AQA Approved

AQA A-level

Includes AS and A-level

Computer Science

Bob Reeves

DYNAMIC LEARNING

HODDER EDUCATION
LEARN MORE

AQA
A-level

Computer Science

Includes AS and A-level

Bob Reeves

Approval message from AQA

The core content of this digital textbook has been approved by AQA for use with our qualification. This means that we have checked that it broadly covers the specification and that we are satisfied with the overall quality. We have also approved the printed version of this book. We do not however check or approve any links of any functionality. Full details of our approval process can be found on our website.

We approve print and digital textbooks because we know how important it is for teachers and students to have the right resources to support their teaching and learning. However, the publisher is ultimately responsible for the editorial control and quality of this digital book.

Please note that when teaching the ***AQA A-level Computer Science*** course, you must refer to AQA's specification as your definitive source of information. While this digital book has been written to match the specification, it cannot provide complete coverage of every aspect of the course.

A wide range of other useful resources can be found on the relevant subject pages of our website: www.aqa.org.uk

DYNAMIC LEARNING
HODDER EDUCATION
AN HACHETTE UK COMPANY

The Publishers would like to thank the following for permission to reproduce copyright material:

P.11 © chombosan - Fotolia.com; **P.24** © VvoeVale - iStock via Thinkstock.com; **P.69** Courtesy of Wikipedia, The Opte Project (http://creativecommons.org/licenses/by/2.5/); **P.111** © Hodder & Stoughton; **P.136** *middle* © Sergey Kamshylin - Fotolia.com, *bottom* © mark huls - Fotolia.com; **P.137** © Jenny Thompson - Fotolia.com; **P.142** screenshot from TRANSYT from TRL Software (trlsoftware.co.uk); **P.214** © ra3rn - Fotolia.com; **P.217** © davemhuntphoto - Fotolia.com; **P.218** © Bob Reeves; **P.231** *top* © TheVectorminator - Fotolia.com, *bottom* © R+R - Fotolia.com; **P.267** *top* © Maksym Yemelyanov - Fotolia.com, *bottom* © finallast -Fotolia.com; **P.271** © KarSol - Fotolia.com; **P.289** © Igor Mojzes - Fotolia.com; **P.295** Courtesy of Wikimedia Commons, author Ordercrazy, Creative Commons CC 1.0 (http://creativecommons.org/publicdomain/zero/1.0/deed.en); **P.313** © Maxim Pavlov - Fotolia.com

Every effort has been made to trace all copyright holders, but if any have been inadvertently overlooked the Publishers will be pleased to make the necessary arrangements at the first opportunity.

Although every effort has been made to ensure that website addresses are correct at time of going to press, Hodder Education cannot be held responsible for the content of any website mentioned. It is sometimes possible to find a relocated web page by typing in the address of the home page for a website in the URL window of your browser.

Hachette UK's policy is to use papers that are natural, renewable and recyclable products and made from wood grown in sustainable forests. The logging and manufacturing processes are expected to conform to the environmental regulations of the country of origin.

Orders: please contact Bookpoint Ltd, 130 Milton Park, Abingdon, Oxon OX14 4SB.
Telephone: (44) 01235 827720. Fax: (44) 01235 400454. Lines are open 9.00–17.00, Monday to Saturday, with a 24-hour message answering service. Visit our website at www.hoddereducation.co.uk

First published in 2015 by
Hodder Education
An Hachette UK Company,
Carmelite House
50 Victoria Embankment
London EC4Y 0DZ
www.hoddereducation.co.uk

Impression number 5 4 3 2 1

Year 2019 2018 2017 2016 2015

Cover photo © LaCozza – Fotolia

A catalogue record for this title is available from the British Library

ISBN 978 1 447 183951 1

Introduction

What is computer science?

The world of computer science continues to develop at an amazing rate. If you had spoken to an A-level student embarking on a computer science course just ten years ago they might not have believed that in the year 2015 we would all be permanently connected to the Internet on smart phones, watching movies in high definition on 55-inch curved-screen TVs, streaming our favourite music to our phones from a database of millions of tracks stored in 'the cloud' or carrying round a tablet that has more processing power than the flight computer on the now decommissioned space shuttle.

