1 Principles of Computer Science

Getting to know your unit

Assessment
You will be assessed by a written examination.

The key to computer science is solving problems. In order to solve problems effectively it is necessary to understand them fully, pulling them apart until the required data types, intricate algorithms and programming language syntax start to coalesce into an efficient program that is fit for purpose.

The aim of this unit is to encourage you to transform yourself from a computer program user to a computer program developer, capable of designing and programming smart solutions to new problems by drawing across your full study programme for the skills and knowledge required.

You will learn about computational thinking skills, the common components of different types of programming languages and the tools used to design and develop high-quality applications. In doing so, you will travel through the complete software development life cycle, honing your analytical and problem-solving skills so that you can gain employment as a junior software developer or progress to higher education and specialise further as a computing professional.

How you will be assessed

This unit is assessed through a written examination set and marked by Pearson.

The examination is two hours in length. During the supervised assessment period, learners will be assessed on their ability to apply their computational thinking skills to solve problems.

Grade descriptors

To achieve a grade, you will be expected to demonstrate these attributes across the essential content of the unit. The principle of best fit will apply when awarding grades. The maximum number of marks for this unit is 90.

To pass this unit:

- You will be able to use problem-solving skills to develop a solution to given problems in context.
- You will be able to use standard programming constructs to demonstrate an understanding of how data is handled in a computer program.
- You will be able to construct, propose, develop and explain solutions to a problem and demonstrate an understanding of data validation and error checking.

To gain a level 3 distinction:

- You will demonstrate that you can analyse and interpret problems and develop a detailed and complex solution in response.
- You will demonstrate an in-depth understanding of programming constructs and a thorough understanding of how data is handled in a computer program.

Assessment outcomes

AO1 Demonstrate knowledge and understanding of computing facts, terms, standards, concepts and processes

Command words: complete, draw, give, identify, name, state

Marks: range from 1 to 5 marks

AO2 Apply knowledge and understanding to communicate understanding of computing facts, terms, standards, concepts and processes

Command words: calculate, complete, demonstrate, describe, draw, explain, produce

Marks: range from 1 to 5 marks

AO3 Select and use computing technologies and procedures to explore outcomes and find solutions to problems in context

Command words: calculate, demonstrate, develop, explain, produce

Marks: range from 1 to 6 marks

AO4 Analyse data and information related to computer science in order to predict outcomes and present solutions

Command words: analyse, demonstrate, discuss, produce, write

Marks: range from 6 to 12 marks

AO5 Evaluate technologies, procedures, outcomes and solutions to make reasoned judgements and make decisions

Command words: evaluate, produce, write

Marks: range from 6 to 12 marks

Getting started

Programming is chiefly concerned with problem solving.
Write down a list of the tasks that you think a programmer carries out and the questions they need to answer when designing and building a computer program. When you have completed this unit, discuss with your peers, comparing and contrasting your ideas.

A Computational thinking

Many problems are solved by experienced programmers before they even touch a computer keyboard. You should be able to use computational thinking skills to analyse a problem, identify patterns and break complex tasks down into more manageable chunks so that they become easier to tackle.

Successful computer programming relies on exercising your computational thinking skills. It is these skills which help you to investigate a problem, analyse it methodically and identify potential solutions that you can further develop into working software applications.

Computational thinking skills can be divided into four separate, but interlocking, steps, as shown in Figure 1.1.

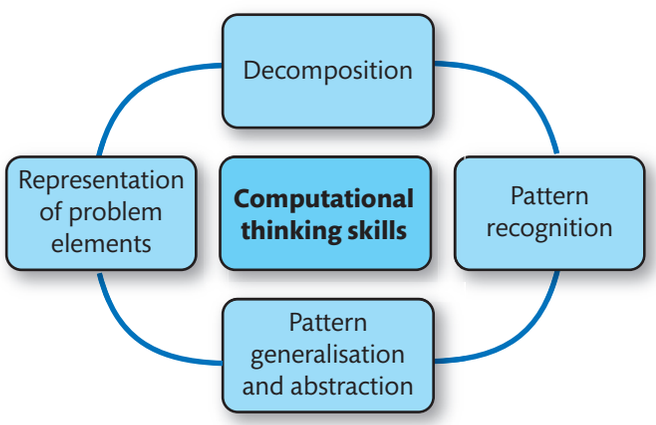


Figure 1.1: Computational thinking skills

You will examine each step in turn in the next section.

Decomposition

Decomposition is the process of breaking complex ideas down into smaller, more manageable parts. Sometimes this process may be called factoring. Generally, problems which are not decomposed prove to be more difficult to solve. The process of breaking a larger problem down into a number of smaller problems often improves the chances of success. This is chiefly because it allows you to focus on just one thing at a time, permitting its details to be examined more closely.

Everyone uses decomposition every day, often without realising. For example, the process of making a family meal involves:

1. choosing an appropriate recipe to follow
2. calculating the correct quantity of ingredients for the recipe and the family size
3. collecting appropriate ingredients
4. preparing the ingredients
5. cooking the ingredients in the right order, using the correct methods and for the correct duration
6. putting the meal on plates, ready to be eaten.

In this way, a single problem (making a family meal) can be decomposed into at least six ordered sub-tasks, each of which could be further decomposed, if necessary, until the steps required to solve each task are relatively straightforward to understand. In programming, decomposition involves the following four stages.

Identifying and describing problems and processes

At this stage, you will list problems and processes concisely, using language that matches the problem's source. For example, if you are dealing with a financial problem, you should accurately use terms from the financial sector. This means that you need to be familiar with the technical language used in the business sector relevant to the problem.

Breaking down problems and processes into distinct steps

At this stage, you will decompose complex problems and processes into separate steps which, when taken together, can be reassembled correctly. There is no specific limit to

the number of steps included or of levels to which you may decompose. You will simply continue to decompose each step until you reach an acceptable level of understanding. For example, the problem of calculating someone's net pay (salary after tax is deducted) is decomposed into several steps in Figure 1.2.

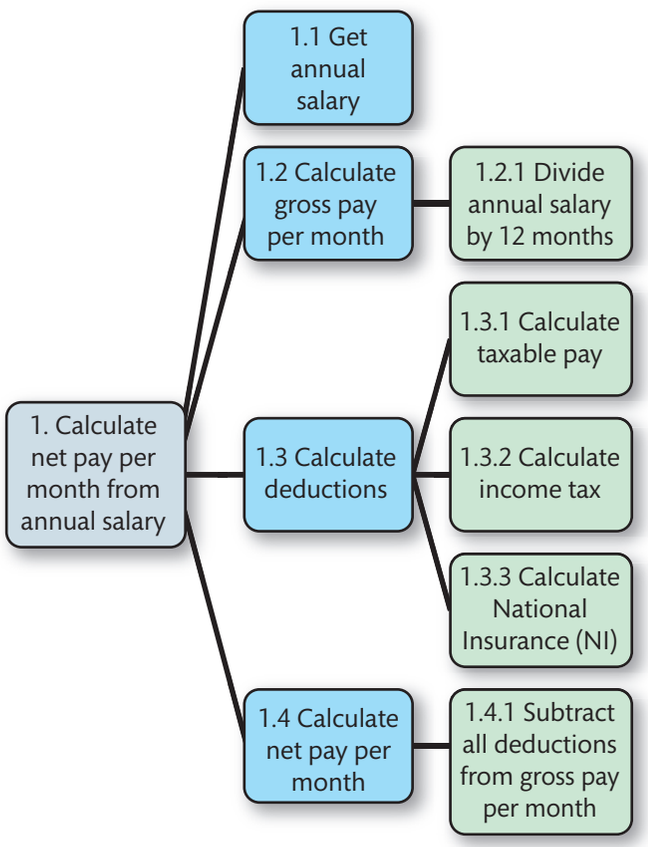


Figure 1.2: Decomposing a single process into progressively smaller steps

Describing problems and processes as a set of structured steps

At this stage, you will document the problems and processes that you have decomposed as a set of structured steps. This should be straightforward enough to be followed by you or by others.

Communicating the key features of problems and processes to others as relevant

At this stage, you will discuss the problems and processes with others. This may include other programmers or the client. Once you have decomposed a complex problem, it is possible to start analysing the steps involved to see if there are any repeating patterns.

Reflect

As a programmer, your communication skills must be flexible. You will need to adjust your delivery to meet the needs of different audiences. For example, programmers will understand technical 'jargon' while a client may not. On the other hand, the client may appreciate the use of business sector-specific language, but programmers may not.

You must be able to communicate a problem to others, because having a really clear understanding of the problem is essential to your eventual success in solving it. Albert Einstein is often quoted as having said, 'If you can't explain it to a six year old, you don't understand it yourself.'

Pattern recognition

Pattern recognition is the ability to see recurring attributes within the same problems and between different problems. For example, a new problem may have features that are similar to problems that have been previously encountered and solved. Recognising these repeating patterns can make problem solving much easier, as it can provide a good place to start.

Pattern recognition is a process based on five key steps.

- Identifying common elements or features in problems or systems. This involves:
  - examining problems or systems
  - listing elements or features that exist in each
  - highlighting those which exist in multiple places
  - recognising these as patterns.
- Identifying and interpreting common differences between processes or problems. This involves:
  - examining problems and processes
  - listing elements or features that exist in each
  - highlighting those that are unique to each
  - recognising these as differences.
- Identifying individual elements within problems. This involves:
  - examining problems to identify the inputs, processes (including selections and iterations) and outputs that are present.
- Describing patterns that have been identified.
- Making predictions based on identified patterns:
  - for each identified pattern, determine how it could be used in the future or how it may appear in similar situations.

Pattern generalisation and abstraction

In computing, abstraction is a concept whereby systems are split into different layers, with each layer hiding the complexity of the layer existing beneath it. This allows a programmer to use a feature without having to know exactly how it works: the irrelevant and intricate mechanics are simply 'abstracted' away or removed.

Pattern generalisation happens when relationships between patterns can be identified and simple conclusions can be drawn. For example, patterns can be identified even when, at first, it does not look like there are many similarities, as shown in the photos.

- Repeating design patterns (if you look closely...)

In each of the forms of transport shown in Figure 1.3, you can see similar elements repeated in the design patterns. Each vehicle includes four wheels, two axles, a chassis, a steering mechanism and so on. In order to solve a problem effectively, you must look beyond the obvious (in this case, the physical differences between the three forms of transport). Instead, try to identify the elements that are common and the relationships between these elements.

There are two parts to this phase of computational thinking.

First, **identify the information required to solve an identified problem**. You can achieve this by:

- knowing what information you need
- knowing your reasons for needing this information (the 'rationale')
- knowing the format in which this information needs to be provided
- knowing how soon this information is required to prevent the solution from being delayed.

Second, **filter out the information *not* required to solve an identified problem**. You can achieve this by:

- knowing what information is not needed, as this will be a distraction
- knowing (and justifying) why you have excluded this information.

Representing parts of a problem or system in general terms

To do this, you need to identify the following.

- **Variables** – these are the values in a problem or system that may change, which are typically input by the user or as the result of a required calculation.
- **Constants** – these are the values in a problem or system that do not change often or that remain fixed for a reasonable period of time (e.g. the base rate of income tax being 20%).
- **Key processes** – these are the processes that are absolutely critical to understanding a problem or how a system works.
- **Repeated processes** – these are processes that occur multiple times within a problem.
- **Inputs** – these are the values entered into the system, including the units used and, potentially, any valid values or ranges (e.g. where gender is 'M' for male or 'F' for female, or where a house price has to be between £20,000 and £2,000,000).
- **Outputs** – this is information presented to the user in a required format, which is generally specified by the client as part of their requirements.

Discussion

In small groups or pairs, think about an everyday process such as calculating the cost of decorating a room with fresh paint. Try to determine the variables, constants, key processes, inputs and outputs that are involved.

Algorithm design

An algorithm is simply a set of instructions that is followed to solve a problem or perform a particular stage of processing in the overall solution, for example to validate user inputs. Programs may be made from a collection of many different algorithms.

A core part of the algorithm design is the description of the main processes, the required inputs, the desired outputs and any data storage that is required.

Link

Algorithms used to design mobile apps are discussed in *Unit 17: Mobile Apps Development*.

Worked Example: Creating an algorithm

You are now going to explore a simple algorithm that can be used to convert an amount of UK pounds into American dollars (D) or euros (E), depending on the user's choice.

Step 1: Understand the problem

- *Make sure that you fully understand the problem you have been asked to solve.*

You need to convert an amount of pounds into dollars or euros (D or E), depending on the user's choice.

Step 2: Identify the inputs

- *What values will be input by the user?*
- *What type of data will this be?*

An amount in pounds.

A choice of converting to D or E.

Step 3: Identify the processes

- *What kind of calculations are involved?*
- *Are there any special values we need to know, for example conversion rates?*

You need the conversion rates from pounds to D and from pounds to E.

You also need to validate the currency choice to D or E.

Step 4: Identify the data storage

- *This would mean thinking about storing data that would be needed next time the solution is used.*
- *If this is needed, what type of data do you need to store?*

Data does not need to be stored.

Step 5: Identify the outputs

- *What information must be output?*
- *In what format must the output be shown?*

You need to output the converted sums in a decimal format using the correct currency symbols ($ or €).

Step 6: Collect your notes

- *Use a quad diagram to represent the inputs, processes, data storage and outputs required.*

A breakdown of this problem into these four separate areas is shown in Figure 1.3.

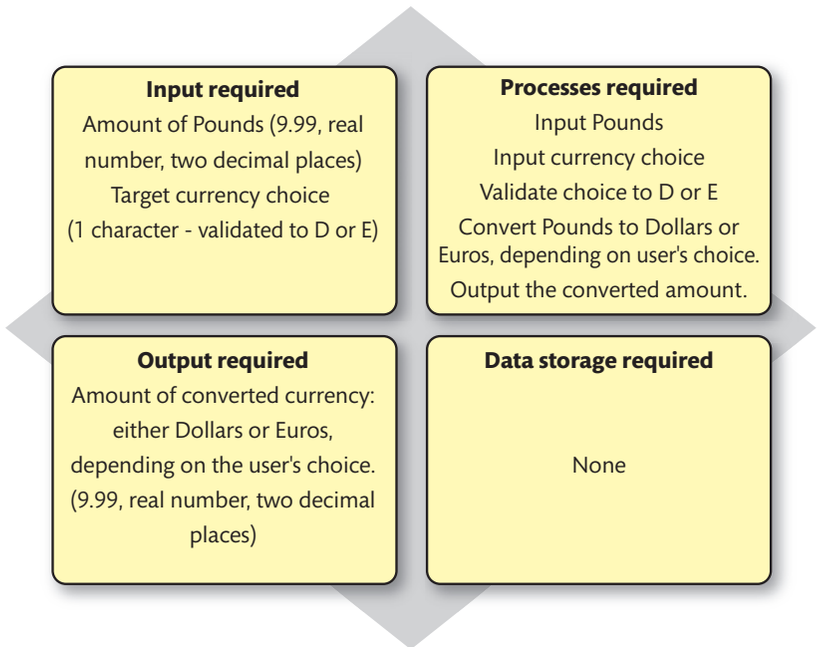


Figure 1.3: A quad diagram showing inputs, processes, outputs and data storage

The quad diagram is a simple way of starting to organise your thoughts while exploring a problem and starting to assemble a potential solution.

The processes you have identified will need to be developed into more detailed algorithms before you can form a potential solution.

In this example, we have listed all the inputs and outputs and noted their formats (particularly the quantity of decimal places to use) and any validation that is required. When specifying formats it is typical to use 9 for a numeric digit, A for an alphabetic character and X for any character.

PAUSE POINT Can you explain what the assessment outcome was about? What elements did you find easiest?

Hint Close the book and make a list of the features of computational thinking skills and the steps you might take to start building an algorithm.

Extend Are there any other inputs, outputs or processes that you think have been missed?

B Standard methods and techniques used to develop algorithms

Because they can be complicated to communicate verbally, algorithms are best represented using a number of different design tools. This section looks at two of the more common tools that can be used – structured English (pseudocode) and flowcharts.

Structured English (pseudocode)

Pseudocode is an informal English-like outline of the algorithm which can be converted to the target programming language.

Pseudocode should avoid commands or syntax that are found only in a particular programming language. Instead, standardised terms from English are used to specify particular types of actions. A categorised list of these is shown in Table 1.1. You may be introduced to others in your studies.

Table 1.1: Categorised pseudocode terms

Operations	Decisions	Repetition
BEGIN	IF	FOR
END	THEN	REPEAT UNTIL
INPUT	ELSE	WHILE
OUTPUT	ELSEIF (ELIF)	WHILE NOT
PRINT	WHEN	
READ		
WRITE		

Figure 1.4 shows pseudocode being used to describe a simple algorithm that validates a user's age between 18 and 60 (inclusive).

```
BEGIN
  REPEAT
    min = 18
    max = 60
    OUTPUT "What is your age?"
    INPUT age
    IF age < min OR age > max THEN
      OUTPUT "Error – age must be between 18 and 60"
    ELSE
      OUTPUT "Age is accepted"
    ENDIF
  UNTIL age >= min AND age <= max
END
```

Figure 1.4: Pseudocode which validates a user's age

You should note the use of **indentation** to highlight the hierarchy of the algorithm's logic. When indentation is used correctly, it should be possible to see which actions are performed in the true or false branches of each decision or which actions are being repeated by a loop.

Key term

Indentation – optional styling applied to pseudocode and the source code of certain programming languages to help convey the code's underlying structure to a reader. Indentation is created through the use of whitespace and the vertical alignment of related code elements. Different indent styles exist and are usually specified as part of an organisation's house rules.

Link

For more on how to produce pseudocode, including sequence, structure, operations, decisions and repetition, see Writing pseudocode, in Structured English (pseudocode) in *Unit 4: Software Design and Development Project*.

Interpreting pseudocode

To succeed in this unit, you must be able to not only produce pseudocode to solve set problems but also be able to correctly interpret existing pseudocode statements so that you can describe the tasks or processes they are performing.

