. The separator used is a vertical line (a bar):|

If Prolog tries to match `[H|T]` to `[car, lorry, boat, ship]`, it will instantiate `H` to `car` and `T` to `[lorry, boat, ship]`.

The clause definition `showHeadAndTail([H|T], H, T).` can be used to pose the query:

```
showHeadAndTail([fred, jack, emma], Head, Tail).
```

Prolog responds with:

```
Head = fred,
Tail = [jack, emma].
```

The clause definition `myList([1,2,3]).` can be used to pose the query:

```
myList([H|T]).
```

Prolog responds with:

```
H = 1,
T = [2, 3].
```

The clause definition `emptyList(A) :- A = [].` can be used to pose the query:

```
emptyList([1]).
```

Prolog responds with:

```
false.
```

List-processing predicate: `append`

The built-in predicate `append(A, B, C)` joins list A and list B and produces list C.

```
append([a, b], [c, d], MyList).
```

produces `MyList = [a, b, c, d].`

```
append(FirstList, [c, d], [a, b, c, d]).
```

produces `FirstList = [a, b].`

List-processing predicate: `member`

The built-in predicate `member(A, B)` returns true if item A is in list B.

```
member(c, [a, b, c, d, e]).
```

produces:

```
true.
```

And

```
member(X,[a, b, c, d]).
```

produces:

```
X = a ;
X = b ;
X = c ;
```

```
X = d.
```

List-processing predicates: write and read

The built-in predicate `write(A)` outputs A to the screen.

`write('message: ').` outputs `message: .`

`write(X).` outputs the value currently instantiated with the variable X.

The built-in predicate `read(A)` reads a value from the keyboard into variable A.

`read(Name).` waits for an atom to be input from the keyboard and instantiates the variable `Name` with that value.

Note that the input must start with a lower case letter and not have spaces or be enclosed in quotes.

`nl` moves the output to a new line.

We can write user-friendly programs using the `read` and `write` predicates.

WORKED EXAMPLE 29.06

Using the read and write predicates

Note how the interface with the user in the code below is written as a rule with the separate steps separated by commas (representing AND).

`assert/1` adds the clause given as the argument to the knowledge base.

`retractall/1` takes the given clause out of the knowledge base, so the next time the program is run, the new facts will be added and used in the goal.

```
/* Weather knowledge base */
weather(good):-
   temp(high),
   humidity(dry),
   sky(sunny).
weather(bad):-
   (humidity(wet);
   temp(low);
   sky(cloudy)).
/* interface */
go:-
   write('Is the temperature high or low? '),
   read(Temp), nl,
   write('Is the sky sunny or cloudy? '),
   read(Sky), nl,
   write('Is the humidity dry or wet? '),
   read(Humidity), nl,
   assert(temp(Temp)),
   assert(sky(Sky)),
   assert(humidity(Humidity)),
   weather(Weather),
   write('The weather is '), write(Weather),
   retractall(temp(_) ),
   retractall(sky(_) ),
   retractall(humidity(_) ).
```

To run the program, type `go`.

TASK 29.06

Test the recursively defined rule writelist/1 to output the elements of a list.

```
writeList([]).
writeList([H|T]):-write(H), nl, writeList(T).
```

Reflection Point:

Can you explain the difference between a fact and a rule?

Summary

- Imperative programs reflect the steps of how to solve a problem.
- Declarative programs reflect what the problem is.
- A knowledge base consists of two types of clause: facts and rules.
- Clauses are sometimes referred to as predicates.
- The arity of a predicate shows how many arguments it takes.
- To solve a problem, the user of the knowledge base poses a query.
- A recursive rule is defined in terms of itself.
- In logic programming, a list is manipulated by separating the head from the tail ([H T]).

Exam-style Questions

1 A logic programming language is used to represent, as a set of facts and rules, details of cities of the world. The set of facts and rules are shown below in clauses labelled 1 to 17.

```
01  capital(vienna).
02  capital(london).
03  capital(santiago).
04  capital(caracas).
05  capital(tokyo).
06  cityIn(vienna, austria).
07  cityIn(santiago, chile).
08  cityIn(salzburg, austria).
09  cityIn(maracaibo, venezuela).
10  continent(austria, europe).
11  continent(chile, southAmerica).
12  continent(uk, europe).
13  continent(argentina, southAmerica).
14  iVisited(vienna).
15  iVisited(tokyo).
16  capitalOf(City, Country)
      IF capital(City) AND cityIn(City, Country).
17  europeanCity(City)
       IF cityIn(City, Country) AND continent(Country, europe).
```

These clauses have the following meanings:

Clause	Meaning
01	Vienna is a capital.
06	Vienna is in Austria.
10	Austria is in the continent of Europe.
14	I visited the city of Vienna.
16	`City` is the capital of `Country` if `City` is a capital and it is in `Country`.
17	`City` is a city in Europe if `City` is in `Country` and `Country` is in Europe.

a Write down the extra clauses needed to express the following facts:

i London is in the UK. [1]

ii I visited the city of Strasbourg. [1]

b The clause `cityIn(City, austria)` would return the result: `vienna, salzburg`.

Write down the result returned by the clause:

`continent(Country, southAmerica).` [2]

c Complete the rule to list countries that I have visited

`countriesIVisited(Country) IF ...` [3]

2 In a particular country, to become a qualified driver you must:

- have a licence: there is a minimum age at which a person can be issued with a licence and it is different for cars and trucks
- pass a theory test: it is the same test for cars and trucks
- pass a driving test for a specific vehicle (car or truck).