No-one really knows where the next ten years will take us. The challenge for you as a computer scientist is to be able to respond to this ever-changing world and to develop the knowledge and skills that will help you to understand technology that hasn't yet been invented!

Studying A-level computer science gives you a solid foundation in the underlying principles of computing, for example: understanding how algorithms and computer code are written; how data are stored; how data are transmitted around networks; and how hardware and software work. It also provides you with a deeper level of understanding that goes beyond the actual technology. For example, you will learn about how to use computation to solve problems and about the close links between computer science, mathematics and physics.

You might be surprised to learn that many of the key principles of computing were developed before the modern computer, with some concepts going back to the ancient Greeks. At the same time, you will be learning about the latest methods for solving computable problems in today's world and developing your own solutions in the form of programs or apps.

Studying computer science at A level is challenging, but it is also highly rewarding. There are very few jobs that do not involve the use of computers and having a good understanding of the science behind them will effectively prepare you for further study or employment.

Course coverage and how to use this book

This book has been written to provide complete coverage of the AQA Computer Science specifications for AS and A level that are taught from September 2015. The content of the book is matched and sequenced according to the specification, and organised into sections in accordance with the main specification headings used by AQA.

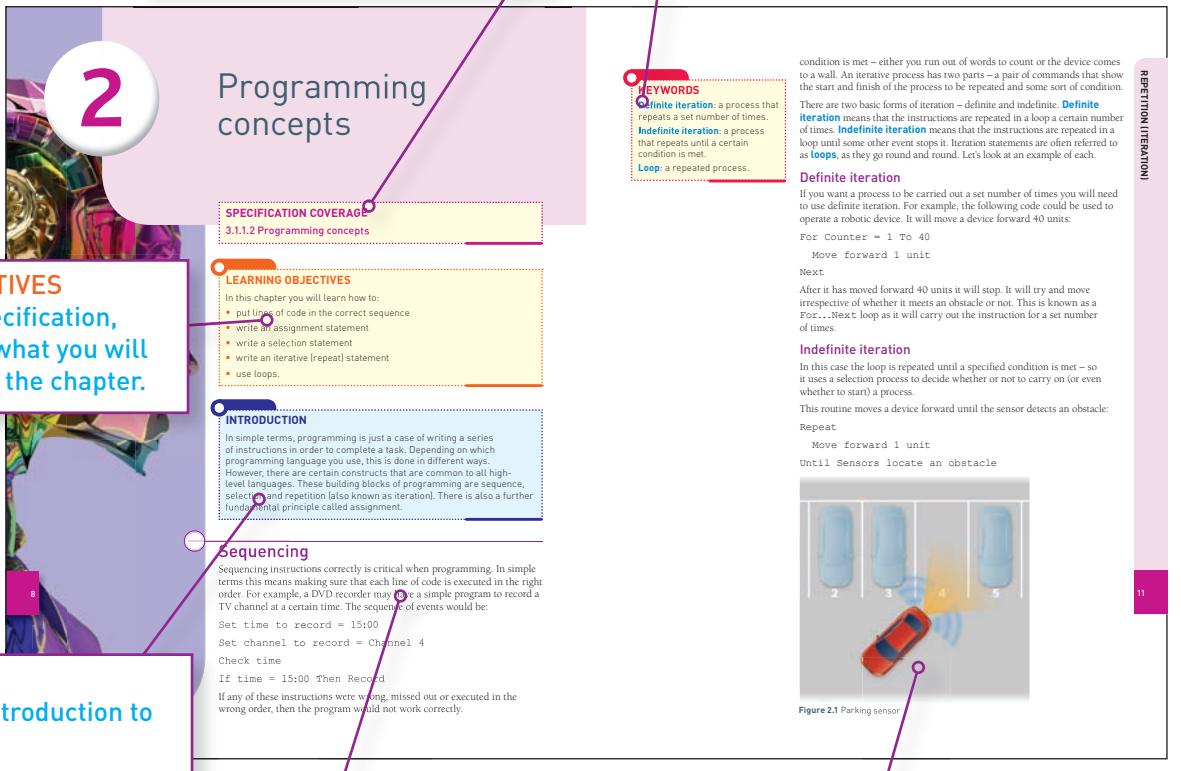
Students studying A level need to be familiar with all of the content of the AS specification and in addition need to cover those sections highlighted throughout the text and are flagged up as A level only. There is support for every section of the specification including the written papers and coursework element.