Apply processes to calculate outcomes

Interpreting existing pseudocode involves applying the processes that are shown to determine potential outputs and actions, which will help you to understand the true purpose of the pseudocode presented. You should also be able to evaluate the structure and logic of the pseudocode provided against the original requirements: that is, does the solution meet the target user's needs?

Evaluate the structure and logic of given code against given requirements

As part of any development, you should try to create solutions that demonstrate effectiveness and efficiency of code and identify and fix any errors that may exist within the code.

Suggest improvements to logical structures and processes

Finally, you should be able to suggest improvements to logical structures and processes. Try to interpret the line-numbered pseudocode shown in Figure 1.5 and decide what it is trying to achieve.

```
1   BEGIN
2     INPUT value
3     total = 1
4     counter = 1
5     REPEAT
6       total = total * counter
7       Increment counter
8     UNTIL counter > value
9     PRINT total
10  END
```

Figure 1.5: Sample pseudocode

So, what does this pseudocode algorithm actually do?

If you examine the code line by line, you should be able to work out what it is attempting to do.

Line 1 – indicates the start of the pseudocode.

Line 2 – the user inputs a value, storing it in a variable called 'value'.

Line 3 – creates another variable called 'total', initialising it to 1.

Line 4 – creates a variable called 'counter', initialising it to 1.

Line 5 – starts a looping process.

Line 6 – adds 'counter' times 'total' to the existing 'total'.

Line 7 – increments 'counter' (increment means to increase by 1).

Line 8 – a loop condition checks the value of 'counter'; if 'counter' is greater than the value input in line 2, the loop will stop, otherwise, it returns to line 5 and performs the actions again.

Line 9 – prints the 'total' variable on screen.

Line 10 – indicates the end of the pseudocode.

Link

See Variables, in Pattern generalisation and abstraction, for a definition of variables. For more about loops, see Loops, in Control structures and, for a definition of operators, see Operators, in Arithmetic operations.

If you input 4 for 'value' and trace the values of 'counter' and 'total' line by line, you will see the following results (Table 1.2).

Table 1.2: Tracking pseudocode loops and variables

Value	4			
Counter	1	2	3	4
Total	1	2	6	**24**

Do you recognise this pattern? This is four factorial (denoted by 4!) which is 4×3×2×1 = 24. This pseudocode algorithm generates factorials from an inputted value. Algorithms can get much more complicated than this, of course, but identifying this is a good start.

Improve the effectiveness and efficiency of code

As part of any development, you should try to create solutions that demonstrate effectiveness and efficiency of code and identify and fix any errors that may exist within the code.

Link

See also Improve the effectiveness and efficiency of code, in Developing pseudocode, in *Unit 4: Software Design and Development Project*.

Identify and fix errors

Sometimes, errors will occur in the pseudocode and these may be difficult to find. For example, examine the following code (shown in Figure 1.6) and try to identify the error.

```
1   BEGIN
2     INPUT value
3     total = 1
4     counter = 1
5     REPEAT
6       total = total * counter
7     UNTIL counter > value
8     Increment counter
9     PRINT total
10  END
```

Figure 1.6: Sample pseudocode with an error

If you examine the code carefully, particularly between lines 5 and 8, you will see the problem.

In this version of the pseudocode the REPEAT...UNTIL loop stops when the 'counter' becomes larger than the 'value' entered by the user. However, if you look closely, you will see that the line of code which increments the 'counter' variable is not performed until line 8 – it is outside the loop, which cannot be correct. Doing this means that you will never make the 'counter' variable increase inside the loop. As a consequence, the loop condition will never become true and it will never end – you have created an 'infinite' loop.

To fix this, you would need to move line 8 back inside the REPEAT...UNTIL loop. Such errors can be very subtle, but you will be expected to be able to identify them and fix them. This process is called debugging.

Link

See also Identifying and fixing errors within pseudocode, in Developing pseudocode, in *Unit 4: Software Design and Development Project*.

Flowcharts using standard symbols

A flowchart is a graphical representation of the algorithm, showing its actions and logic through a set of standardised symbols which are standardised by the **British Computer Society (BCS)**.

Key term

British Computer Society (BCS) – the Chartered Institute for IT, which collaborates with the Government and industry to set and promote common standards within IT.

Link

To learn more about the flowcharts and the common BCS flowchart symbols, see Flowcharts and the use of standard symbol conventions, especially Table 4.1, in *Unit 4: Software Design and Development Project*.

Worked Example: Creating a flowchart

You are now going to use these symbols to build a flowchart that will represent the age validation algorithm as shown in figure 1.4.

Step 1
Think about the operations you need to perform, for example inputs, processes, outputs.

Step 2
Place them into a possible sequence.

Step 3
Are there any decisions to be made?
Are any actions dependent on these decisions?

Step 4
Are there any loops/repetitions required?
What conditions control the loops?

Step 5
Select the correct symbols needed and start to match these to the content already decided.

Step 6
Draw the flowchart, remembering to make sure that all flowchart symbols are correctly connected using arrows which demonstrate the correct logic flow when the flowchart is read.

Figure 1.7 shows the age validation algorithm using flowchart symbols.

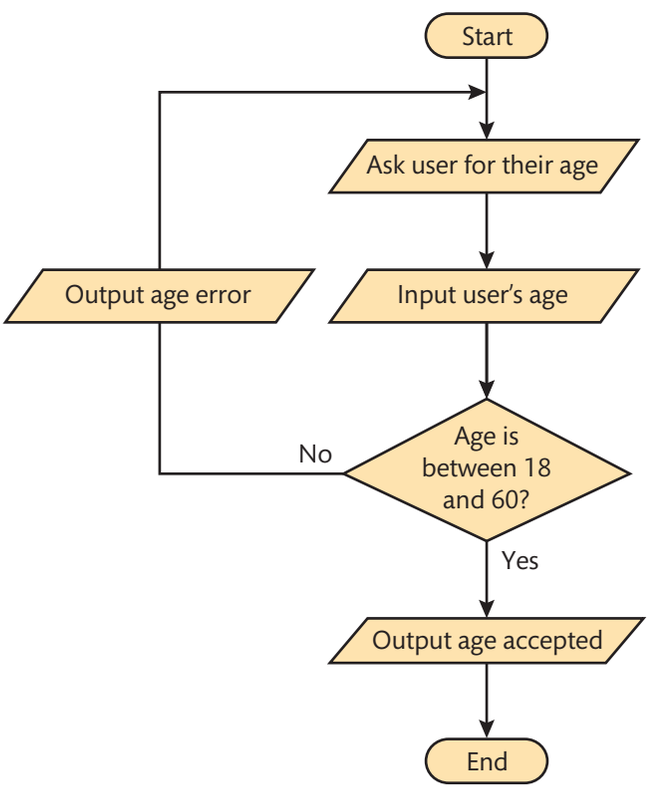


Figure 1.7: Flowchart representation of a simple algorithm

PAUSE POINT Can you explain what the assessment outcome was about? What elements did you find easiest?

Hint Close the book and make a list of common pseudocode operation, decision and action words.

Extend You can create algorithms using any standard design method or technique. Which method do you think is easier to use – flowcharts or pseudocode? Why do you think this is?

Assessment practice 1.1

A01 A02 A03

Read through the following scenario and using computational thinking skills (decomposition, pattern recognition, pattern generalisation and abstraction) solve the problem using standard methods and techniques such as pseudocode and flowcharts.

An online payment system, myBill, allows users to effortlessly pay for a variety of website purchases on the internet. Each myBill customer can link their account to a bank account, debit or credit card. Amounts are debited from the preferred funding source to create a myBill balance. Usually, myBill only takes what is needed to pay a transaction so the myBill balance remains at zero. However, sometimes a website will refund part of a transaction leaving the myBill balance in credit. When this occurs any subsequent purchase has to use the myBill credit first and then subtract the difference from the preferred funding source.

A simulation of this process is required for demonstration purposes where users can select their preferred funding stream, process a transaction (payment or refund) and keep an accurate running myBill balance. Each funding source set up should have its own balance and if a myBill transaction occurs when the preferred funding stream is unable to provide sufficient money, then the transaction should be politely refused.

Plan

- Do I fully understand the nature of the problem?
- Am I clear about what I am being asked to do?
- Do I know how to apply computational thinking skills to solve the problem?
- Am I able to use standard methods and techniques, such as pseudocode and flowcharts, to solve the problem?

Do

- I know what it is that I am doing and what I want to achieve.
- I can use a variety of computational thinking skills to analyse the problem and break it down into much more manageable chunks.
- I will start from the beginning of the scenario and work through to the end.

Review

- I can explain the computational skills I have used.
- I can explain how pseudocode and flowchart solutions both help to represent the algorithms I have created.
- I can explain how I would approach the hard elements differently next time (i.e. what I would do differently).

C Programming paradigms

In this section, you will learn about standard structures and conventions (programming paradigms) used to build and develop accurate, efficient and effective computer code to fulfil identified criteria and solve problems.

Handling data with a program

Data is typically represented in a program using an **identifier**. Many types of identifier are used in programs, but the most common types are constants (an identifier representing a value that will not change while the program is running) and variables (an identifier representing a value that may change while the program is running).

Key term

Identifier – a programmer-friendly name which represents a value stored in the computer's RAM. Before the use of identifiers, a programmer would need to know the actual memory address of a value in order to access it.

For example, when a person logs into a program, their name and password would typically be stored as variables, as these details would be different for each different user. However, a program calculating the price of a new television in a shop's electronic till would need to ensure that the current rate of value added tax (VAT) is added to the price. This rate would not change for every sale, so it could be set as a constant and only be modified if it changed in the Chancellor of the Exchequer's budget.

Defining and declaring constants and variables

In order to create (or declare) an identifier, whether it is a constant or a variable, you usually provide the programming language with two things: a name and a data type.

Almost all programming languages support the concept of data types. A data type is used to define what kind of value a variable or constant can store, what operations can be performed upon it and its behaviour within the program. Most programming languages offer many different data types for the programmer to use.

You should always choose a sensible and meaningful name. For example, a good variable name for storing the user's name would be 'username'. Some names cannot be used because they are reserved by the language for a particular use, typically because they are command words. These 'reserved' words vary between different programming languages, so be careful.

Link

A list of reserved words for JavaScript is shown in *Unit 17: Mobile Apps Development*.

To learn about the most common data types used in computer programs to define and declare constants and variables, see Defining and declaring constants and variables, in *Unit 4: Software Design and Development Project*.

Managing variables

Once you have established the variables for a program, they will need to be managed by determining what type of variable they are and by applying naming conventions.

Local and global variables

Most programming languages contain the concepts of **local and global variables**.

Key terms

Local variables – these are limited to being used in the block of code in which they are declared, for example within a particular function.

Global variables – these can be used anywhere in your program code.

Link

To learn more about local and global variables, see Local and global variables, in Managing variables, in *Unit 4: Software Design and Development Project*.

Naming conventions

Apart from selecting meaningful identifiers, you will find that many different naming conventions exist for creating identifiers, although these will vary depending on the programming language you are using.

Two widely popular techniques that are used by industry professionals are **snake_case** and **camelCase**. Table 1.3 demonstrates some variables named using these two naming conventions.

Table 1.3: snake_case and camelCase

Identifier needed for:	snake_case	camelCase
user's password	user_password	userPassword
gross pay	gross_pay	grossPay
customer's postage charge	customer_postage_charge	CustomerPostageCharge

As you can see, snake case uses an underline to link multiple words together. Camel case prefers to capitalise the first letter of each word and close any spaces; making the first letter of the first word uppercase is optional.

Research

Many software publishers recommend standards for naming identifiers. Microsoft® freely provides online documentation containing advice and guidance on these technical issues with the aim of encouraging good programming practices. Visit the following online resource to find out about their recommended identifier naming conventions:

https://msdn.microsoft.com/en-us/library/ms229045(v=vs.110).aspx

Link

For more on the topic of naming variables, see Naming conventions, in Managing variables, in *Unit 4: Software Design and Development Project*.

Arithmetic operations

The majority of algorithmic solutions that you design will involve the use of arithmetic operations. These can be categorised as mathematical **operators**, relational operators, Boolean operators or date/time.

Key term

Operators – these are special symbols (or multiple symbols) which tell the program to perform a specific arithmetic, relational or logical operation on its data. Operators have to be used in a specific order of precedence (this describes the order of execution). You may be familiar with this concept from using BIDMAS or BODMAS (Brackets, Indices/pOwers, Divide and Multiply, Add and Subtract) in mathematics.

Link

To learn more about these arithmetic operators, see Arithmetic operations, in *Unit 4: Software Design and Development Project*.

Mathematical operators

The four core arithmetic operators will be familiar from your mathematics studies: add, subtract, divide and multiply. In computer science, symbols that are different from those you will be familiar with are often used, so be careful.

Link

See Table 4.7 and the section Mathematical operators, in Arithmetic operations, in *Unit 4: Software Design and Development Project* for the core mathematical operators.

In addition to the four core mathematical operators, there are also **modulus division** and **integer division**, the symbols for which are shown in Table 1.4.

Table 1.4: Modulus and integer division operators

Operators	Traditional symbol	Computer science operator
Modulus division	Mod	% MOD Modulo Rem
Integer division	/	DIV

- **Modulus division** simply divides two numbers and returns the whole number and remainder. This is probably the type of division you performed before you understood fractions and decimals.
  For example, you are probably familiar with the 24-hour clock.
  Well, you are likely to perform a modulus division of 12 to convert 15:00 to 3pm.
  For example 15 mod 12 = 1 remainder 3
- **Integer division** is simply when one integer number (whole number) is divided by another and you are only concerned with the integer part of the result. In other words, the remainder (decimal) is discarded, e.g.
  Normally 10 / 4 = 2.5
  But 10 DIV 4 = 2

Relational operators

These types of operators define the relationship between two different values. Generally speaking, there are six operations that you need to be familiar with.

Link

To learn about the six relational operators that you need to know about, see Relational operators, in Arithmetic operations, in *Unit 4: Software Design and Development Project*.

Boolean operators

Boolean or logical operators are used to combine expressions together. They give you the ability to test multiple conditions.

Link

To learn more about Boolean operators, see Boolean operators, in Arithmetic operations, in *Unit 4: Software Design and Development Project*.

Date/time

Many programming languages that support date and time data types use a variety of functions and operators to perform basic arithmetic on them. For this reason, it is difficult to identify particular operators for performing calculations with dates and times. However, Figure 1.8 demonstrates the use of simple date/time arithmetic in Python® through the use of the common + and – operators.

```
# Date/Time arithmetic in Python
import datetime

#grab date and time
today = datetime.date.today()
now = datetime.datetime.now()

#specify differences
daydiff1 = datetime.timedelta(days=14)
daydiff2 = datetime.timedelta(weeks=3)
timediff1 = datetime.timedelta(hours=4)

#display calculated differences
print "Two weeks from now is:", today + daydiff1
print "Three weeks ago was:", today - daydiff2
print "In four hours the time is:", now + timediff1
```

Figure 1.8: Python's® use of + and – operators to perform date/time calculations

Link

To learn more about how to apply date and time data types in C++, see Date and time, in Arithmetic operations, in *Unit 4: Software Design and Development Project*.

Built-in functions

Almost all programming languages have built-in library functions which programmers can use when solving complex problems. These allow the programmer to perform common but crucial tasks on data, such as formatting its appearance, finding the length of a string or the square of a number. You can also download and install third-party functions from reputable websites to expand programming languages.

Built-in functions are grouped by category: arithmetic functions, string handling functions or general functions.

Arithmetic functions

Arithmetic functions perform mathematical operations.

- Random

Random numbers are important for generating test data, particularly for statistics and probability applications and introducing the element of unpredictability into a program, e.g. for simulation or games of chance such as dice, roulette or card-based games such as pontoon or poker.

Most programming languages support the concept of pseudorandom numbers. These numbers are often generated (through hardware or software algorithms) via a process of manipulating seemingly random events (real-time clock readings, input/output activity etc) to generate a pool of random values.

It is quite common for the pseudorandom number generator (PRNG) to be given an initial seed value that acts as a starting point for the creation of the pseudorandom number pool. If multiple starting points are chosen too quickly in succession, e.g. inside a short loop, it is possible for a PRNG to generate the same sequence of random numbers, which essentially defeats the objective, so caution is advised.

Most programming languages have built-in functions which return a random number within a requested range. Figure 1.9 shows a typical C# method for generating and displaying pseudorandom numbers.

```
using System.IO;
using System;

class Program
{
    static void Main()
    {
        //create array for months of the year
        string[] months = new string[12]
        {"Jan","Feb","Mar","Apr","May","Jun","Jul","Aug","Sep","Oct","Nov","Dec"};

        //initialise prng with a time-dependent default seed value
        Random rnd = new Random();

        // will create a number between 1 and 12
        int month = rnd.Next(1, 13);

        //output the random month name
        Console.WriteLine("Month is {0}", months[month-1]);
    }
}
```

Figure 1.9: Selecting a random month using a built-in PRNG in C#

Range

Although many languages have similar functions, Python's® Range function is an example of a good utility function. Despite being able to arithmetically generate a list of numbers, its uses can be quite general-purpose, especially when used in conjunction with FOR loops.

Range is a flexible function, accepting a variety of different arguments as shown in Figure 1.10.

```
# range function examples

# One argument, generates numbers from 0 to 5
for i in range(6):
 print(i)

# Two arguments, generates numbers from 2 to 7
for i in range(2, 8):
 print(i)

# Three arguments, generates numbers from 1 to 10 in steps of 2
for i in range(1, 10, 2):
 print(i)

# Going backwards, counting down from 10 to 0 in steps of 2
for i in range(10, 0, -2):
 print(i)
```

Figure 1.10: Python® range function

The next example as seen in Figure 1.11 uses the range function to generate the counter values to traverse the array's 7 elements using a FOR loop.

```
# More complex example of range function in Python

days = ['Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun']
size = len(days)

for counter in range(0, size):
    print(days[counter])
```

Figure 1.11: Use of range function to control a FOR loop

Research

Investigate other programming languages to see if functions are available with similar functionality to Python's® Range function.