A declarative programming language is to be used to represent the knowledge base shown below:

```
01  minimumAge(car, 18).
02  minimumAge(truck, 21).
03  age(fred, 19).
04  age(jack, 22).
05  age(mike, 17).
06  age(jhon, 20).
```

```
07  age(emma, 22).
08  age(sheena, 19).
09  hasLicence(fred).
10  hasLicence(jack).
11  hasLicence(mike).
12  hasLicence(jhon).
13  hasLicence(emma).
14  hasLicence(sheena).
15  allowedToDrive(X, V)
       IF hasLicence(X) AND minimumAge(V, L)
          AND age(X, A)
          AND A >= L.
16  passedTheoryTest(jack).
17  passedTheoryTest(emma).
18  passedTheoryTest(jhon).
19  passedTheoryTest(fred).
20  passedDrivingTest(jhon, car).
21  passedDrivingTest(fred, car).
22  passedDrivingTest(jack, car).
23  passedDrivingTest(jack, truck).
24  passedDrivingTest(sheena, car).
25  qualifiedDriver(X, V)
       IF allowedToDrive(X, V)
          AND passedTheoryTest(X)
          AND passedDrivingTest(X, V).
```

These clauses have the following meanings:

Clause	**Meaning**
01	The minimum age for a car licence is 18.
03	Fred is aged 19.
09	Fred has a licence.
15	Person `X` is able to drive vehicle `V` if person `X` has a licence and the age `A` of person `X` is greater than or equal to the minimum age `L` to drive vehicle `V`.

a **i** Give **one** example of a fact in this knowledge base. [1]

ii Give **one** example of a rule in this knowledge base. [1]

b State the output produced from these clauses:

i `passedDrivingTest(Who, truck).` [1]

ii `allowedToDrive(mike, car).` [1]

iii `NOT(hasLicence(sheena)).` [1]

c Write a clause to output:

i all qualified car drivers [2]

ii all drivers who have passed the theory test but not a driving test. [3]

d To produce the output from a clause, the inference engine uses a process called backtracking.

Consider the clause:

`AllowedToDrive(mike, car).`

Identify the order in which clauses are used to produce the output. For each clause, describe the result that it returns. [5]

Acknowledgements

The authors and publishers acknowledge the following sources of copyright material and are grateful for the permissions granted. While every effort has been made, it has not always been possible to identify the sources of all the material used, or to trace all copyright holders. If any omissions are brought to our notice, we will be happy to include the appropriate acknowledgements on reprinting.

Cover Photo: Westend61/Getty Images

Galushko Sergey/Shutterstock; Solomonkein/Shutterstock; Pingingz/Shutterstock; Photo by Frank Wojechowski, used with permission of Michael McAlpine and Princeton University; Nikita Veremcuks/EyeEm/Getty Images; ilbusca/Getty Images; digitalreflections/Shutterstock;

Chapter Openers

polygraphus/Shutterstock; dgbomb/Shutterstock; Pingingz/Shutterstock;

agsandrew/Shutterstock; Raimundas/Shutterstock; Best Backgrounds/Shutterstock; Toria/Shutterstock; deepadesigns/Shutterstock;

Gino Santa Maria/Shutterstock; kubais/Shutterstock; bluebay/Shutterstock; Best Backgrounds/Shutterstock; Kheng Guan Toh/Shutterstock;

Best Backgrounds/Shutterstock; foxaon1987/Shutterstock; Kheng Guan Toh/Shutterstock; bluebay/Shutterstock; James Steidl/Shutterstock; Lukas Rs/Shutterstock; kentoh/Shutterstock; jijomathaidesigners/Shutterstock;

kentoh/Shutterstock; mistery/Shutterstock; kentoh/Shutterstock;

Titima Ongkantong/Shutterstock; Best Backgrounds/Shutterstock;

Mike McDonald/Shutterstock; Tashatuvango/Shutterstock;

Best Backgrounds/Shutterstock

CAMBRIDGE
UNIVERSITY PRESS

University Printing House, Cambridge CB2 8BS, United Kingdom
One Liberty Plaza, 20th Floor, New York, NY 10006, USA
477 Williamstown Road, Port Melbourne, VIC 3207, Australia
314–321, 3rd Floor, Plot 3, Splendor Forum, Jasola District Centre, New Delhi – 110025, India
79 Anson Road, #06 -04/06, Singapore 079906

Cambridge University Press is part of the University of Cambridge.

It furthers the University's mission by disseminating knowledge in the pursuit of education, learning and research at the highest international levels of excellence.

www.cambridge.org
Information on this title: www.cambridge.org/9781108733755

First published 2015
Second edition 2019

20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1

Printed in the United Kingdom by Latimer Trend

A catalogue record for this publication is available from the British Library

ISBN 978-1-108-73375-5 Paperback
ISBN 978-1-108-56832-6 Paperback with Cambridge Elevate edition (2 years)
ISBN 978-1-108-70041-2 Cambridge Elevate edition (2 years)
ISBN 978-1-108-70039-9 Digital edition

Cambridge University Press has no responsibility for the persistence or accuracy of URLs for external or third-party internet websites referred to in this publication, and does not guarantee that any content on such websites is, or will remain, accurate or appropriate. Information regarding prices, travel timetables, and other factual information given in this work is correct at the time of first printing but Cambridge University Press does not guarantee the accuracy of such information thereafter.

All exam-style questions and sample answers in this title were written by the authors. In examinations, the way marks are awarded may be different.

Past exam paper questions throughout reproduced by permission of Cambridge Assessment International Education.