The main objective of the book is to provide a solid foundation in the theoretical aspects of the course. Further support and practical examples of coded solutions are provided on line via Dynamic Learning.

Chapters contain:

Specification coverage
Taken directly from the specification, it shows which elements of AS and A level are covered within each chapter.

KEYWORDS
All of the keywords are identified with concise definitions. These form a glossary, which is useful for revision and to check understanding.



LEARNING OBJECTIVES
Matched to the specification, these summarise what you will learn by the end of the chapter.

INTRODUCTION
This is a concise introduction to set the scene.

The main text
This contains detailed definitions, explanations and examples.

Diagrams and images
The book uses diagrams and images wherever possible to aid understanding of the key points.

Acknowledgements

Dave Fogg for producing the VB code examples used.

Matthew Walker for producing the Python code example used.

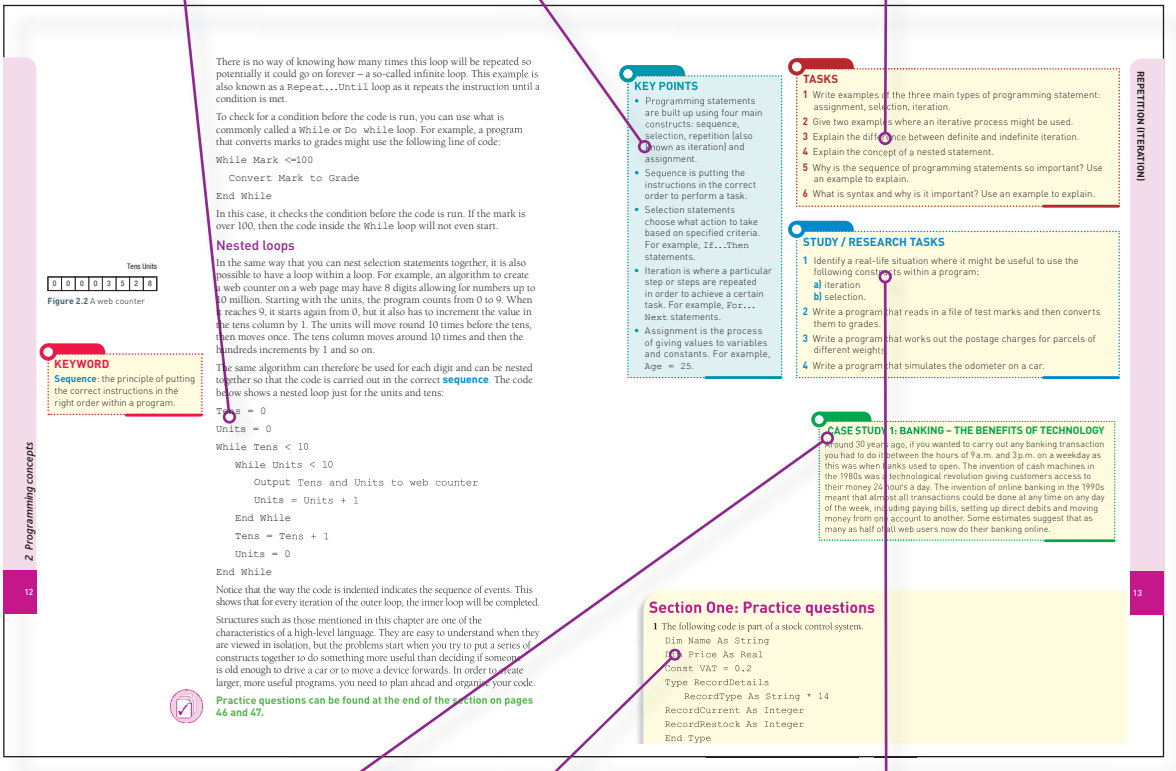
Paul Varey for his initial proofread.