Round

An important objective when dealing with any problem involving money or quantities is the ability to round decimal numbers to a desired number of places or simply round to an integer (whole number).

As you might expect, most programming languages contain built-in functions to perform a simple rounding function although you should take care with numbers such as 1.5 as some rounding functions will round up and others may unexpectedly round down.

Figure 1.12 shows a simple example of using a C# built-in function to round a decimal number to a preferred number of decimal places.

```
using System.IO;
using System;

class Program
{
    static void Main()
    {
        //original floating point values
        float originalNumber1 = 1.23f;
        float originalNumber2 = 3.15f;
        float originalNumber3 = 1.9f;

        //rounded numbers - parameters are original number, dec places
        float roundedNumber1 = (float)Math.Round(originalNumber1, 1);
        float roundedNumber2 = (float)Math.Round(originalNumber2, 1);
        float roundedNumber3 = (float)Math.Round(originalNumber3, 0);

        //output the rounded numbers
        Console.WriteLine("roundNumber1 is 1.2, roundedNumber1);
        Console.WriteLine("roundNumber2 is 3.2, roundedNumber2);
        Console.WriteLine("roundNumber3 is 2, roundedNumber3);
    }
}
```

Figure 1.12 Using the round method of the Math class as a built-in function

Additional functions may be available which always round down ('floor') or always round up ('ceiling') as shown in Figure 1.13.

```
using System.IO;
using System;

class Program
{
    static void Main()
    {
        //original floating point value
        float originalNumber = 2.45f;

        //rounded numbers - parameter is the original number
        float roundedNumber1 = (float)Math.Ceiling(originalNumber);
        float roundedNumber2 = (float)Math.Floor(originalNumber);

        Console.WriteLine("roundNumber1 is 3, roundedNumber1);
        Console.WriteLine("roundNumber2 is 2, roundedNumber2);
    }
}
```

Figure 1.13: Using floor and ceiling

In this example, ceiling would give 3.0 and floor would give 2.0; the variables are still floating point numbers. However, you must remember that a negative value such as –2.8 would 'floor' to –3 and 'ceiling' towards –2.

- Truncation

In classical programming, truncating a number should simply 'chop off' the decimal part of a floating point number leaving the integer part behind.

String handling functions

Although strings are simply a collection of characters, they may still require processing as part of any potential solution. This task is commonly known by programmers as string handling. Programmers often have the need to perform certain operations on strings and most high-level languages have built-in library functions to help.

Common tasks are shown below, with C# practical code examples showing their implementation.

- Find the length of a string in characters

```
string month = "January";
int length;

length = month.Length;
Console.WriteLine("{0} is {1} characters long", month, length);
```

In this example, the **length** property of the 'month' **string object** is used to access the string's number of characters.

- Examining single characters

```
string month = "January";
char singleLetter;
int whichLetter = 3;

singleLetter = month[whichLetter];
Console.WriteLine("Character {0} of {1} is {2}", whichLetter, month, singleLetter);
```

In this example the output will be:

```
Character 3 of January is u
```

This is because 'u' is the character in position 3; the string starts with a 'J' in position 0.

- Extract a sub-string (part of a string)

```
string month = "January";
string subString;
int startPos = 3;
int howMany = 2;

subString = month.Substring(startPos, howMany);
Console.WriteLine("Substring is {0}", subString);
```

In this example the output will be:

```
Substring is ua
```

This is output because we start extracting at position 3 ('u'), and we want to extract the next 2 characters, including the first one ('ua').

Link

To learn more about common arithmetic functions, see Arithmetic functions, in Built-in functions, in *Unit 4: Software Design and Development Project*.

Key terms

Concatenation – is the process of joining things together. In programming, it is used to describe the process of joining shorter strings together to form longer ones, e.g. 'Hello' and 'World!' to form 'Hello World!'.

Casting – is where a value is converted from one type of data to another, potentially losing data during the process, e.g. when truncating decimal places when converting a fractional number to an integer. Casting can be implicit (the language automatically handles it for the programmer) or explicit (the programmer has to specify the conversion).

▶ **Concatenating** two strings

```
string month = "January";
string saying = " has 31 days.";
string combined;

combined = month + saying;
Console.WriteLine("Joined string is {0}", combined);
```

In this example the output will be:

```
Joined string is January has 31 days.
```

This is output because we use the '+' operator to join the two strings together. Other techniques are available in C#, but this is perhaps the simplest to use.

▶ String conversion

It is important to understand that numeric values can be stored as integers (if 'whole') or floating point (if they need a decimal part) or even as a string.

If the number is stored in a string it is generally impossible to perform arithmetic on it. Sometimes this might be ok; a telephone number typically only has digits and could safely be stored in a string as it is unlikely we would want to add or subtract with it.

However, sometimes we may legitimately want to convert from a number (integer or floating point number) to a string or vice versa. Most programming languages do this via built-in functions or data type **casting**.

Integer/floating point to string

Sometimes it is necessary to convert a numeric value (integer or floating point) to a string representation. This may happen because we want to format it in a particular way, e.g. as currency with a £ or $ symbol prefixed or simply to store it in a text file.

Figure 1.14 demonstrates how Python® converts integers and floating point numbers to strings.

```
# Converting numbers to strings in Python

# store the numbers
num1 = 1.4
num2 = 99

#convert them using str() function
string1 = str(num1)
string2 = str(num2)

#prove they are strings
print "String1",string1,"is",len(string1),"character(s) long"
print "String2",string2,"is",len(string2),"character(s) long"
```

▶ **Figure 1.14:** Python® conversion of numbers to strings using built-in functions

String to floating point number

This C# example demonstrates the use of Parse method (function) of the float class.

```
using System.IO;
using System;

class Program
{
    static void Main()
    {
        string mystring = "123.45";
        float mynumber = 0;

        mynumber = float.Parse(mystring);
        Console.WriteLine("My number, 100 times bigger is {0}", mynumber * 100);
    }
}
```

We can prove the number has been successfully converted by performing a simple arithmetic operation on it, in this case multiplying it by 100.

String to integer

This C# example demonstrates the use of the Parse method (function) of the int class.

```
using System.IO;
using System;

class Program
{
    static void Main()
    {
        string mystring = "123";
        int mynumber = 0;

        mynumber = int.Parse(mystring);
        Console.WriteLine("My number, subtract 1 is {0}", mynumber - 1);

    }
}
```

Once again, we prove the number has been successfully converted by performing a simple arithmetic operation on it, this time by simply subtracting 1.

General functions

Most programming languages that execute on the command line interface (CLI) or shell have functions or commands that allow the user to input data and output messages. These are both fairly fundamental objectives for any programmed algorithm and you are certain to encounter them in any programming language.

The following examples focus on functions found in the Python® programming language.

Link

To learn more about common string handling functions, see String handling functions, in Built-in functions, in *Unit 4: Software Design and Development Project*.

Input

Python's® input function is used to store user input that has been keyed. It does so by displaying a user prompt, waiting for keyboard input and then storing this in a specified variable in RAM once the enter/return key is pressed. Figure 1.15 shows this Python® code and on-screen prompt.

Link

To learn about the built-in general functions in C++, see General functions, in Built-in functions, in *Unit 4: Software Design and Development Project*.

```
# Simple use of input function in Python to get user input

username = input('Enter your name: ')
```

Figure 1.15: Python® input function

Tip

This use of the Input function is specific to Python® 3; in earlier versions, the raw_input function should be used instead.

Research

The majority of programming languages need to permit keyboard input and screen output.

Compare the use of the Python® **print function** shown in Figure 1.16 with the C# **Console.WriteLine function** shown in Figure 1.18.

What is C#'s equivalent to the **input function**?

As you can see, the use of the prompt is important because, without it, the user would not know what the program is expecting from them (or that the program has halted for their input). Of course, at the moment, you are not doing anything with the input.

Print

Python® also has the ability to display messages on the CLI/shell using the Print function. The example shown in Figure 1.16 expands the previous example so that the computer responds with the number of characters in your input name by using the Print function.

```
# Simple use of input function in Python to get user input

username = input('Enter your name: ')

length = len(username)

print('Hello,', username, '!', sep='')
print('Your username is', length, 'character(s) long.', sep=' ')
```

Figure 1.16: Use of Python's® print function to display text and variables

You should notice that the 'sep' argument is used to specify the character to use between the different parts of the print string. If it is not specified, the default separation is a single space.

Open, close, read and write

Designing any software solution typically involves dealing with persistence of data. A computer's data storage in its random access memory (RAM) is said to be volatile – it is lost when power is removed, that is, when the computer is switched off.

This means that it is fine to store data in RAM while the program is running but to have data existing between program uses, especially when the power is removed, requires the use of a more permanent (non-volatile) form of data storage, for example magnetic storage, such as a hard disc, or a universal serial bus (USB) flash drive.

The most common form of non-volatile data storage is the data file and these are supported by most programming languages, including Python® and C#.

Basic Python® data files can be explored using just four main built-in functions:

- **Open** – opens a file
- **Read** – reads data from a file into RAM
- **Write** – writes data from RAM to a data file
- **Close** – closes a file.

The example shown in Figure 1.17 demonstrates the creation of a simple data file (in ASCII text file format) and its reading back into the RAM to verify its contents.

```
# Simple example showing the creation and reading of a data file

# Open a new file for writing
myfile = open("demo.txt", "w")

#write data to the file
myfile.write( "This is my test data");

# Close the opened file
myfile.close()

# Re-open the file, this time for reading
myfile = open("demo.txt", "r")

#write data to the file
message = myfile.read();

#print the contents
print ('Data stored in file was:',message,sep=' ');

# Close the opened file again
myfile.close()
```

Figure 1.17: Basic Python® data file handling

Research

Investigate other programming languages to see if built-in functions are available with similar functionality to Python's® Input, Print and Output functions. C++ is covered in *Unit 4: Software Design and Development Project* but you should also research other programming languages.

Validating data

You may have heard of the expression 'garbage in, garbage out' or GIGO, which suggests the rule that the quality of output is directly dependent on having sensible inputs. **Validation** is the process that attempts to prevent nonsensical inputs.

Although the use of checkboxes, buttons and list boxes limit input choices, most programs still require validation to handle the probability of problematic input from the user, particularly when using traditional keyboard input.

Validation check techniques

The term "constraints" is often used to describe limiting factors which can be used to validate user inputs. Although constraints are often available as special functions, e.g. NotNull, NotBlank, etc. in certain programming languages, they can be created using standard validation check techniques, as shown throughout the following section.

In this section, you will look at a variety of different validation check techniques that programmers can use to validate their programs.

It is very important to build validation rules into a solution. They will check whether different inputs are sensible and prevent inaccurate results, **run-time errors** or fatal application crashes.

Range check

A range check assesses whether data entered is within a valid minimum-to-maximum range. For example, if a customer is limited to buying up to 10 of a particular item, then the valid range would be 0 to 10. Input outside this inclusive range would be considered invalid. Typical C# program code to perform this type of validation is shown in Figure 1.18.

Key terms

Validation – a process that checks to see if an input value makes sense and is reasonable before it is processed. Validation does not check to see if the data has been keyed in accurately – this is a separate process called verification.

Run-time error – a problem that occurs while an application is being used. These errors result in the application locking (refusing to accept user input) or crashing (terminating and returning the user to the device's menu or desktop).

Link

To learn about other validation check techniques (data type, range, constants and Boolean), see Validation check techniques, in Validating data, in *Unit 4: Software Design and Development Project*.

```
const int MIN = 0;       //minimum value of the range
const int MAX = 10;      //maximum value of the range

string strNumber;        //string to temporarily store our input
int qtyValue;            //our inputted quantity

//perform loop while quantity is outside range
do
{
    //input quantity
    Console.WriteLine("Enter quantity between {0} and {1}:", MIN, MAX);
    strNumber = Console.ReadLine();
    qtyValue = int.Parse(strNumber);

    //check is quantity outside range
    if (qtyValue < MIN || qtyValue > MAX)
    {
        Console.WriteLine("Sorry, the quantity entered is outside the allowed range");
    }

} while (qtyValue < MIN || qtyValue > MAX);

Console.WriteLine("Your quantity of {0} is valid, thank you.",qtyValue);

//@TODO other things with the quantity...

//wait for keypress to continue
Console.ReadKey();
```

Figure 1.18: Validation routine for a defined range in C#

Of course, an input of 0 would indicate that the customer is not trying to purchase any number of the item. However, it is still a logically valid input and would pass validation.

Ranges do not have to be limited to numerical input. For example, the characters 'A' to 'F' could also represent a valid range.

Length check

A length check assesses how many characters have been entered. Some very well-known inputs have limited length, for example:

- short message service (SMS) texts are limited to 160 characters
- tweets were originally limited to 140 characters (derived from SMS length minus 20 characters for the user's unique address).

Figure 1.19 shows a C# extract which limits character input to a predetermined maximum length set by a constant.

```
const int MAX = 10;      //set maximum number of characters

string message;          //our message to input and process
int messageLength;       //our message's length in characters

//perform loop while string too long!
do
{
    //input string
    Console.WriteLine("Enter message (max {0} characters)", MAX);
    message = Console.ReadLine();

    messageLength = message.Length;

    //check its length
    if (messageLength > MAX)
    {
        Console.WriteLine("Sorry, your message of {0} characters is too long.",messageLength);
    }

} while (messageLength > MAX);
```

Figure 1.19: Validation routine for a typical length check in C#

Presence check

A presence check assesses whether data is present – that is, whether it exists. For example, a user has to input whether they are 'male', 'female' or 'would prefer not to say' when completing a registration input form. The presence check validates that they have not left the entry blank or unselected.

Type check

A type check assesses whether the data entered is of the correct data type, as in Figure 1.20. For example, if the user has to enter their age, this would require the input to be an integer (a whole number). If input of incorrect data types is not prevented by the programmer, it can cause a fatal run-time error.

Age?	A	(data type is character, this is invalid)
Age?	16	(data type is integer, this is valid)
Age?	01/09/16	(data type is date, this is invalid)

Figure 1.20: Validation responses based on a type check

Format check

A format check assesses whether the data entered is in the correct format: for example, whether a string containing a UK postcode follows the format 'PO1 3AX'.

- PO is the area, such as GL (Gloucester) – this has to be capitals, alphabetic, 1 or 2 characters.
- 1 is the district, usually between 1 and 20 per area – this has to be up to 2 numeric digits.
- 3 is the sector, usually covering up to 3000 addresses – this has to be 1 numeric digit.
- AX is the unit, usually covering up to 15 addresses – this has to be capitals, alphabetic, 2 characters.

A format check on a postcode would apply these rules as shown in Figure 1.21.

Postcode?	GL2 4TH	(postcode is valid)
Postcode?	W1A 1AA	(postcode is valid)
Postcode?	GL2 24TH	(postcode is invalid; sector can only be 1 numeric digit)

Figure 1.21: Validation responses based on a format check

Check digit

A check digit is a single character (usually a numeric digit) derived from an algorithm which is performed on a piece of data. The algorithm is designed to only generate this particular digit if the data (e.g. a string of characters) has exactly those characters and they are arranged in that specific order. Any incorrect character or swapping of character positions generates a different check digit value and fails the test. Check digits are used mostly to detect errors in inputted values such as barcodes, bank account numbers and software registration codes.

An ISBN-10 (10-digit International Standard Book Number) uses a check digit. For example, the previous version of this very book had an ISBN-10 of 1846909287. The last digit (7) is treated as the check digit. You can determine if this code is correct using the follow technique.

Step by step: Check digit validation

4 Steps

1 Multiply each digit by smaller and smaller positional weights, as shown in Table 1.5.

Table 1.5: Positional weights used to check an ISBN-10 checksum validation

Digits in ISBN	1	8	4	6	9	0	9	2	8	7
Factor of multiplication	×10	×9	×8	×7	×6	×5	×4	×3	×2	×1
Sub-totals	**10**	**72**	**32**	**42**	**54**	**0**	**36**	**6**	**16**	**7**

2 Sum or add up the sub-totals in all columns:

10 + 72 + 32 + 42 + 54 + 0 + 36 + 6 + 16 + 7 = 275

3 Perform **modulus division** by 11:

275 MOD 11 = 25 remainder 0

4 Is there a remainder of 0? Yes, there is, which means that the ISBN is valid. Any other result means an incorrect ISBN has been entered

Key term

Modulus division – this performs a division operation and returns the remainder rather than working out the decimal or fractional answer.

This technique identifies incorrect or transposed digits in the code. In the example given in the step by step, the last digit (7) is considered to be the check digit.

Spelling

A spelling check assesses whether the words being entered can be found within an electronic dictionary – that is, that they are valid words.

Error handling and reporting

Many modern programming languages have syntax features which are designed to handle run-time errors when they occur. If they did not have these features, the application would crash or lock unresponsively.

A common error handling technique is the use of the 'try...throw...catch' expression that can be found in many languages, including C++, Microsoft® C#, Oracle® JavaScript and PHP. This error handling technique works by 'trying' an operation, 'catching' any possible errors and (optionally) 'throwing' an appropriate exception. This approach prevents the application from failing the operation in an uncontrolled way, as this would cause the application to crash completely.

An example of this error handling technique is shown in Figure 1.23 (in the next section, Post-check actions) by illustrating the dangers of dividing by zero. In the example shown in Figure 1.23 the division operation is placed inside a try block, just in case the user has entered '0' (zero) as their second number. This has been done because dividing a number by zero can generate a serious error on a computer platform and may result in a run-time crash. By using the 'try..catch' block this is avoided by displaying the exception that has been caught, which, in this case, is a "DivideByZero" error.

Post-check actions

Error reporting is an important aspect of software development. It may be directed towards the user, by telling them that they have made a mistake, or towards the developer, so that they can understand where a program has encountered an error and the nature of this error.