Dedicated to Tommy and Eli.

Code examples
Where relevant there are examples of pseudo-code or actual code to demonstrate particular concepts. Code examples in this book are mainly written using the Visual Basic framework. Visual Basic 2010 Express has been used as this is available as a free download. The code can also be migrated into other versions of VB. Note that code that is longer than one line in the book is shown with an underscore (_). It should be input as one line in VB.

KEY POINTS
All of the main points for each chapter are summarised. These are particularly useful as a revision aid.

TASKS
These are activities designed to test your understanding of the contents of the chapter. These may be written exercises or computer tasks.



2 Programming concepts

12

There is no way of knowing how many times this loop will be repeated so potentially it could go on forever – a so-called infinite loop. This example is also known as a `Repeat...Until` loop as it repeats the instruction until a condition is met.

To check for a condition before the code is run, you can use what is commonly called a `While` or `Do while` loop. For example, a program that converts marks to grades might use the following line of code:

```
While Mark <=100
   Convert Mark to Grade
End While
```

In this case, it checks the condition before the code is run. If the mark is over 100, then the code inside the `While` loop will not even start.

Nested loops

In the same way that you can nest selection statements together, it is also possible to have a loop within a loop. For example, an algorithm to create a web counter on a web page may have 8 digits allowing for numbers up to 10 million. Starting with the units, the program counts from 0 to 9. When it reaches 9, it starts again from 0, but it also has to increment the value in the tens column by 1. The units will move round 10 times before the tens, then moves once. The tens column moves around 10 times and then the hundreds increments by 1 and so on.

Tens Units

0 0 0 0 3 5 2 8

Figure 2.2 A web counter

KEYWORD
Sequence: the principle of putting the correct instructions in the right order within a program.

The same algorithm can therefore be used for each digit and can be nested together so that the code is carried out in the correct **sequence**. The code below shows a nested loop just for the units and tens:

```
Tens = 0
Units = 0
While Tens < 10
   While Units < 10
      Output Tens and Units to web counter
      Units = Units + 1
   End While
   Tens = Tens + 1
   Units = 0
End While
```

Notice that the way the code is indented indicates the sequence of events. This shows that for every iteration of the outer loop, the inner loop will be completed.

Structures such as those mentioned in this chapter are one of the characteristics of a high-level language. They are easy to understand when they are viewed in isolation, but the problems start when you try to put a series of constructs together to do something more useful than deciding if someone is old enough to drive a car or to move a device forwards. In order to create larger, more useful programs, you need to plan ahead and organise your code.

Practice questions can be found at the end of the section on pages 46 and 47.

KEY POINTS

- Programming statements are built up using four main constructs: sequence, selection, repetition (also known as iteration) and assignment.
- Sequence is putting the instructions in the correct order to perform a task.
- Selection statements choose what action to take based on specified criteria. For example, `If...Then` statements.
- Iteration is where a particular step or steps are repeated in order to achieve a certain task. For example, `For... Next` statements.
- Assignment is the process of giving values to variables and constants. For example, `Age = 25`.

TASKS

1 Write examples of the three main types of programming statement: assignment, selection, iteration.
2 Give two examples where an iterative process might be used.
3 Explain the difference between definite and indefinite iteration.
4 Explain the concept of a nested statement.
5 Why is the sequence of programming statements so important? Use an example to explain.
6 What is syntax and why is it important? Use an example to explain.

STUDY / RESEARCH TASKS

1 Identify a real-life situation where it might be useful to use the following constructs within a program:
a) iteration
b) selection.
2 Write a program that reads in a file of test marks and then converts them to grades.
3 Write a program that works out the postage charges for parcels of different weights.
4 Write a program that simulates the odometer on a car.