Common techniques for error reporting include:

- displaying an on-screen error message and/or error code
- appending the error details to an electronic log file which can be viewed separately
- sending an email to the developer that includes the error details.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace test
{
    class Program
    {
        static void Main(string[] args)
        {
            float num1;       // first number
            float num2;       // second number
            float quotient;   // result of dividing first number by second
            string strNumber; // string to temporarily store our input

            quotient = 0;

            //get 1st number
            Console.WriteLine("Enter 1st number");
            strNumber = Console.ReadLine();
            num1 = float.Parse(strNumber);

            //get 2nd number
            Console.WriteLine("Enter 2nd number");
            strNumber = Console.ReadLine();
            num2 = float.Parse(strNumber);

            //try the division
            try
            {
                quotient = num1 / num2;
                Console.WriteLine("{0} / {1} = {2}", num1, num2, quotient);
            }
            catch (DivideByZeroException e)
            {
                //output the handled exception
                Console.WriteLine("Error exception caught was: {0}", e);
            }

            //wait for key press before finishing
            Console.ReadKey();
        }
    }
}
```

Figure 1.22: Microsoft® C# example of error handling using try and catch blocks

Tip

You should always take care to not display debug information on the user's screen. This is because it may accidentally reveal sensitive data or the inner workings of the program. This is particularly important when developing online applications in order to prevent hacking and potential fraud.

Control structures

The algorithms that control programs are built using a combination of three basic programming building blocks or control structures.

These control structures are sequence, selection and iteration.

1. Sequence – one action after another, none missed, none repeated.
2. Selection – actions chosen based on a supplied condition which is evaluated to true or false.
3. Iteration – actions repeated a number of times until a condition is no longer true.

Link

To learn more about post-check actions, see Post-check actions, in Validating data, in *Unit 4: Software Design and Development Project*.

It is quite common to see iterations referred to simply as loops and for selections to be called branches.

Loops

By their nature, loop control structures allow parts of an algorithm to be repeated. Although infinite loops are designed to repeat forever, most loops have controlling conditions that tell the loop when to stop or keep repeating. Loop conditions are formed in a similar way to those used in branches.

Most programming languages support a number of different loop types but, in general, they can be classified as being either pre- or post-conditioned. Pre-conditioned loops, such as FOR and WHILE**,** have a controlling condition that is situated before the actions which need repeating. If the controlling condition is not true, the loop will not work at all and these actions are not executed. The pseudocode for each pre-conditioned loop is shown in Figure 1.23.

FOR loop	WHILE loop

FOR counter = 1 to 10
 	
    OUTPUT counter
 	
ENDFOR
	counter = 1
	
	WHILE counter <= 10
	
	    OUTPUT counter
	    Increment counter
	
	ENDWHILE

Figure 1.23: Pre-conditioned loops

In addition, the FOR loop is generally differentiated from the WHILE loop because it is ideally used when the number of iterations is fixed and known before the loop starts.

For this reason, it is very useful when traversing arrays, linked lists and other data structures. Additionally, most FOR loops can count up and down and usually increase in set steps, for example jumping in twos or fours.

In contrast, post-conditioned loops such as REPEAT...UNTIL have their controlling condition after the actions being repeated. Because this condition is not tested until after the loop's actions have been executed, you can always rely on the loop working at least once.

REPEAT loop

counter = 1
 	
REPEAT
 	
    OUTPUT counter
    Increment counter
 	
UNTIL counter > 10

Figure 1.24: Post-conditioned REPEAT loop

BREAK is used to terminate a loop that is designed to work a fixed number of times: usually, a FOR loop. An example of using BREAK is shown in Figure 1.49.

This may seem rather silly, but it makes sense in situations such as linear search routines where a FOR loop may be used to freely traverse data items in an array or list but, once a particular value has been found, there is no need to continue. In this instance, the BREAK can be used to prematurely exit the loop in a controlled fashion.

Link

To learn more about these four kinds of loop, see Loops, in Control structures, in *Unit 4: Software Design and Development Project*.

The C# code for selections and iterations is similar to the JavaScript examples in *Unit 7: Mobile Apps Development*.

Branches

Branches allow you to make decisions within an algorithm, which are usually based on whether a specific condition is true or false.

Conditions are built from combinations of identifiers, **literals**, relational operators and logical operators. For example, to order a holiday online, you may have the following condition, as shown in Table 1.6.

Table 1.6: Branches example

age >= 18 AND validDebitCard = true						
age	**>=**	**18**	**AND**	**validDebitCard**	**=**	**true**
Identifier (variable)	Relational operator	Numeric literal	Logical operator	Identifier (variable)	Relational operator	Boolean literal

If the condition evaluates to true, one set of actions is performed. If the condition evaluates to false, another set of actions is performed (this can be optional). We can represent a branch using both pseudocode and a flowchart, as shown in Figures 1.25 and 1.26.

```
IF age >= 18 AND validDebitCard = true THEN
  OUTPUT "Holiday booked!"
ELSE
  OUTPUT "Cannot book holiday at this time!"
ENDIF
```

Figure 1.25: Representing a branch using pseudocode

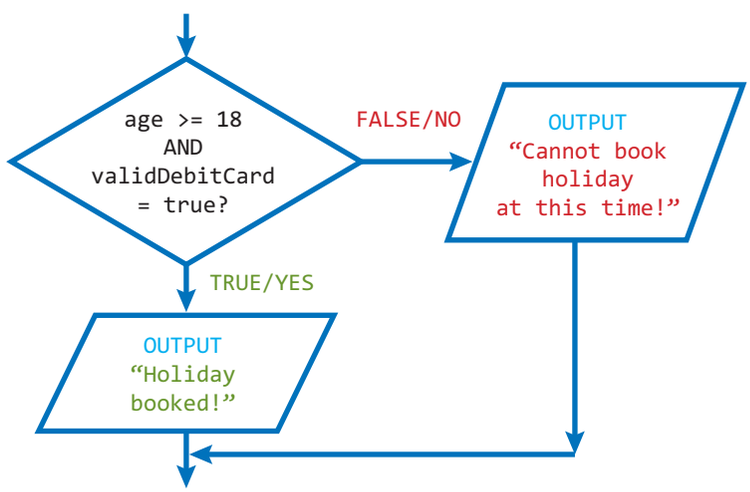


Figure 1.26: Representing a branch using a flowchart

Sometimes, we may wish to have multiple branches within the same part of the algorithm, particularly when there are several TRUE possibilities. For example, as part of a company's online loyalty scheme, customers are awarded bronze, silver or gold discounts depending on the value of goods they have previously purchased.

<£50 No discount

£50 - £100 bronze

£101 - £200 silver

£201+ gold

Key term

Literal – this is a value in an algorithm or program which can be numeric (e.g. 10) or string (e.g. 'hello').

Link

To learn more about types of branches, including IF, THEN, ELSE and ELSEIF (ELIF), see Branches, in Control structures, in *Unit 4: Software Design and Development Project*.

You can use a series of IF...THEN...ELSEIFs to handle this situation, as shown in Figures 1.27 and 1.28.

```
IF valueGoods < 50 THEN
   OUTPUT "No discount!"
ELSEIF valueGoods < 101 THEN
   OUTPUT "Bronze discount"
ELSEIF valueGoods < 201 THEN
   OUTPUT "Silver discount"
ELSE
   OUTPUT "Gold discount"
ENDIF
```

Figure 1.27: IF...THEN...ELSEIFs using pseudocode to represent the branch

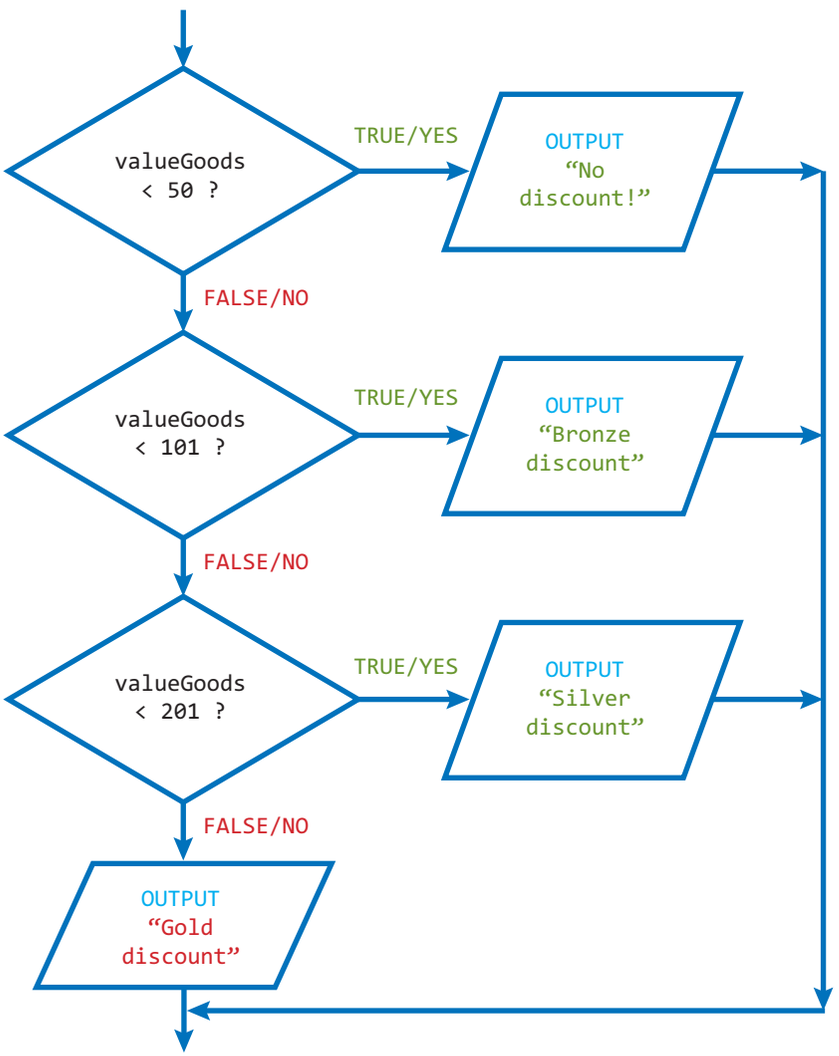


Figure 1.28: IF...THEN...ELSEIFs using a flowchart to represent the branch

In programming languages such as Python®, ELIF can be used to offer multiple alternate branches, as shown in Figure 1.29.

```
# Branch example program in Python

#!/usr/bin/python

valueGoods = 117;

if valueGoods < 50:
   print("No discount")
elif valueGoods < 101:
   print("Bronze discount")
elif valueGoods < 201:
   print("Silver discount")
else:
   print("Gold discount")
```

Figure 1.29: Python® ELIF branch example

Function calls

A function is a separate block of code that performs a specific job. Some programming languages define a function by stating that it should return a value of some kind, differentiating it from a procedure which merely performs an action. Many languages, particularly those in the C family, make no such distinction – they are simply 'functions'.

Functions, also called modules, subroutines or procedures, depending on the programming language, are responsible for performing a single task and are typically somewhere between 5 and 50 lines of code long.

Key term

Function – a separate block of code which performs a specific job.

The use of such functions means that code tends to be easier to write, easier to read and debug, reusable through multiple solutions and allows a single application to be divided and worked on by several programmers simultaneously (saving development time).

Figure 1.30 shows modules (functions) being called from a main program.

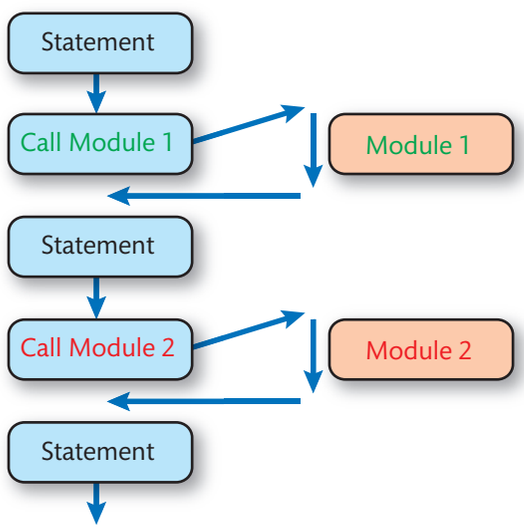


Figure 1.30: Modules (functions) being called from a main program

Program flow follows line by line until a function is called. When this happens, control is passed to the module which is then also executed line by line. At the end of the module, control is passed back to the line of code after the original function call. This pattern repeats until the end of the program. It is very common for the same functions

to be called multiple times in the life of a working program. Functions can even call other modules as necessary. Functions which are part of the programming languages are called library or built-in functions. Functions created by a programmer are called user or programmer-defined.

The C# code shown in Figure 1.31 demonstrates the use of a programmer-defined function to calculate the area of a circle, given a specific radius.

```
class Program
{
    const float PI = 3.14f;

    static void Main(string[] args)
    {
        float radius;
        float area;

        radius = 10.0f;
        area = calcAreaCircle(radius);  // function call

        Console.WriteLine("Area of circle is {0}", area);
        Console.ReadKey();
    }

    //function declaration
    public static float calcAreaCircle(float radius)
    {
        return PI * radius * radius;
    }
}
```

Figure 1.31: C# programmer-defined function

Declaration

Functions are usually defined (declared) in a separate section of the program's code, possibly even in a separate file (depending on the programming language). In the C# example shown in Figure 1.31, the function is declared at the end of the main function's code.

Arguments and formal paramaters

It is commonly accepted that the function's **arguments** are the values (variables, constants or literals) that are passed *into* the function to be processed. In Figure 1.31 this would be the 'radius' variable which is declared and initialised as a float variable with a value of 10 in the main function.

The function declaration will include not only the function's name ('calcAreaCircle') but also the **formal parameters** expected ('radius') and the return type of the function (it returns a float). Arguments and formal parameters do not need to have the same name but they typically have the same quantity, data type and order: that is, if a function expects three arguments (for example, two integers and a float) then these values must be passed to it for it to work correctly.

Calling functions

A function is called by using its name and, if needed, arguments are passed to it to be processed. The result is stored in a variable which is of the same data type as the function's return type.

Data structures

A data structure is a technique used to collect and organise data items into a formal structure, which allows more efficient processing. Although the availability of certain data structures varies between different programming languages, many are very common. These include strings, arrays (one and two dimensional), stacks, queues and records.

Sometimes, it is possible to program more efficiently through the selection and use of specific data structures, especially when combined with iteration (loop) control structures. Software developers need to become familiar with many data structures as they learn about different programming languages.

A data structure is used to store a collection of characters with one character minimally requiring one byte of RAM.

String (or text)

Strings may be of fixed length (e.g. 10 characters long) or use a special 'terminator' character to mark their end. This may mean that a string will require an additional character but it allows them to be dynamic in length.

Planet

0	1	2	3	4	5
E	a	r	t	h	#

Each individual character is accessible using its positional index, for example **Planet[2] is 'r'**, as seen in the sample C# code extract in Figure 1.32.

```
string Planet;
Planet = "Earth";

Console.WriteLine("Whole string is {0}", Planet);
Console.WriteLine("3rd character is {0}", Planet[2]);
```

Figure 1.32: Working with C# strings

This can be used to store simple text entered by the user, for example a username.

Strings may also be used to enter more complex text for processing, for example the content of an email, an instant message or a tweet.

Some languages may use a static one-dimensional array of characters to simulate a string.

Array (one-dimensional)

Classically, this is a static data structure (having a fixed size), but modern programming languages are generally more flexible, allowing an array to be dynamically resized as needed.

Typically, an array can store only one type of data, but any type of data (integers, characters, decimal, Booleans etc.) is acceptable.

For example, if you wanted to store 7 daily maximum temperatures (in degrees Celsius) for a local weather station, you could create an array of 7 decimal numbers.

Temperatures

0	1	2	3	4	5	6
12.50	10.45	12.30	14.60	17.70	11.20	12.50

Tip

Almost all programming languages have library functions which are freely available for the programmer to use when solving complex problems. These allow the programmer to perform common (but crucial) tasks on data, for example formatting its appearance, finding the length of a string or the square of a number. It is also possible to download and install third-party functions from reputable websites to expand the programming language.

Link

To learn more about how to define, declare and call functions, see Function calls, in Control structures, in *Unit 4: Software Design and Development Project*.

This may appear similar to a string (indeed, you can think of a string as an array of characters) and, again, it is possible to access individual elements (or items) in the array by using the required index, for example Temperatures[4] is 17.70.

Figure 1.33 demonstrates the creation and access of a simple one-dimensional array in C#.

```
//create the array of decimal temperatures
float[] temperature = new float[7];

//initialise each element
temperature[0] = 12.50f;
temperature[1] = 10.45f;
temperature[2] = 12.30f;
temperature[3] = 14.60f;
temperature[4] = 17.70f;
temperature[5] = 11.20f;
temperature[6] = 12.50f;

//select Thursday's and output it
Console.WriteLine("Temperature on Thursday is {0} degrees C", temperature[4]);

//wait for keypress to continue
Console.ReadLine();
```

Figure 1.33: C# one-dimensional zero-based array

Array (two-dimensional)

These data structures are similar to one-dimensional arrays but store multiple rows of data.

For example, if you wanted to store 7 daily maximum temperatures (in degrees Celsius) for a local weather station over three consecutive days, you could create a two-dimensional array of 7 by 3 decimal numbers.

Temperatures

	0	1	2	3	4	5	6
0	12.50	10.45	12.30	14.60	17.70	11.20	12.50
1	12.22	11.00	10.00	20.00	21.00	14.00	15.50
2	13.00	14.00	14.50	14.60	12.30	14.00	14.60

Tip

The actual order of indices may vary between different programming languages. The general rule is that the element access order will reflect the declaration order of the array.

Once again, it is possible to access individual elements (or items) in the array by using both indices, for example Temperature[4][1] is 21.00.

Higher orders of dimensions are possible. For example, a three-dimensional array could be used to store 21 temperatures for 5 different weather stations.

Record (or structure)

A record is a similar concept to an array but differs because it can store a mix of data types within its structure.

For example, storing a student's details.

Student

StudentID	Forename	Surname	Age	Enrolled
2001	Preston	Myla	21	True

In this example, StudentID and Age are integers, Forename and Surname are strings and Enrolled is a Boolean. A C# implementation of a simple record structure is shown in Figure 1.34.