CASE STUDY 1: BANKING – THE BENEFITS OF TECHNOLOGY

Around 30 years ago, if you wanted to carry out any banking transaction you had to do it between the hours of 9a.m. and 3p.m. on a weekday as this was when banks used to open. The invention of cash machines in the 1980s was a technological revolution giving customers access to their money 24 hours a day. The invention of online banking in the 1990s meant that almost all transactions could be done at any time on any day of the week, including paying bills, setting up direct debits and moving money from one account to another. Some estimates suggest that as many as half of all web users now do their banking online.

Section One: Practice questions

1 The following code is part of a stock control system.

```
Dim Name As String
Dim Price As Real
Const VAT = 0.2
Type RecordDetails
    RecordType As String * 14
RecordCurrent As Integer
RecordRestock As Integer
End Type
```

REPETITION (ITERATION)

13

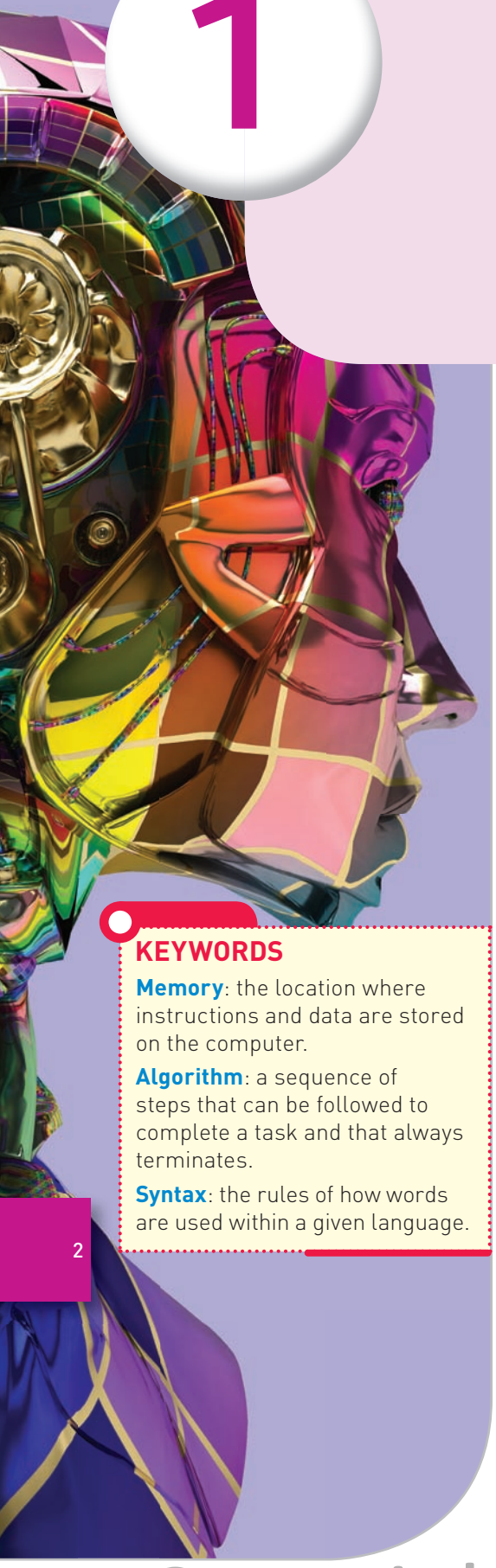
CASE STUDY
These provide real-life examples of the applications of computing.

Practice questions
These are revision questions designed to check understanding of the topics covered across the whole section.

STUDY/RESEARCH TASKS
These questions go beyond the specification and provide a further challenge designed to encourage you to 'read around the subject' or develop your skills and knowledge further.

Section One: Fundamentals of programming

1 Programming basics



SPECIFICATION COVERAGE

3.1.1.1 Data types

3.1.1.2 Programming concepts

3.1.1.6 Constants and variables in a programming language

3.5.1.2 Integers

LEARNING OBJECTIVES

In this chapter you will learn:

- the basic principles of writing instructions in the form of programming code
- what constants and variables are and how to use them
- what the main data types are
- how to store data using meaningful names.