```
using System;
using System.Linq;
using System.Text;

namespace ConsoleApplication1
{
    public class testStructure
    {
        //create the record structure
        struct Student
        {
            public int StudentID;
            public string Surname;
            public string Forename;
            public int Age;
            public bool Enrolled;
        };

        public static void Main(string[] args)
        {
            Student mystudent;    //declare mystudent of type Student structure

            //initialise the elements in the record structure
            mystudent.StudentID = 2001;
            mystudent.Surname = "Preston";
            mystudent.Forename = "Myla";
            mystudent.Age = 21;
            mystudent.Enrolled = true;

            //print the data in the record structure
            Console.WriteLine("ID        : {0}", mystudent.StudentID);
            Console.WriteLine("Surname   : {0}", mystudent.Surname);
            Console.WriteLine("Firstname : {0}", mystudent.Forename);
            Console.WriteLine("Age       : {0}", mystudent.Age);
            Console.WriteLine("Enrolled  : {0}", mystudent.Enrolled);

            //wait for a keypress to continue
            Console.ReadKey();
        }
    }
}
```

Figure 1.34: C# simple record structure using struct

It is also possible to create an array of structures to store multiple instances of a record. For this reason, a record (or struct) can be seen as the basis for simple record storage in a database table or a binary data file.

Lists

A list (usually called a **linked list**) has a number of data items connected together in a 'chain' using **pointers**.

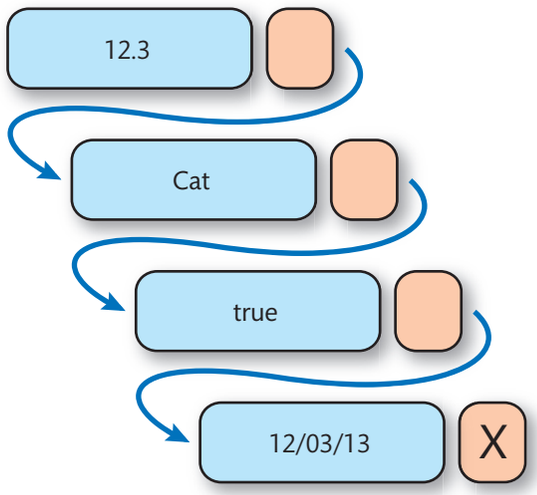


Figure 1.35: Linked list of four data items

Figure 1.36 shows the code for the linked list.

```
using System.IO;
using System;
using System.Collections.Generic;

class Program
{
    static void Main()
    {
        // Create a new LinkedList.
        LinkedList<String> ll = new LinkedList<String>();

        // Create a new LinkedListNode of type String
        LinkedListNode<String> lln = new LinkedListNode<String>( "Rebekah" );

        // Add the "Rebekah" node to the empty linked list
        ll.AddLast( lln );

        // Add more nodes to the linked list, but specify where...
        ll.AddFirst( "Josie" );
        ll.AddLast( "Christopher" );
        ll.AddFirst( "Matthew" );

        //display the linked list using a for loop

        Console.WriteLine("We currently have {0} student(s) stored.", ll.Count);
        foreach (var student in ll)
        {
            Console.WriteLine("Node's data is {0}", student);
        }
    }
}
```

Figure 1.36: C# simple linked list using built-in linked list class

Each **pointer** connects the next data item in the list.

Unlike a typical array, each data item may be a different data type. The last pointer has a NULL value to show the end of the list.

Key term

Pointer – this is a special type of variable which stores the memory address of another variable in the RAM. This allows the second variable to be accessed indirectly by referencing the first variable. Pointers are used in more advanced programming to make solutions more efficient but can be dangerous if not used correctly.

The logical order of the data items does not have to match its physical order in the RAM. Linked lists are fast, flexible alternatives to using an array.

After running this code, the output shown in figure 1.37 should be displayed.

```
We currently have 4 student(s) stored
Node's data is Matthew
Node's data is Josie
Node's data is Rebekah
Node's data is Christopher
```

Figure 1.37: Sample output of linked list

As you can see, C# has inserted each node into the requested location in the list. It is also possible to insert a new node between existing nodes.

Sets

Sets can be used to organise data and define the interrelationships between different sets of data. This allows you to easily search and **filter** potentially complex data. Some programming languages have data types which support set-style functionality and this logic can be used to form user permissions.

Key terms

Set – a collection of distinct objects. Sets can contain anything (e.g. names, numbers, colours or letters of the alphabet) and may consist of many different members.

Filter – by using a filter you can include or exclude certain values when running a search.

For example, you may have two groups of people with different access rights in a particular application.

Set A is the Admin group, which contains John, Ahmed and Jo.

This can be expressed as Set A = {John, Ahmed, Jo}.

Set B is the Finance group, which contains Claire, Jo, Niamh and Phil.

This can be expressed as Set B = {Claire, Jo, Niamh, Phil}.

You can represent these sets of data as a Venn diagram, as shown in figure 1.38.

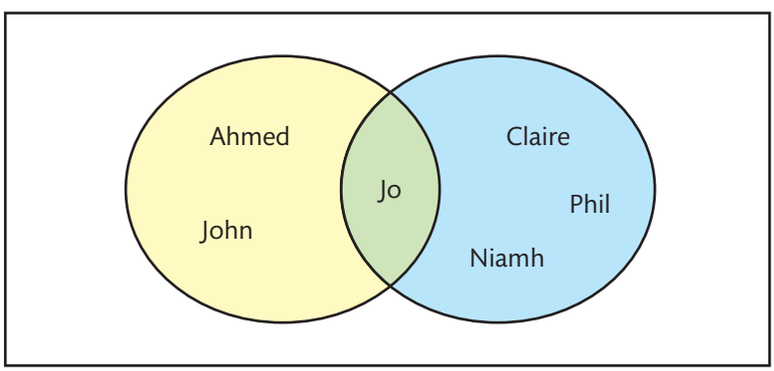


Figure 1.38: Venn diagram

Figure 1.39 demonstrates how programming languages such as C# can declare this type of set data and interrogate it.

```
using System.IO;
using System;
using System.Collections.Generic;

class Program
{
    static void Main()
    {
        //declare and initialise each set
        HashSet<string> admin = new HashSet<string>() {"John","Ahmed","Jo"};
        HashSet<string> finance = new HashSet<string>() {"Jo","Claire","Niamh","Phil"};

        //display the admin set
        Console.WriteLine("admin set contains {0} member(s): ", admin.Count);
        foreach (string member in admin)
        {
            Console.WriteLine("{0}", member);
        }

        //display the finance set
        Console.WriteLine("finance set contains {0} member(s): ", finance.Count);
        foreach (string member in finance)
        {
            Console.WriteLine("{0}", member);
        }

        //Create a new empty HashSet for everyone
        HashSet<string> everyone = new HashSet<string>(admin);

        //perform a union operation
        everyone.UnionWith(finance);

        //display the union
        Console.WriteLine("Both sets contain {0} member(s): ", everyone.Count);
        foreach (string member in everyone)
        {
            Console.WriteLine("{0}", member);
        }

        // Create a new empty HashSet for just members belong to both groups
        HashSet<string> both = new HashSet<string>(admin);

        //perform an intersect operation
        both.IntersectWith(finance);

        //display the intersect
        Console.WriteLine("Members that occur in both sets: {0}", both.Count);
        foreach (string member in both)
        {
            Console.WriteLine("{0}", member);
        }
    }
}
```

Figure 1.39: C# using sets to solve a problem

There is a particular function in this example application that can only be accessed by administrators (members of the Admin group) who work in the Finance department (members of the Finance Group). You can use the intersection of the sets to find out who this person is.

```
admin set contains 3 member(s):
John
Ahmed
 Jo
finance set contains 4 member(s):
Jo
Claire
Niamh
Phil
Both sets contain 6 member(s):
John
Ahmed
 Jo
Claire
Niamh
Phil
Members that occur in both sets: 1
 Jo
```

Figure 1.40: Sample output of set example

Looking at Figure 1.40, you can easily see that the only user who exists in both sets is Jo. This means that only Jo has access to this function.

All algorithms rely on combinations of the control structures that you have already encountered – loops, branches, function calls, etc.

Link

To learn more about some of the most common data structures (lists, arrays, records and sets), see Data structures, in *Unit 4: Software Design and Development Project*.

Common/standard algorithms

Developing programs successfully often makes use of standard types of algorithm to store and process data. Although the real-life context of a problem may be different (for example, insurance, education, retail) the techniques that are required are often generic. For example, the need to sort a list of names into alphabetical order could occur in many different business situations.

For this reason, it is a good idea to become familiar with a range of different types of algorithm that will form part of your problem-solving toolkit; that is, techniques you can rely on and apply to new situations.

Common or standard algorithm types include those for sorting, searching, counting and validation, as well as stacks and queues to implement sorting and searching.

Using stacks and queues to implement sorting and searching

Stack and queue algorithms are used to implement sorting and searching algorithms.

Stack (LIFO)

A stack is known as a **LIFO** data structure and has two basic operations: push and pull (sometimes known as pop).

Stacks are a vital part of any computer platform's operating system and are a common tool for the programmer when developing solutions.

Key term

LIFO – last in, first out – a way of describing how data is treated in some data structures; the last item of data pushed on is also the first item of data that may be pulled back off.

Stacks are conceptually viewed vertically as shown in Figure 1.41.

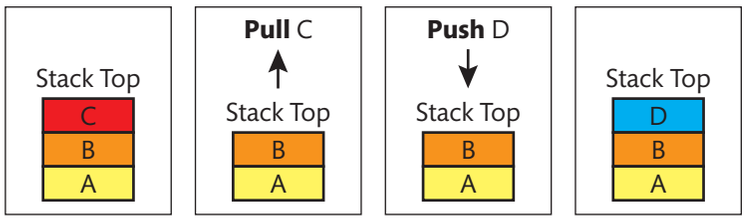


Figure 1.41: Stack operations

Figure 1.42 shows a stack operation in C#.

```
using System;
using System.Collections;

namespace stack
{
    class Program
    {
        static void Main(string[] args)
        {
            char data;                  //single item of stack data
            Stack st = new Stack();     //our stack

            //push onto stack
            st.Push('A');
            st.Push('B');
            st.Push('C');

            //show the stack
            Console.WriteLine("Current stack: ");
            foreach (char ch in st)
            {
                Console.WriteLine(ch);
            }

            //remove from stack top - it should be "C"
            data = (char)st.Pop();
            Console.WriteLine("The popped value: {0}", data);

            //push onto stack again
            st.Push('D');

            //show the changed stack
            Console.WriteLine("Current stack: ");
            foreach (char ch in st)
            {
                Console.WriteLine(ch);
            }

            //wait for keypress to continue
            Console.ReadKey();
        }
    }
}
```

Figure 1.42: Stack operation in C#

Unlike an array where elements may be accessed in any order, a stack can only be accessed in a last in, first out manner. This can be particularly useful, especially when processing recursive algorithms. Stacks are often included in a programming language's library, complete with the necessary methods to process them.

Key term

FIFO – first in, first out – a way of describing how data is treated in some data structures; the first item of data added is also the first item of data that may be removed.

Queues

In contrast, a simple queue is known as a **FIFO** data structure and has two basic operations.

FIFO

This adds (to the tail) or 'enqueues' the data and removes (from the head) or 'dequeues' data. Only data at the head of the queue can be accessed and removed.

Queues, like stacks, are a vital part of any computer platform's operating system. For example, you are likely to be familiar with the concept of a printer queue. For the programmer, they are an excellent way of managing task processing.

Queues are conceptually viewed horizontally as shown in Figure 1.43.

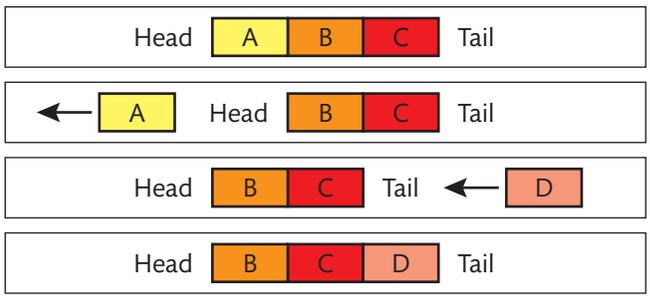


Figure 1.43: Queue operations

Figure 1.44 demonstrates the creation and access of a simple queue in C#.

```
using System;
using System.Collections;
using System.Linq;
using System.Text;

namespace queue
{
    class Program
    {
        static void Main(string[] args)
        {
            char data;              //single item of queue data
            Queue q = new Queue();  //our queue

            //add to queue
            q.Enqueue('A');
            q.Enqueue('B');
            q.Enqueue('C');

            //show the queue
            Console.WriteLine("Current queue: ");
            foreach (char ch in q)
            {
                Console.Write(ch + " ");
            }

            //remove from head - it should be 'A'
            data = (char)q.Dequeue();
            Console.WriteLine("The removed value: {0}", data);

            //add to queue again
            q.Enqueue('D');

            //show the changed queue
            Console.WriteLine("Current queue: ");
            foreach (char ch in q)
            {
                Console.Write(ch + " ");
            }

            //wait for keypress to continue
            Console.ReadKey();
        }
    }
}
```

Figure 1.44: C# simple queue

Sorting

Sorting is a basic process whereby items are ordered based on a chosen attribute, for example ordering videogames alphabetically by their title or albums in a media library by the band or artist's name.

In computer science, many different sorting algorithms exist, each offering various levels of complexity and efficiency that make some more suited to certain situations, for example sorting small or large sets of data quickly.

The three most common sorting algorithms are:

- bubble sort
- insertion sort
- quicksort.

You will now look at each sorting algorithm in turn to identify how they work and their strengths and weaknesses.

Bubble sort

The bubble sort is probably the easiest sorting algorithm to understand. As you may have guessed, its name mirrors the way that bubbles float to the surface of a liquid. In a bubble sort, a data item is physically moved through a list until it reaches its correctly sorted position. This is achieved by comparing each neighbouring pair and swapping their position as necessary. This process can require many complete '**passes**' along the data until all data items are in their correctly sorted positions.

A simple example of using a bubble sort algorithm to sort an array of five random integers into ascending order (small to large) is shown in Table 1.7.

Table 1.7: Bubble sort example

First Pass	Second Pass	Third pass	Fourth pass (0 swaps)
(9, 2, 3, 1, 6) -> (2, 9, 3, 1, 6)	(2, 3, 1, 9, 6) -> (2, 3, 1, 6, 9)	(2, 1, 3, 6, 9) -> (1, 2, 3, 6, 9)	(1, 2, 3, 6, 9) -> (1, 2, 3, 6, 9)
(2, 9, 3, 1, 6) -> (2, 3, 9, 1, 6)	(2, 3, 1, 9, 6) -> (2, 1, 3, 6, 9)	(1, 2, 3, 6, 9) -> (1, 2, 3, 6, 9)	(1, 2, 3, 6, 9) -> (1, 2, 3, 6, 9)
(2, 3, 9, 1, 6) -> (2, 3, 1, 9, 6)	(2, 1, 3, 6, 9) -> (2, 1, 3, 6, 9)	(1, 2, 3, 6, 9) -> (1, 2, 3, 6, 9)	(1, 2, 3, 6, 9) -> (1, 2, 3, 6, 9)
(2, 3, 1, 9, 6) -> (2, 3, 1, 6, 9)	(2, 1, 3, 6, 9) -> (2, 1, 3, 6, 9)	(1, 2, 3, 6, 9) -> (1, 2, 3, 6, 9)	(1, 2, 3, 6, 9) -> (1, 2, 3, 6, 9)

You should be able to see the larger values moving (bubbling) to the right and the smaller values to the left after each pass. In the simplest bubble sort algorithm, it is common for the data to become fully sorted *before* the algorithm is able to detect this. The algorithm will only stop when a full pass is completed with no swap occurring (see the fourth pass). This is usually achieved through the use of a Boolean 'swapped' flag, as shown in Figure 1.45, which shows the bubble sort algorithm written as pseudocode.

```
BEGIN
  size = length (data)
  REPEAT
   swapped = false
   FOR index = 1 TO size -1
     IF data[index + 1] < data [index] THEN
       swap data[index + 1], data[index]
       swapped = true
     ELSE
       no swap needed
     ENDIF
    ENDFOR
  UNTIL swapped = false
END
```

Figure 1.45: A bubble sort algorithm

Although various tweaks to the algorithm can improve its efficiency, bubble sorts are not the fastest sorting method and should be avoided when dealing with large data sets.

Insertion sort

This type of sorting algorithm is similar to a bubble sort as it sorts 'in-place'. However, its algorithm is very different from that of the bubble sort and involves maintaining a sub-list within the unsorted data, which is always kept sorted.

Traditionally, this is achieved in the lower part of an array with newly selected elements being inserted into their correct position. A simple example of using an insertion sort algorithm to sort an array of five random integers into ascending order (small to large) is shown in Table 1.8.

Table1.8: Tracing through an insertion sort algorithm

Action	Data (sub-list)	Decision
Unsorted array of data	(1,10,4,5,19)	
Compare element 0 and 1	(1,10,4,5,19)	No change, put 1into sorted sub-list
Compare element 1 and 2	(1,10,4,5,19)	10 is not in correct position
Swap element 1 and 2	(1,4,10,5,19)	
Check element 0 and 1 of sub-list	(1,4,10,5,19)	1 and 4 are already in the correct order, so leave sub-list alone
Compare element 2 and 3	(1,4,10,5,19)	10 is not in correct position
Swap element 2 and 3	(1,4,5,10,19)	
Check elements 0, 1 and 2 of sub-list	(1,4,5,10,19)	1, 4 and 5 are already in the correct order, so leave sub-list alone
Compare element 3 and 4	(1,4,5,10,19)	10 is in the correct position
Check elements 0,1,2 and 3 of sub-list	(1,4,5,10,19)	1, 4, 5 and 10 are already in the correct order, so leave sub-list alone
No comparison; it is the last element	(1,4,5,10,19)	Array should be sorted

Research

Modify the original data set to (1, 10, 5, 4, 19) and trace the insertion sort algorithm to see what would happen.

Figure 1.46 shows the insertion sort algorithm written as pseudocode.

```
BEGIN
  size = length (data)
  hole = 0
  insertValue = 0

  FOR index = 0 TO size -1

    insertValue = data[index]
    hole = index

    WHILE hole > 0 AND data[index - 1] > insertValue
      swap data[hole], data[hole-1]
      decrement hole
    ENDWHILE

    Data[hole] = insertValue

   ENDFOR
END
```

Figure 1.46: An insertion sort algorithm

Similarly to the bubble sort, the insertion sort algorithm is not suitable for larger data sets.

Key term

Recursion – this is an important concept in computer science and occurs when the success of solving a larger problem relies on solving smaller copies of that problem first. In programming, recursion is typically achieved when a function calls itself a number of times until a terminating condition is met and each result is returned to its parent function until the final result is attained.

Quicksort

Of the three sorting algorithms we are focusing on, the quicksort, invented by C.A.R. Hoare, is probably the most complex to understand. The reason for its complexity is that it involves a process known as **recursion**.

As the name suggests, the quicksort's main selling point is that, compared with other sorting algorithms, it is considerably faster in most instances, especially when dealing with larger data sets. This is because it uses a 'divide and conquer' approach by using two phases:

1. the **partition** phase – which works out how to divide the data into two partitions
2. the **sort** phase – which does the actual sorting of each smaller partition.

Below is the pseudocode for each phase. Figure 1.47 shows the pseudocode for the quicksort phase. Take note of the recursive call to sort each half.

```
FUNCTION quicksort (data, start, end)
BEGIN
  IF start < end
    pivot = partition(data, start, end)
    CALL quicksort (data, start, pivot - 1)
    CALL quicksort (data, pivot + 1, end)
  ENDIF
  RETURN data
END
```

Figure 1.47: Recursive quicksort phase

Figure 1.48 shows the pseudocode for the partition phase.

```
FUNCTION partition (data, start, end)
BEGIN
  pivot = data[start]
  left = start+1
  right = end
  done = false
  WHILE done == false
    WHILE left <= right AND data[left] <= pivot
      left = left + 1
    ENDWHILE
    WHILE data[right] >= pivot and right >=left
      right = right -1
    ENDWHILE
    IF right < left THEN
      done = true
    ELSE
      temp = data[left]
      data[left] = data[right]
      data[right] = temp
    ENDIF
  ENDWHILE
  temp = data[start]
  data [start] = data[right]
  data [right] = temp
  RETURN right
END
```

Figure 1.48: Partition phase

Most modern programming languages have pre-written libraries that contain standard functions for sorting data stored in arrays and lists. Despite this, having a working knowledge of the processing involved often proves invaluable when solving more complex problems.

Searching

Another basic algorithm enables data to be searched for a particular value. There are two basic types of searching algorithm: serial/linear and binary search.

Serial/linear search

A linear search of a specific value within a list of data items is a reasonably simple algorithm to understand and implement. It is perhaps one of the most frequently used algorithms in a developer's toolkit.

This algorithm needs to 'walk' along the data (usually stored in an array), compare the element against the search value and, if they are identical, return the index position where the match has been made. If all data items have been checked and no match has occurred, an error should be flagged. This process is shown in Table 1.10.

Table 1.10: Linear search with successful and unsuccessful outcomes

Search value	Data items	Linear search result
3	0 1 2 3 4 (9, 2, 3, 1, 6)	2
5	0 1 2 3 4 (9, 2, 3, 1, 6)	-99

Care should be taken not to return a 0 if no match is found; if the array is zero-based then 0 would in fact be the first element. For this reason, using a **sentinel value** (something unusual, for example –99) may be considered a good replacement. At first glance, you may have thought that using a Boolean false seems a good choice, but care should be taken, as false usually equates to 0 (which could again be a valid position depending on the array). Figure 1.49 shows the linear search algorithm written as pseudocode.

```
BEGIN
  INPUT search
  size = length (data)
  found = -99
  FOR index = 0 TO size-1
    IF data[index] = search THEN
      found = index
      BREAK
    ENDIF
  ENDFOR
  OUTPUT found
END
```

Figure 1.49: Linear search algorithm using a sentinel

Linear searches are improved by pre-sorting the data but the impact of performing a double operation (sorting then searching) should be considered. Processing problems often occur when there is a large quantity of data, especially if the search value is found towards the end of the list (or not at all, having checked every single one). An alternative to a linear search is a binary search.

Binary search

Unlike a linear search, the binary search has to work on a sorted list of data.

Once the data is successfully sorted, the algorithm picks the middle value in the list. Then it compares this value to the search value. If the search value is smaller, it discards the values above the middle value; if the search value is bigger, it discards the values below. It repeats this process with increasingly smaller blocks of data until the search value is found or established to not be in the list. This process is shown in Figure 1.50.

Search value	Unsorted data
10	0 1 2 3 4 5 6 7 8 9 (25,13,8,7,6,5,20,19,**10**,17)
	Sorted data
	0 1 2 3 4 5 6 7 8 9 (5,6,7,8,**10**,13,17,19,20,25)
	Find midpoint in data
	0 1 2 3 4 5 6 7 8 9 (5,6,7,8,**10**,13,17,19,20,25) Is 10 < 13? **Yes**, so discard 5–9 0 1 2 3 4 (5,6,7,8,**10**)
	Find midpoint in data
	0 1 2 3 4 (5,6,7,8,**10**) Is **10** < 7? **No**, so discard 0–2
	Only two are left; is our search value one of them?
	3 4 (8,**10**) **Yes**, **10** is in position **4**.

Figure 1.50: Binary search

As you can see, the binary search method has only required three comparisons in total. If you examine the original unsorted data, a linear search would have needed to perform nine comparisons to find it in location 8. Even if you had performed the linear search on the sorted data, it still would have required five comparisons to find it in position 4. It would appear, therefore, that the binary search is more efficient, although you would have to include the time taken to sort the data first before making a true comparison.

Figure 1.51 shows a simplified algorithm for this type of search.

```
BEGIN

  data = array of integers
  sortedData = Sort(data)
  size = size of sortedData
  found = false
  midPoint = 0

  INPUT search

  lowerBound = 0
  upperBound = size-1

  WHILE found = false

   IF upperBound < lowerBound THEN
     BREAK
   ENDIF

   midPoint = lowerBound + (upperBound - lowerBound) / 2

   IF sortedData[midPoint] < search THEN
     lowerBound = midPoint + 1
   ENDIF

   IF sortedData[midPoint] > search THEN
     upperBound = midPoint – 1
   ENDIF

   IF sortedData[midPoint] = search THEN
     found = true
     BREAK
   ENDIF
  ENDWHILE

  IF found = false THEN
   OUTPUT search “ was not found.”
  ELSE
   OUTPUT search “ was found in location “ midPoint
  ENDIF

END
```

Figure 1.51: Simplified binary search algorithm

Other standard algorithms

There are two other types of standard algorithm that will be looked at in this section. These are count occurrences and input validation.

Count occurrences

This is a variation of the linear search algorithm. However, rather than confirming the match, you are interested in keeping a count of how many times a specific value is found in a set of data. This counter is then output after the loop finishes.

Much of the pseudocode can be adapted from the linear search, as shown in Figure 1.52.

```
BEGIN
  INPUT search
  size = length (data)
  count = 0
  FOR index = 0 TO size-1
    IF data[index] = search THEN
      increment count
    ENDIF
  ENDFOR
  OUTPUT search " was found " count " time(s)."
END
```

Figure 1.52: Typical count occurrences algorithm

Input validation

Validation techniques vary greatly depending on the type of validation required, for example whether these are for checking the type of data being entered or whether the data entered is in a desired range. Most validation techniques use a combination of looping and comparison.

Figure 1.53 shows a typical range check algorithm written as pseudocode.

```
BEGIN
  min = 1
  max = 10
  REPEAT
    OUTPUT "Enter a value between " min " and " max
    INPUT value
    IF value < min OR value > max THEN
       OUTPUT "Value must be between " min " and " max
    ENDIF
  UNTIL value >= min AND value <= max
END
```

Figure 1.53: Typical range check algorithm

PAUSE POINT

Can you explain what the assessment outcome was about? What elements did you find easiest?

Hint

Close the book and think about the different operators, control structures, data structures, algorithms and functions that are used to build a program.

Extend

You have seen a number of different search and sort algorithms; why do you think there are different methods for achieving the same result – how do they differ?

D Types of programming and mark-up languages

Different types of problem have spawned different programming styles. Each style, known as a paradigm, aims to solve a problem in a different way, often to fulfil different user needs.

There are three main types of computer programming – procedural, object oriented and event driven – and each solves problems in a different way and therefore has different uses. When coding for the web, different issues come into play in terms of user needs and this section will also look at the type of programming and mark-up languages used to code websites. In this section, you will explore the issues surrounding the translation of code between programming languages.

Procedural programming

Procedural languages are often the first ones learned by a programmer. They are often considered to be a general-purpose tool and are used to create many different types of application.

Common procedural programming languages include:

- C
- Perl
- Python®.

Procedural programs are typically written as a series of well-defined steps which solve a set problem, for example performing a simple calculation in C, as shown in Figure 1.54.

```
#include <stdio.h>
#include <string.h>

int main()
{
  int a;
  int b;
  int c;

  puts("Enter first number");
  scanf("%d", &a);
  puts("Enter second number");
  scanf("%d", &b);

  c = a + b;
  printf("%d + %d = %d",a,b,c);

  return 0;
}
```

Figure 1.54: C program code displaying the sum of two inputted numbers

You need to be able to recognise the structure of a procedural program, identify its major components by name and state their purpose.

Figure 1.55 shows a typical program written in C, demonstrating the common procedural features that you should be able to identify, interpret and use correctly. As you will notice, various aspects of the program have been tagged with descriptions to help you to analyse the code.

Link

For more about types of programming, see *Unit 4: Software Design and Development Project*, *Unit 15: Website Development* and *Unit 17: Mobile Apps Development*.

```
#include <stdio.h>

#define MAXSIZE 5

//create a Boolean data type
typedef enum {false,true} bool;


//global data
int data[MAXSIZE];
int stackTop;

//function prototypes
bool stackFull(void);
bool stackEmpty(void);
char showMenu(void);
void pushData(void);
void popData(void);
void showStack(void);
void clearBuffer(void);

int main()
{
  //local data
  char menu;
  stackTop = -1;

  //main loop
  do
  {
    menu = showMenu();
    switch (menu)
    {
      //push chosen
      case '1' : if (!stackFull())
                 {
                   pushData();
                 }
                 else
                 {
                   puts("Sorry, stack is full.\n");
                 }
                 clearBuffer();
                 break;

      //pop chosen
      case '2' : if (!stackEmpty())
                 {
                   popData();
                 }
                 else
                 {
                   puts("Sorry, stack is empty.\n");
                 }
                 clearBuffer();
                 break;
```

Built-in libraries

Constant

Global variables

Functions and procedures

Local variables

Function call

Function calls

Conditional branches

Function calls

Conditional branch

```
      //show stack chosen
      case '3' : if (!stackEmpty())
            {
              showStack();
            }
            else
            {
              puts("Sorry, stack is empty.\n");
            }
            clearBuffer();
            break;
    }

  } while (menu != 'X' && menu !='x');

  puts("Goodbye!");

  return 0;
}

//show the stack simulator menu and get option
char showMenu (void)
{
  char option;

  puts("\n\nStack Simulator");
  puts("1 - Push data");
  puts("2 - Pop data");
  puts("3 - Show stack");
  puts("X - eXit");
  puts("Enter choice: 1,2,3 or X:");
  scanf("%c",&option);
  return option;
}

//is the stack empty?
bool stackEmpty(void)
{
  if (stackTop==-1)
  {
    return true;
  }
  else
  {
    return false;
  }
}
//is the stack full?
bool stackFull(void)
{
  if (stackTop==(MAXSIZE-1))
  {
    return true;
  }
  else
  {
    return false;
  }
}
```

Function calls

Conditional branch

Iteration

Local variable

Sequence of output statements

Input statement

Function

Conditional branches

Function

Conditional branches

```
//push data onto the stack
void pushData(void)
{
  int newData;
  puts("Enter new Data");
  scanf("%d",&newData);
  stackTop++;
  data[stackTop] = newData;
}

//pop data from the stack
void popData(void)
{
  int poppedData;
  poppedData = data[stackTop];
  printf("Popped data is %d",poppedData);
  stackTop--;

}
//display the stack
void showStack(void)
{
  puts("Stack Top");
  for (int counter=stackTop; counter>=0;counter--)
  {
    printf("%d is %d\n",counter,data[counter]);
  }
}

//clear keyboard buffer of any stray return/line feeds
void clearBuffer(void)
{
  while (getchar() != '\n');
}
```


Figure 1.55: C program using many common procedural features

As you may have noticed, the C program shown in Figure 1.57 is a full stack simulator, which is coded in a procedural fashion, suitable for execution on various computer platforms (for example Microsoft® Windows® operating system and Linux).

At this level of study, you should be able to interpret programs of this complexity (or greater), be comfortable with the challenge of identifying the different components of a procedural program and recognise how they connect to form a fully working solution.

Object-oriented programming

Object-oriented (OO) programming is a popular modern approach to programming. OO languages rely on the concepts of 'classes' and 'objects' to solve real-world problems. Due to this approach, they are a popular choice for designing videogames, e-commerce websites, database systems and user interfaces.

Common object oriented programming languages include:

- C++
- Microsoft® C#
- Oracle® JavaScript
- PHP
- Ada.

In OO, objects are created from classes, which are usually modelled on real-world 'things' such as customers, bank accounts, products, orders etc. Each class acts as a software blueprint that encapsulates (or contains) the thing's state (its data or properties) and behaviour (its functions or methods) in program code (see Figure 1.56).

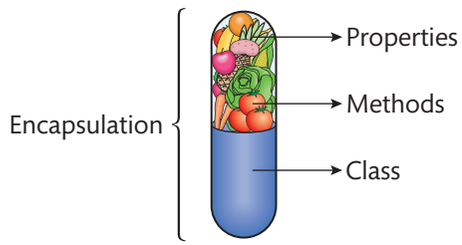


Figure 1.56: Encapsulation

The programmer creates specific interactions between different objects in order to solve the problem. As each class exists separately, they can easily be modified or adapted to reflect changes happening in the real world without having a negative effect on the whole solution. This makes OO languages very attractive to developers when they consider the demands of tackling ongoing maintenance.

You need to be able to recognise the structure of an OO program, identify its major components by name and state their purpose.

Figure 1.57 details the most common components that you will see in an OO program by re-implementing the stack simulator using C++.

```
#include <iostream>

using namespace std;

#include <stdio.h>

//constants
const int MAXSIZE = 5;

//function prototypes
char showMenu(void);
void clearBuffer(void);

class stack
{
 //hidden data (private properties)
 private:
 int data[MAXSIZE];
 int stackTop;

 public:
 //class methods
 stack() {stackTop =-1;}    //inline constructor

 bool stackFull(void);
 bool stackEmpty(void);

 void pushData(void);
 void popData(void);
 void showStack(void);