KEYWORDS

Memory: the location where instructions and data are stored on the computer.

Algorithm: a sequence of steps that can be followed to complete a task and that always terminates.

Syntax: the rules of how words are used within a given language.

INTRODUCTION

In its simplest form a computer program can be seen as a list of instructions that a computer has to work through in a logical sequence in order to carry out a specific task. The instructions that make up a program are all stored in **memory** on the computer along with the data that is needed to make the program work.

Programs (also known as applications or apps) are created by writing lines of code to carry out algorithms. An **algorithm** is the steps required to perform a particular task and the programming code contains the actual instructions written in a programming language. This language is in itself an application that has been written by someone else to enable you to write your own programs.

In the same way that there are lots of different languages you can learn to speak, there are also lots of programming languages, and in the same way that some languages have many different dialects, there are also different versions of some of the more popular programming languages.

Another similarity with natural languages is that each programming language has its own vocabulary and rules that define how the words must be put together. These rules are known as the **syntax** of the language. The difference between learning a foreign (natural) language and a computer language is that there are far fewer words to learn in a computer language but the rules are much more rigid.

Naming and storing data

In addition to instructions, the computer program also needs data to work with. For example, to add two numbers together requires an `add` instruction and then the two numbers that need to be added. You need to give these two data items names so that the computer will know which data to use.

KEYWORD

Memory address: a specific location in memory where instructions or data are stored.

The data are stored in memory along with the instructions. You could view memory rather like a series of pigeon-holes, each having a unique address, known as a **memory address**.

It is a really good idea to use names that indicate the purpose of the data – in the case of the example above the two numbers might be called `Number1` and `Number2`. Using meaningful names will help you when they are trying to trace bugs and it also allows other programmers to follow the code more easily. It is good practice to adopt a common naming convention. In this case the first character in upper case and the rest in lower case.

This process of giving data values is called 'assigning', and it looks something like these two:

```
Number1 ← 23
```

```
Name ← "Derek"
```

The ← means 'becomes' or 'equals'. `Number1` is an example of a variable. In the example above it has been given a value of 23, though this value will change while the program is being run. `Name` is another example of a variable and has been given the value `"Derek"`.

KEYWORD

Assignment: the process of giving a value to a variable or constant.

Different programming languages have slightly different ways of assigning values. For example, you may need to use the equals sign to make the **assignment** in the code you are writing. So a simple algorithm to add two numbers together might look like this:

```
Number1 = 2
```

```
Number2 = 3
```

```
Answer = Number1 + Number2
```

Figure 1.1 shows a simplified visualisation of how this program is handled. There will be millions of memory addresses, of which just three are shown in this diagram.

Memory address	1000	1001	1002
Variable	Number1	Number2	Answer
Data	2	3	5

Figure 1.1

Constants and variables

KEYWORDS

Constant: an item of data whose value does not change.

Variable: an item of data whose value could change while the program is being run.

Data are stored either as **constants** or as **variables**. Constants (as you'd expect from the name) have values that are fixed for the duration of a program. For example, if you were writing a program that converted miles into kilometres you could set the conversion rate as a constant because it will never change. In this case we could call the constant `ConvertMilestoKm` and assign it a value of 1.6 as there are approximately 1.6 km to the mile. Then whenever we want to convert a distance in miles to its metric equivalent we would multiply it by the constant `ConvertMilestoKm`.

Notice that the name given to the constant is self-explanatory. We could have just called it `Constant1`. However, by giving it a meaningful name it makes the code easier to work with as the program gets bigger. It also makes it easier for anyone else that looks at the code, to work out what the program is doing. This is important for three main reasons:

- It makes it easier to find and correct errors/bugs in code. This is called **debugging**.
- There are many occasions where there are several programmers working on the same program at the same time, so having a sensible naming convention makes it easier for everyone to understand.
- It will be easier to update the code later on when further versions of the program are created.

KEYWORD

Debug: the process of finding and correcting errors in programs.

The value of variables can change as a program is being run. For example, the same conversion program will require the user to type in the number of miles they want to convert. This number will probably be different each time the user enters data. Therefore, you need to have a variable that you could call `NumberOfMiles`.

There are lots of other examples – the number of answers a pupil has got right in a test would (hopefully) increase as they work their way through a test so the data would have to be stored as a variable. The password a user uses to access a network can be changed at any time, so it would also be classed as a variable.

Variable and constant declaration

Declaring a constant or variable means that when you are writing code you describe (or declare) the variables and constants that you are going to use before you actually use them in your program.

Some programming languages force you to declare the variables and constants you intend to use in your program before you start writing any code. The benefits of doing this are that it forces you to plan first and the computer will quickly identify variables it does not recognise.

KEYWORD

Declaration: the process of defining variables and constants in terms of their name and data type.

There are two parts to a **declaration**. You need to supply a suitable name for the constant/variable and you need to specify the data type that will be used. The declarations might look something like this:

```
Dimension Age As Integer

Dimension Name As String

Dimension WearsGlasses As Boolean
```

`Dimension` or `Dim` is one of the command words used in Visual Basic to indicate that a variable is being declared. Once you have declared a variable it starts with a default value. In the above examples `Age` will start as zero, `Name` as nothing (also known as the empty string) and `WearsGlasses` will start with the value False. Other languages may use different default values so it is good practice to assign an initial value to the variable just to make sure it is correct.

Data types

It is important to consider how you want your program to handle data. For example, to create the miles to kilometres conversion program, you have to tell the program that miles and kilometres both need to be stored as numbers.

KEYWORD

Data type: determines what sort of data are being stored and how it will be handled by the program.

There are lots of **data types** you might need to use and you need to think carefully about the best type to use. For example, if storing numbers, how accurate do you need the number to be? Will a whole number be accurate enough or will you need decimals? In addition to numbers, you will probably want to store other data such as a person's name, their date of birth or their gender.

All programming languages offer a range of data types but the actual name of the data type may vary from language to language. Here are some of the most common data types:

KEYWORD

Integer: any whole positive or negative number including zero.

- Integer: This is the mathematical name for any positive or negative whole number. This might be used to store the number of cars sold in a particular month or the number of pupils in a class. The range of numbers that can be stored depends on how much memory is allocated. For example, an **integer** in Visual Basic can store numbers between –2 147 483 648 through to +2 147 483 647.
  Declaring a number as an integer means that the program will then handle the data accordingly. For example, 2 + 3 will equal 5. In some languages, if you did not set it to integer 2 + 3 would equal 23 (two three).
- Real/Float: This is a number that has a fractional or decimal part, for example 3.5 or $3\frac{1}{2}$. In our miles to kilometres conversion program, you would need to store both miles and kilometres using this data type as the user might want to convert a number that is not a whole number. Other examples might include a person's height in metres or their weight in kilograms.
- Text/String: This data type is used to store characters, which could be text or numbers. For example, you could use this to store a person's name or address. Some programming languages refer to this data type as alphanumeric because you can actually store any character you want in a string whilst text implies it can only store letters. Text or string variables are normally shown in quotation marks. For example you might assign the name Frank to a variable like this: `Name ← "Frank"`. House numbers and phone numbers are often stored as text / string as although they are numbers, you would never need to carry out any calculations on them and in the case of telephone numbers the leading zero is important and would be omitted if stored as a number.
- Boolean: The simplest data type is a simple yes/no or true/false. This is called a Boolean data type. It is named after George Boole who discovered the principles behind logic statements. Boolean data types can be used to store any kind of data where there are two possible values.
- Character: This data type allows you to store an individual character, which might be a letter, number or symbol. All computers have a defined character set, which is the range of characters that it understands. This would commonly be all the upper and lower case letters, plus other keyboard characters and any special characters.
- Date/Time: This will store data in a format that is easily identifiable as a date or time, e.g. 30.04.2014 or 12:30. The program will then handle the data accordingly. For example, if you added 5 to the date, it would tell you the date in five days' time. 30.05.2014 + 5 would become 04.06.2015. If you did not declare it as a date you may get the wrong answer, for example 30.05.2019.
- Pointer/Reference: This data type is used to store a value that will point to or reference a location in the memory of the computer. If you think of memory