};
```

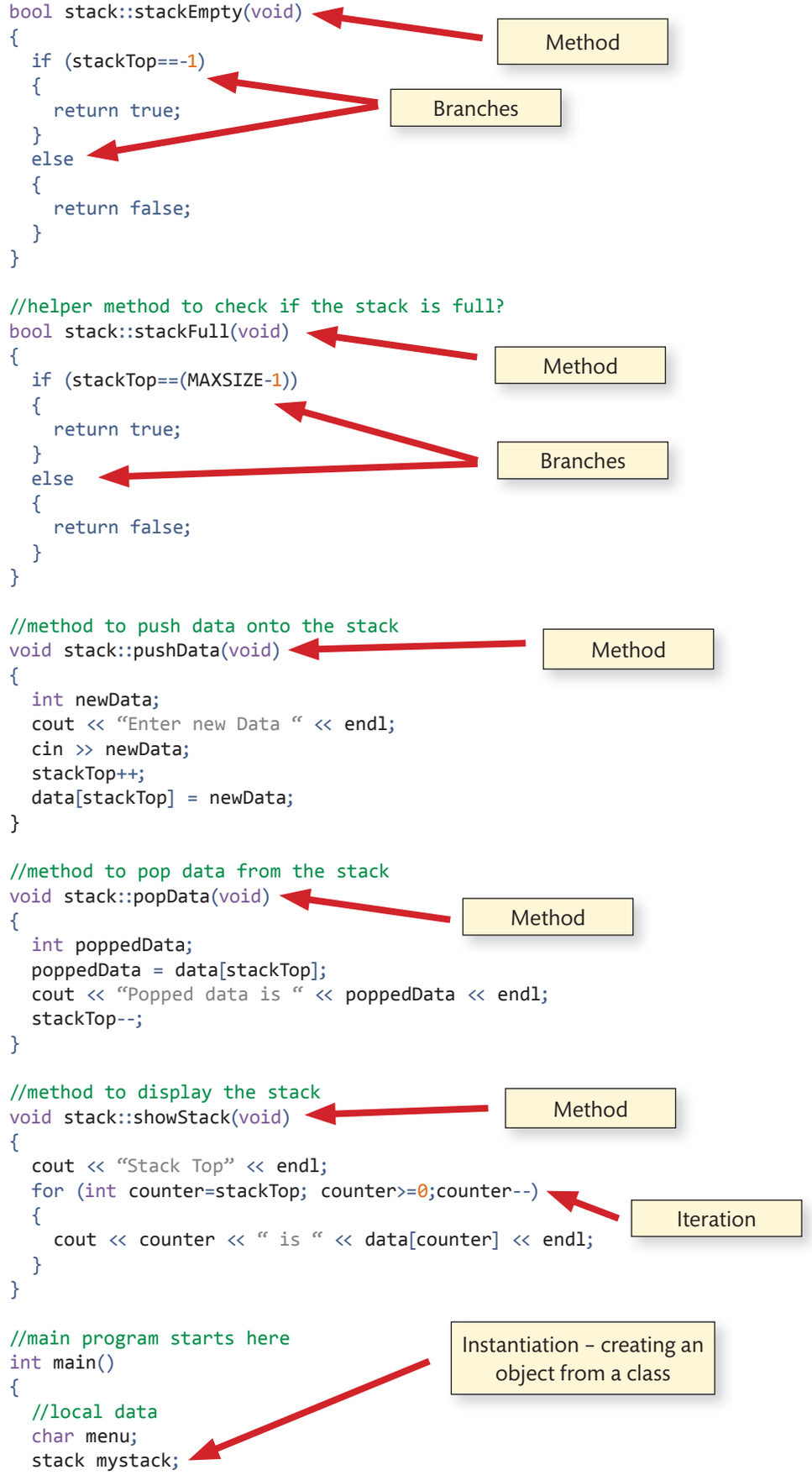
Constant

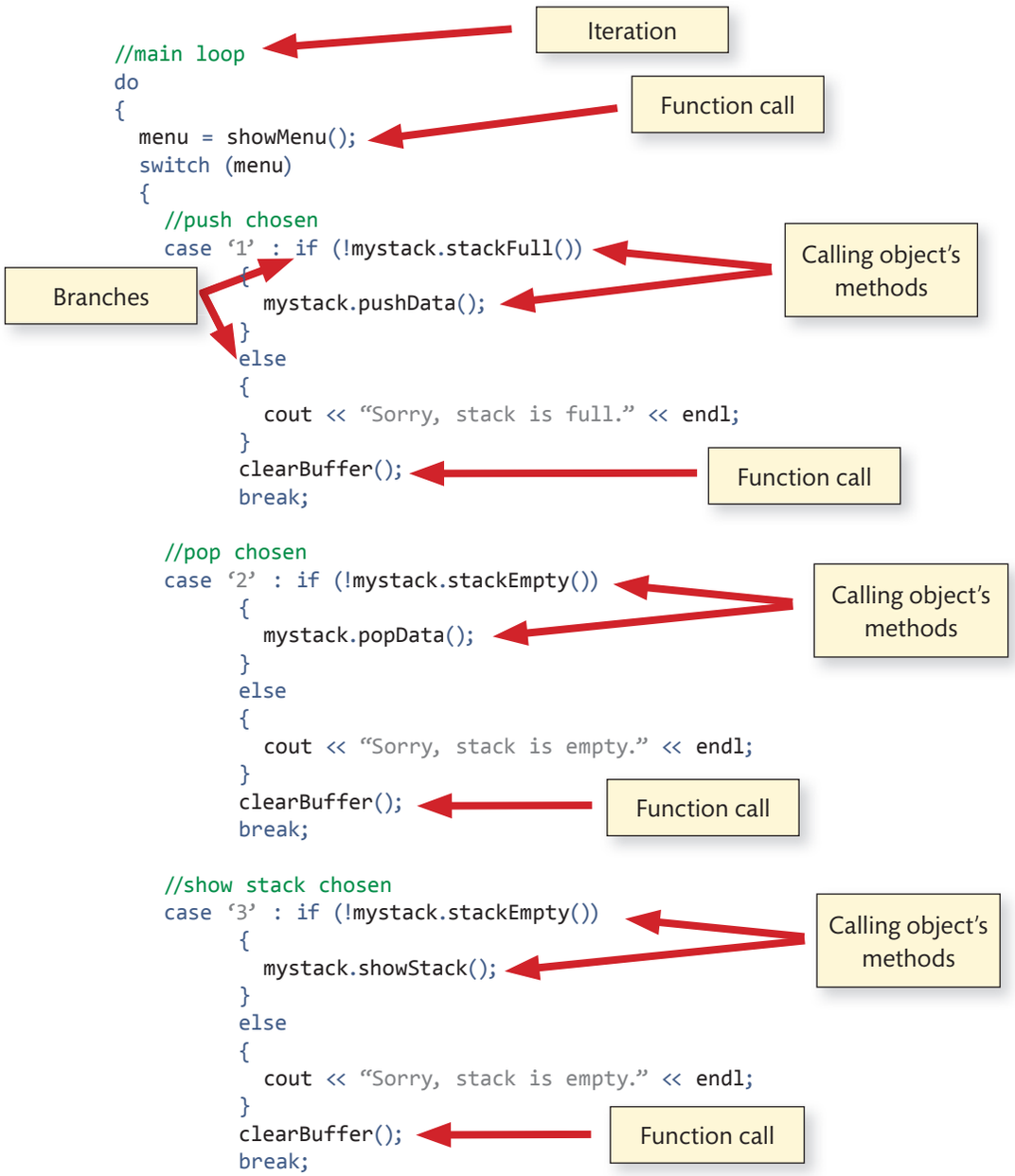
Encapsulation – a class which contains data (properties) and functions (methods)

Hidden data (properties)

Methods (functions)

```
//helper method to check if the stack is empty?
bool stack::stackEmpty(void)
{
  if (stackTop==-1)
  {
    return true;
  }
  else
  {
    return false;
  }
}

//helper method to check if the stack is full?
bool stack::stackFull(void)
{
  if (stackTop==(MAXSIZE-1))
  {
    return true;
  }
  else
  {
    return false;
  }
}

//method to push data onto the stack
void stack::pushData(void)
{
  int newData;
  cout << "Enter new Data " << endl;
  cin >> newData;
  stackTop++;
  data[stackTop] = newData;
}

//method to pop data from the stack
void stack::popData(void)
{
  int poppedData;
  poppedData = data[stackTop];
  cout << "Popped data is " << poppedData << endl;
  stackTop--;
}

//method to display the stack
void stack::showStack(void)
{
  cout << "Stack Top" << endl;
  for (int counter=stackTop; counter>=0;counter--)
  {
    cout << counter << " is " << data[counter] << endl;
  }
}

//main program starts here
int main()
{
  //local data
  char menu;
  stack mystack;
```






```
//main loop
do
{
  menu = showMenu();
  switch (menu)
  {
    //push chosen
    case '1' : if (!mystack.stackFull())
               {
                 mystack.pushData();
               }
               else
               {
                 cout << "Sorry, stack is full." << endl;
               }
               clearBuffer();
               break;

    //pop chosen
    case '2' : if (!mystack.stackEmpty())
               {
                 mystack.popData();
               }
               else
               {
                 cout << "Sorry, stack is empty." << endl;
               }
               clearBuffer();
               break;

    //show stack chosen
    case '3' : if (!mystack.stackEmpty())
               {
                 mystack.showStack();
               }
               else
               {
                 cout << "Sorry, stack is empty." << endl;
               }
               clearBuffer();
               break;
  }

} while (menu != 'X' && menu !='x');

cout << "Goodbye!" << endl;

return 0;
}

//show the stack simulator menu and get option
char showMenu (void)
{
  char option;
```

```
  cout << "\n\nStack Simulator" << endl;
  cout << "1 - Push data" << endl;
  cout << "2 - Pop data" << endl;
  cout << "3 - Show stack" << endl;
  cout << "X - eXit" << endl;
  cout << "Enter choice: 1,2,3 or X:" << endl;
  cin >> option;
  return option;
}

//clear keyboard buffer of any stray return/line feeds
void clearBuffer(void)
{
  while (getchar() != '\n');
}
```

Input and Output statements

Iteration

Figure 1.57: C++ program using many common OO features

Although this example includes many OO features, some are not required in this solution. These include the following.

- **Inheritance:** This is the ability to create derived (or child) classes from a base class in order to reuse existing code (saving development time) and help to extend existing solutions with new functionality.
- **Polymorphism:** This involves using OO concepts, such as overloading, where multiple methods with the same function name exist but each accepts arguments of different data types or a different quantity of arguments. This allows the programmer to use just one named function to perform a set task but alter its behaviour depending on its use.

Again, at this level of study, you should be able to interpret programs of this complexity (or greater), be comfortable with the challenge of identifying the different components of OO programming and recognise how they connect to form a fully working solution. The key to OO programming is the potential reuse and extensibility of the program code to create amended and new solutions.

Key term

Polymorphism – this comes from the Greek word, meaning 'many forms' and is a core feature of the object-oriented (OO) programming paradigm. In C++, common uses of polymorphism include the 'overloading' of functions and operators, thus changing their use (but not their name) depending on the context in which they are used.

Event-driven programming

This is a popular paradigm for the development of graphical applications which respond to events generated either by the system (for example a system clock) or the user (for example a mouse click).

Event-driven programs typically work non-sequentially, with users able to select the operations they want to perform rather than follow the preset inputs of a more rigid program structure.

Developers focus on programming event handlers, which is the code that specifies the actions to perform when a particular event is triggered via a listener. A listener is a process that waits for a certain event to happen. For example, if a File->Open menu option is clicked, a file open dialog will be displayed.

Common event-driven programming languages include:

- Microsoft® Visual Basic .Net
- Microsoft® Visual C++
- Microsoft® Visual C#
- Oracle® JavaScript

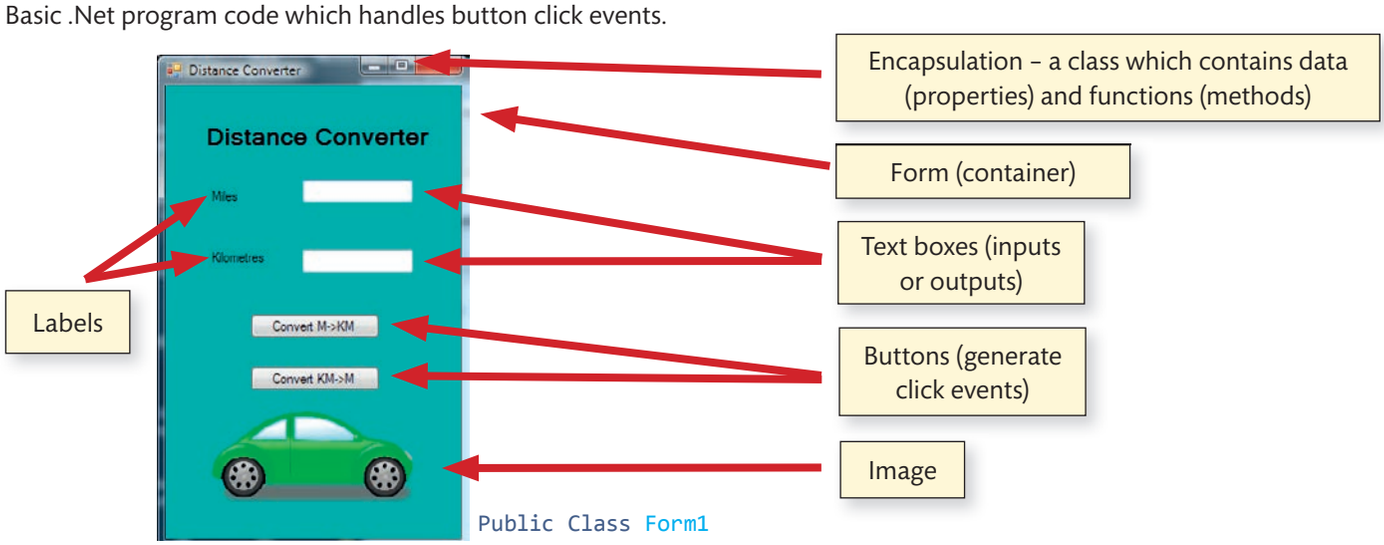
You may notice that some programming languages, such as Microsoft® Visual C++ and C#, offer features also common in object-oriented (OO) programming paradigms; this is not unusual – many modern programming languages are actually hybrids of multiple paradigms.

Event-driven programs work by operating in a **main loop**. This loop is effectively **listening** for different **events** to occur. Some events are **user generated**, for example clicking a button, and others are **system generated**, for example low RAM.

When the event is **triggered**, a suitable **event handler** (a specially written sub-routine called a **callback** is executed which responds to the event. The following simple example is created using Microsoft's® Visual Basic .Net. It uses a form, as shown in Figure 1.58, to provide a user interface and event handlers to process click events that are triggered from the form's buttons. Figure 1.58 also shows the Microsoft® Visual Basic .Net program code which handles button click events.

Link

Event handling is a core aspect of creating mobile applications, as shown in *Unit 17: Mobile Apps Development*.




```
Public Class Form1

    'constant (some rounding errors)
    Const MtoKM As Decimal = 1.61
    Const KMtoM As Decimal = 0.62

    'variables
    Dim miles As Decimal
    Dim kilometres As Decimal

    'event handler M->KM button
    Private Sub Button1_Click(sender As System.Object, e As System.EventArgs) \
    Handles Button1.Click

      miles = Val(TextBox1.Text)
      kilometres = miles * MtoKM
      TextBox2.Text = Str(kilometres)
    End Sub

    'event handler KM->M button
    Private Sub Button2_Click(sender As System.Object, e As System.EventArgs) \
    Handles Button2.Click

      kilometres = Val(TextBox2.Text)
      miles = kilometres * KMtoM
      TextBox1.Text = Str(miles)
    End Sub

End Class
```

Constants

Variables

Event handlers

Calculations forming the core actions of the event handler

Figure 1.58: Microsoft® Visual Basic .Net form and program code which handles button click events

Key terms

Service oriented processing – this is the breaking down of complex problems into a collection of separate (but potentially linked) processes, each providing a specific service for client applications.

Time driven processing – this is a form of event-driven programming whereby each process is triggered by a time-based event, typically using a computer's real-time clock (RTC).

Although this example includes many event-driven features, some are not required in this solution. These include **service oriented processing** and **time driven processing**.

Again, at this level of study, you should be able to interpret programs of this complexity (or greater), be comfortable with the challenge of identifying the different components of an event-driven programming and recognise how they connect to form a fully working solution.

The key to event-driven programming is realising that, unlike creating a procedural solution that leads the user through a sequence of actions, an event-driven program simply responds to different events that the user or computer may trigger.

Coding for the web

In this section, you will learn how to interpret, analyse and evaluate the use of code written for web languages.

The characteristics, features and implications of mark-up and web languages

Mark-up is a form of language used to specify the content (but not formatting) of a document in a structured manner using special tags. For example, in HTML, <p> tags are used to denote the start and end of a paragraph:

```
<p>A new paragraph</p>
```

Other mark-up languages can be used to represent complex data structures in a platform-neutral manner, such as entries from a CD library stored in XML, as shown in Figure 1.59.

```
<CD>
 <TITLE>Let's dance</TITLE>
 <ARTIST>Bowie, David</ARTIST>
 <YEAR>1983</YEAR>
</CD>
<CD>
 <TITLE>Rubber Soul</TITLE>
 <ARTIST>The Beatles</ARTIST>
 <YEAR>1965</YEAR>
</CD>
```

Figure 1.59: XML data

XML is often used as the preferred format to transfer data between different computer systems and applications, such as when exporting and importing data between relational database systems that are normally incompatible.

Because HTML focuses on the content of a web page, modern website development uses a separate technology called cascading style sheets (CSS) to add formatting, for example to change the colour of text or specify a font size or horizontal alignment.

For example, if you add **inline CSS** to the previous HTML example, the CSS formatted paragraph would now appear like this:

```
<p style="font-family:arial;font-size:32px; color:red">A new paragraph</p>
```

the CSS formatted paragraph would now appear like this:

A new paragraph

Web technologies such as HTML5 and CSS3 (the latest standards) can be used to create wonderful web page designs with attractive layouts and engaging content.

It is the job of the application's web browser to render the HTML and CSS on screen for the user; programmers tend to call this 'static' content as each page has been created manually by a web developer.

After completing your studies, you should be capable of writing and interpreting HTML5 and CSS3 with relative ease.

Link

For more detailed information on mark-up and style sheet technologies, see *Unit 15: Website Development*.

Uses, applications and implications of mark-up and web languages

HTML and CSS render very quickly on modern devices and, because there is a recognised world-wide standard, most web browsers render these pages very similarly to give users a consistent experience.

Free web browsers are available for a multitude of computer platforms, from powerful PCs running Microsoft® Windows to mobile phones running Android. This means that HTML and CSS achieve a form of platform independence: with appropriate internet connectivity, any web page created in standard HTML and CSS can be viewed on any device – anywhere in the world. This fact alone makes web page development technologies exceptionally powerful.

In addition, although standard web pages use a protocol called HTTP (hypertext transfer protocol) to transfer data between a web server and a web browser client, a secure version called HTTPS (hypertext transfer protocol secure) can be used to communicate sensitive data, such as passwords and electronic payment information.

The development and refinement of web-based internet services has fundamentally changed the way in which people communicate socially, shop, work, learn, listen to music, watch television and even do their banking. HTML, CSS and other web languages are at the very heart of these changes.

Although HTML5 adds extra interactive possibilities to a web page, other web technologies have been providing dynamic content for many years. These scripting technologies are said to be either client side or server side.

Uses, applications and implications of client-side processing and scripting

Client-side processing is achieved through the use of event-aware scripting languages such as JavaScript and VBScript™. JavaScript, which may be part of a HTML page or stored in a separate .JS file, is freely viewable once downloaded. Structurally, it is similar to many languages in the C family, as shown in Figure 1.60.

```
<script>
  var message = "Hello";
  var length = message.length;
  alert("Your string "+ message + " is " + length + " character(s) long");
</script>
```

Figure 1.60: JavaScript pop-up message

These client-side scripting languages are executed from within the web browser and are used to add event-based interactivity to a web page. For example, they can:

- validate user's input into HTML data forms, e.g. postcodes and telephone numbers
- perform bespoke animations
- bring up pop-up message boxes
- provide search boxes

- perform spell checking
  - provide auto-completion of forms
  - enable real-time updates to a page, e.g. breaking news or stock ticker
  - provide partial page updates without reloading, using a technology called AJAX (asynchronous JavaScript and XML).

Although JavaScript can be used irresponsibly, generally this is not the case. However, because of this possibility, JavaScript support can be switched off in a web browser client's settings. If this is done, even accidentally, it is possible that a coded solution using the web as its platform could be disrupted.

Comprehensive JavaScript libraries such as jQuert and AngularJS are used commercially to provide powerful built-in functions that make client-side development more flexible and productive.

Uses, applications and implications of server-side processing and scripting

In contrast to client-side processing, there are various scripting languages that are executed on the web server and which generate HTML, CSS (and even JavaScript) to send to the web server client.

Popular server-side scripting technologies include:

- PHP (PHP Hypertext pre-processor)
- Microsoft® ASP .Net
- JavaScript Server Pages (JSP)
- Lua
- Python®
- Ruby
- Perl.

Because the server-side script code is executed on the server, it is never viewable on the client's web browser, making it much more secure for commercial use. Server-side scripting languages often have considerable interaction with the server's operating system and can perform many complex tasks including:

- encryption
- database connectivity, querying, manipulation and management
- email generation
- file creation and management
- user authentication
- HTML form processing.

Server-side scripting languages, such as PHP, are also structurally similar to the C family, as shown in Figure 1.61.

```
<?php
$message = "Hello";
$length = strlen($message);

print "<p>Your string $message is $length character(s) long</p>";
?>
```

Figure 1.61: PHP server-side script

Once this server-side code is executed on the web server and sent to the client, only the underlying HTML code remains:

```
p>Your string Hello is 5 character(s) long</p>
```

Server-side scripting is a very powerful tool and is at the heart of any online retail site, as its ability to process search requests, query stock and customer databases and generate HTML quickly and painlessly makes it an ideal instrument for implementing code solutions on a web platform. The only major disadvantage is that its translation and execution add extra processing load onto the web server.

Issues and implications of implementing code on a web platform

Implementing a code solution on a web platform has a number of advantages and disadvantages.

Advantages

- Code solution runs inside a web platform so may not have full access to the underlying operating system or hardware, making it secure.
- Easy to integrate media such as images, video and music.
- Easy to link to other resources, e.g. spreadsheets, documents, electronic slideshows.
- Code does not need to be installed.
- Code will always be the latest version as updated on the server.
- Usage of the code solution may be easier to monitor.
- Potentially good integration with other online services, e.g. user support, email, instant messaging and social networking.

Disadvantages

- The HTML, CSS and JavaScript is usually viewable in the web browser client, so is not secure and can be reverse-engineered and copied relatively easily.
- Requires a client web browser.
- Code running inside a web browser can be less efficient and slower than code running natively on an operating system.
- An internet connection is probably necessary.
- Security and access may become an issue.
- Requires hosting which is usually not free.
- A target for hackers and website defacement.
- Not all web browser clients support web technology standards equally well.
- Overall functionality can be limited when forced to work within a web browser environment.

Translation

Sometimes it is necessary to translate a coded solution into another programming language. This process is known as 'porting'. A practical example of this could be converting an older procedural C program into a new OO C++ solution.

Reasons for translating code from one language to another

Why translate code from one language to another?

- Programming language can no longer meet the changing needs of the solution.
- Support for the existing programming language is ending.

- Existing programming language has identified security flaws.
- Change of hardware precludes continued development with existing language.
- Change of business/industry/sector policy regarding preferred programming language.
- Modern programming trends, e.g. newer programming languages.
- Availability of developers with suitable experience.

Benefits of translating code from one language to another

What are the benefits of translating code from one language to another?

- New solution uses more modern programming language.
- New solution is more extensible, easier to update and expand.
- Development time is reduced and changes can be made more quickly.
- Costs are reduced.
- Developer support is improved.
- New language features assist creation of new functionality for the solution.

Drawbacks of translating code from one language to another

What are the drawbacks of translating code from one language to another?

- New solution may be inferior to the original; compromises affecting performance may be made as exact translation is not possible.
- Solution may not be as efficient.
- Solution may prove to be overly time consuming to port to new language.
- New programming language is overrated; although some improvements may be possible, other drawbacks with the new language may be discovered.
- Developers for new programming language may be scarce and therefore expensive to hire and/or train.
- Less community knowledge about new programming language to help with bugs.

The implications and impact of translating code

Changing the programming language which creates a solution will always have effects.

The implications and impact of doing this on users, organisations and developers will now be considered.

Users

- Changes their perception of the program's performance – may be better or worse.
- The program may no longer have the same functionality.
- Lack of bug-fixes and maintenance on the old solution while the new solution is being created.

Organisations

- Purchase of new development environments, new programming language and new support materials.
- Potential hiring of new programmers or training of existing programmers.
- Developing a new solution costs money so there will be a need to justify this.
- Review of the new programmed solution – was it worthwhile?

Developers

- Learning of new skills.
- Personal and professional challenge – learning and developing with a new programming language.

- Dealing with the expectations of the users and the organisations.
- Challenges when debugging a new programming language.
- Finding the most efficient solutions using a new programming language.

Alternative ways to implement current code base

If translating a solution to a new programming is not practicable, it may be necessary to consider other ways of implementing the current code base. This could include:

- exploring new versions of the programming language
- using improved third-party function libraries
- investigating faster compilers or interpreters to improve performance
- identifying performance bottlenecks in the solution and creating improved algorithms to make the code base more efficient
- asking newer developers to view the code to see if they can recommend improvements and fresh ideas
- Whatever the programming paradigm being used, the ability to debug programs, identify errors and confidently fix them remains absolutely essential.

Research

Many opportunities exist to research and explore different programming languages online through the use of virtual development environments that let the visitor select a particular language, key some basic program code and execute it. To access a typical website with this functionality go to:

http://www.tutorialspoint.com/codingground.htm

PAUSE POINT	Can you explain what the assessment outcome was about? What elements did you find easiest?
Hint	Close the book and think about the different programming paradigms and their key features.
Extend	You have seen a number of popular programming paradigms; which approach do you think is the most prevalent in the software development industry and why do you think this is so?

Assessment practice 1.2

A04 A05

Read through the following scenario and, using a range of programming paradigms and different types of programming language, create solutions for the following problem.

A program is needed that generates a results table for a local youth football league. The league currently has 20 teams and they play each other, home and away, over the period of 8 months from August through to March.

In common with other leagues, a win is awarded 3 points and a draw is awarded 1 point. The top team is shown in gold and the next two in green. All other teams are shown in black, apart from the bottom three, which risk relegation; these are shown in red.

The program should track the number of games played, won, drawn, lost, goals for, goals against, goal difference and the current number of points.

The user should be able to update this table by selecting two teams using their current table positions and then entering the result of the match. Invalid table positions and nonsensical results, e.g. teams scoring negative or very large quantities of goals should not be allowed. In addition, teams should only be able to play each other twice per season (home and away).

Once this is completed, the table should be automatically updated and displayed in an attractive manner.

Plan

- Do I fully understand the nature of the problem?
- Am I clear about what I am being asked to do?
- Do I know how to handle the data needed in this program?
- Do I know how to structure the data needed in this program?
- Can I identify which operators, built-in functions, validation methods, control structures and algorithms may be required to solve this problem?
- Am I able to create versions of this solution using different types of programming language and compare their effectiveness and suitability?

Do

- I know what it is that I am doing and what I want to achieve.
- I can use a variety of programming paradigms to create a working program.
- I can differentiate different types of programming language and can make effective judgements about their suitability when implementing a solution.
- I start from the beginning of the scenario and work through to the end.

Review

- I can explain different programming paradigms and their importance to the solution of a problem.
- I am able to create solutions using a variety of programming languages.
- I am able to analyse and evaluate their characteristics, uses, issues and implications.
- I can explain how I would approach the hard elements differently next time (i.e. what I would do differently).

Further reading and resources

Books

Johansen A – *Python®, The Ultimate Beginner's Guide* (CreateSpace Independent Publishing Platform, 2016)

Flanagan D – *JavaScript: The Definitive Guide* (O'Reilly Media, 2011)

Nixon R – *Learning PHP, MySQL & JavaScript* (O'Reilly Media, 2014)

Schildt H – *C++: A Beginner's Guide*, *Second Edition* (McGraw-Hill, 2003)

Schildt H – *C# 4.0 The Complete Reference* (McGraw Hill, 2010)

Moore K – *Karl Moore's Visual Basic .NET* (Apress, 2002)

Websites

Coding Ground – Tutorials Point: **http://www.tutorialspoint.com/codingground.htm**

Microsoft® API and Reference Catalog: **https://msdn.microsoft.com/library**

World Wide Web Consortium: **https://www.w3.org/**

THINK FUTURE

Dan Hardy

Junior Programmer in a software development team

I've worked in the team for just over a year. In that time, I've worked on various programs and I've already learned so much. Some of the programming tasks we tackle are very complex – I've found that I can't just sit at my keyboard and start writing program code without thinking about the solution properly. When I first started, my manager encouraged me to draw flowcharts to break down difficult business logic into simple steps. I find this approach really works, and I end up returning to my flowcharts a lot while I'm working on the next part of the problem.

When I started, I only knew Microsoft® C#. I was worried when I was asked to develop in other languages like JavaScript and PHP. After playing around with these languages, though, I discovered they had similar syntax to C#, and many of the C# concepts I knew were easy to translate. I guess you could say I started to identify the repeating patterns!

When I have to describe to my friends what programming is like, I tell them that it's mainly about problem solving. I spend a lot of my time refining complex problems into simpler ones and trying to spot any patterns in the problems I deal with.

Focusing your skills

Being adaptable

There are many different programming languages used in computing and the industry is constantly evolving. It is really important that you keep your skills and knowledge as up to date as possible. To get a position as a junior developer in a software development team, you have to be able to adapt yourself and your skills to meet market needs and to keep up with current programming trends.

Here are some questions to ask yourself to help you to do this.

- Which programming languages are in demand right now?
- What kinds of skills are sought-after? Is there a particular interest in specific operating systems or specific types of application?
- Are there any other skills related to programming that are in demand, such as knowledge of web technologies or relational databases?
- What experience of and exposure to different programming languages is expected of a junior developer? What skills are mentioned in software development job descriptions and vacancies?
- Can I transfer my programming skills to new languages and different development environments?

Getting ready for assessment

This section has been written to help you do your best when you take the assessment test. Read through it carefully and ask your tutor if there is anything that you are still not sure about.

About the test

This unit is assessed under supervised conditions. The number of marks for the unit is 90. Pearson sets and marks the test.

The external assessment will last for two hours. During the supervised assessment period, learners will be assessed on their ability to apply their computational thinking skills to solve problems.

Sitting the test

Listen to and read carefully, any instructions you are given. Lots of marks are often lost through not reading questions properly and misunderstanding what the question is asking.

Arrive in good time so that you are not in a panic.

The test should be carried out under supervised conditions.

- You may use a calculator.
- Internet access is not permitted.
- You must not bring anything into the supervised environment or take anything out without your tutor's knowledge and approval.

Avoid any distractions from a mobile phone!

Make sure that you arrive in good time for each test session and check that you have everything you need for the test ahead of time.

Remember that you cannot lose marks for a wrong answer, but you cannot gain any marks for a blank space!

Ensure that you have checked all sides of the assessment task before starting.

Ensure that you leave yourself enough time at the end to check through your work. Proofread and correct any mistakes before handing in your work.

Command words typically used in assessment

Most questions contain command words. Understanding what these words mean will help you to understand what the question is asking you to do.

Command word	Definition – what it is asking you to do
Analyse	Examine in detail a scenario or problem to discover its meaning or essential features. Learners will need to break down the problem into its parts and show how they interrelate. There is no requirement for any conclusion.
Calculate	Apply some form of mathematical or computational process.
Complete	Complete a diagram or process. This can be applied to problems/solutions of varying complexity.
Demonstrate	Illustrate and explain how an identified computer system or process functions. This may take the form of a piece of extended writing, a diagram or a combination of the two.
Describe	Give an account of something or highlight a number of key features of a given topic. This may also be used in relation to the stages of a process.
Develop	Provide a solution to a problem, typically using an existing system or structure that must be improved or refined.
Discuss	Investigate a problem or scenario, showing reasoning or argument.
Draw	Represent understanding through the use of a diagram or flowchart.
Evaluate	Review and synthesise information to provide a supported judgement about the topic or problem. Typically, a conclusion will be required.
Explain	Make a series of linked points and/or justify or expand on an identified point.
Identify	Assess factual information, typically, when making use of given stimuli. Requires a single word or short sentence answer.
Produce	Provide a solution that applies established constructs to a given computing problem.
State, name, give	Assess factual information. Requires a single word or short sentence answer.
Write	Produce a solution, or a mechanism used as part of a solution, to a given computing problem.

Writing long answers

Work out which questions you need to answer and then organise your time, based on the marks available for each question. Set yourself a timetable for working through the examination and then stick to it – do not spend ages on a short 1–2 mark question and then find you only have a few minutes for a longer 7–8 mark question.

If you are writing a longer answer, try to plan out your answer before you start writing. Have a clear idea of the point your answer is making, and make sure that this comes across in everything you write, so that it is all focused on answering the question.

Try answering all the simpler questions first then come back to the harder questions. This should give your more time for the harder questions.

If you finish early, use the time to reread your answers and make any corrections – this could really help to make your answers even better.

Sample answers

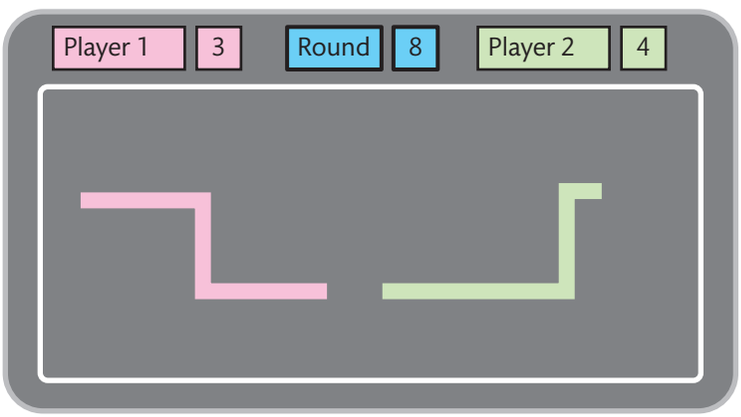
For some of the questions, you will be given some background information on which the questions are based.

Look at the sample questions which follow and our tips on how to answer these well.

Worked Example

Matteusz has been asked to create a simple 2D 'light cycle' game called 'Rainbow Line Warriors' which allows two players to race against each other and try to force the other to crash. Player 1 (red) starts on the left-hand side of the screen and is initially moving right. In contrast, player 2 (green) starts on the right-hand side of the screen and is initially moving left. Both players can move in four directions (up, down, left and right) using a different set of keys to control their cycle. The game awards 1 point for each victory, with the winner being the first to reach an agreed number of rounds.

A design for the starting screen is shown below.



Question 1

Identify three aspects of the game that would be represented as variables.

1.

2.

3.

3 marks

Sample answer:

1. Player 1 score
2. Player 2 score
3. Number of rounds

These answers are correct. The program would need to store both users' scores and the round number as variables. In this example, the easy answers are identifiable from the visual design; this will not always be the case. For example, what are the other variables involved? We could also consider the following:

Player 1 X position

Player 1 Y position

Player 2 X position

Player 2 Y position

Player 1 direction (Up, Down, Left or Right)

Player 2 direction (Up, Down, Left or Right)

Any of these answers would also be worth one mark as they demonstrate that you can analyse a scenario and discover its essential features.

Question 2

Matteusz knows that the 'tip' of each light cycle will be stored using x and y coordinates.

Player 1 (red) will use the following keys to control their light cycle: W – up, D – right, S – down and A – left. Matteusz also knows that the screen's origin (0, 0) is located top left.

Matteusz has written the following pseudocode that describes the movement of player 1's red light cycle, depending on the keys being pressed. He has used a separate function called 'CHECKCOLLISION' to deal with crashes. Unfortunately, he has made some mistakes and has not written it very well.

Rewrite the pseudo code to correct the errors.

```
INPUT PLAYER1KEY
IF PLAYER1KEY = W THEN
PLAYER1X = PLAYER1X - 1
CALL CHECKCOLLISION
ELSE
IF PLAYER1KEY = D THEN
PLAYER1X = PLAYER1X + 1
ELSE
IF PLAYER1KEY = S THEN
PLAYER1X = PLAYER1Y + 1
CALL CHECKCOLLISION
ENDIF
ENDIF
ENDIF
```

5 marks

Sample answer

```
INPUT PLAYER1KEY

IF PLAYER1KEY = W THEN
   PLAYER1Y = PLAYER1Y - 1
   CALL CHECKCOLLISION
ELSE
   IF PLAYER1KEY = D THEN
    PLAYER1X = PLAYER1X + 1
   ELSE
     IF PLAYER1KEY = S THEN
      PLAYER1Y = PLAYER1Y + 1
      CALL CHECKCOLLISION
     ELSE
       IF PLAYER1KEY = A THEN
        PLAYER1X = PLAYER1X - 1
        CALL CHECKCOLLISION
       ENDIF
     ENDIF
   ENDIF
ENDIF
```

This answer has identified and fixed most of the issues in Matteusz's pseudocode.

- Indentation has been added to highlight the structure of the IF…ELSE branches.
- New code has been added to deal with the A (moving left) key press which was originally missing.
- Code has been corrected to adjust the *y* ordinate, not *x*, when pressing W and moving up.
- Code has been corrected to adjust the actions when pressing S (PLAYER1Y should be adjusted, not PLAYER1X).

However, key press 'D' (moving right) still does not call the CHECKCOLLISION function – this has still not been identified. An improved answer should try to identify and correct everything.

Question 3

Matteusz's game is written using procedural programming techniques.

Explain how the structure of an object-oriented language would be used to manage the data and functions in the program.

4 marks

Sample answer:

A light cycle could be encapsulated as a class.

Each light cycle class could contain:

- Data/Properties such as
  - *x* position
  - *y* position
  - colour
  - score
  - player's name
- Functions/Methods such as
  - checkCollision
  - moveLightCycle
  - crashLightCycle
  - updateScore

The answer is acceptable. It has correctly identified a potential class, sensible properties and potential methods. Each of these is likely to be awarded a mark.

However, it could be improved by stating that two objects would be instantiated from the light cycle class for each player: red and green.