KEYWORD

Pointer: a data item that identifies a particular element in a data structure – normally the front or rear.

as a series of pigeon-holes or addresses where instructions and data are stored, the **pointer**/reference is used in a program to go to a specific address. For example, you could set up a pointer called `Pointer1` and put address 1001 in it. The program would then go to memory address 1001 and take the data from it. In the example below it would be the data assigned to `Number2`. Other lines of code will then be needed to tell the program what to do with the data it finds there.
Figure 1.2 shows how a pointer is used to reference an item of data.

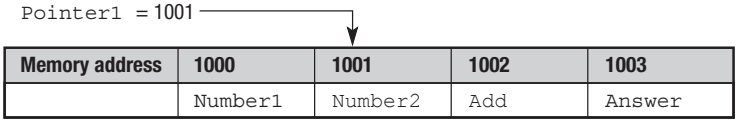


Figure 1.2

KEYWORDS

Array: a set of related data items stored under a single identifier. Can work on one or more dimensions.

Element: an single value within a set or list – also called a member.

Record: one line of a text file.

Brown
Hussain
Koening
Schmidt
Torvill
West

Figure 1.3

- Array: An **array** is a collection of data items of the same type. For example, if you wanted to store a collection of names in a school register, you could call this `Register` and each item of data would be stored as text. Each individual name in the array is called an **element**. Every element is numbered so that `Register(2)` would be the second person in the array, `Register(4)` the fourth person and so on. Note that 0 is often used as the first element of an array, rather than 1. If this was the case then `Register(2)` would actually be the third person in the array, `Register(4)` the fifth and so on.

  Figure 1.3 shows a simple array with six elements. `Register(2) = Hussain`, `Register(4) = Schmidt` (assuming array indexing starts at 1).

- Records: This is used to store a collection of related data items, where the items all have different data types. For example, you might set up a **record** called `Book`, which is used to store the title, author name and ISBN of a book. `Title` and `Author` are text whereas the `PublicationDate` is set as a Date data type.

  You could write it like this:

```
Book = Record
   Title, Author As Text * 50
   ISBN As Text * 13
   PublicationDate As Date
```

  When the program is run, every time data are entered for the book, the user will type in up to 50 characters of text for the title and author and then the ISBN. A variable could now be set up using this record data type and this variable would contain all of this data.

Built-in and user-defined data types

Built-in data types are those that are provided with the programming language that you are using. The list of built-in types varies from language to language, but all will include versions of the types listed above.

Most programming languages allow users to make up their own data types, usually by combining existing data types together. These are simply called user-defined data types. For example, if you were making a program to store user names and IDs, you may create a user-defined data type called `Logon` made up of a set number of characters and numbers.

```
Type Logon
   UserName As String * 10
   UserID As Integer * 5
End Type
```

This code will set a new data type called `Logon`, which will be made up of a 10-character user name followed by a 5-digit user ID. In total, the data type will have 15 characters/digits to store the data.

The reasons for creating user-defined types are mainly to do with efficiency. As you start to write your own programs you will find that they can get very long and complex and that debugging can be very time-consuming. Most programmers try to make their code as organised and efficient as possible as this will save them time as the program develops. For example, it is easier to reuse a block of code rather than have to write it all over again.

Most programmers aim to create code that is 'elegant'. This means that it does exactly what it is supposed to do as efficiently as possible. Often this means writing as few lines of code as possible with no repeated coding.

Using a user-defined data type is just one example of where it is possible to be more efficient. With our example of storing `Logon` information using a user-defined variable, because all the data are stored in one variable rather than two, when the program needs this information, we only need to access one variable rather than two.

Practice questions can be found at the end of the section on pages 46 and 47.

KEY POINTS

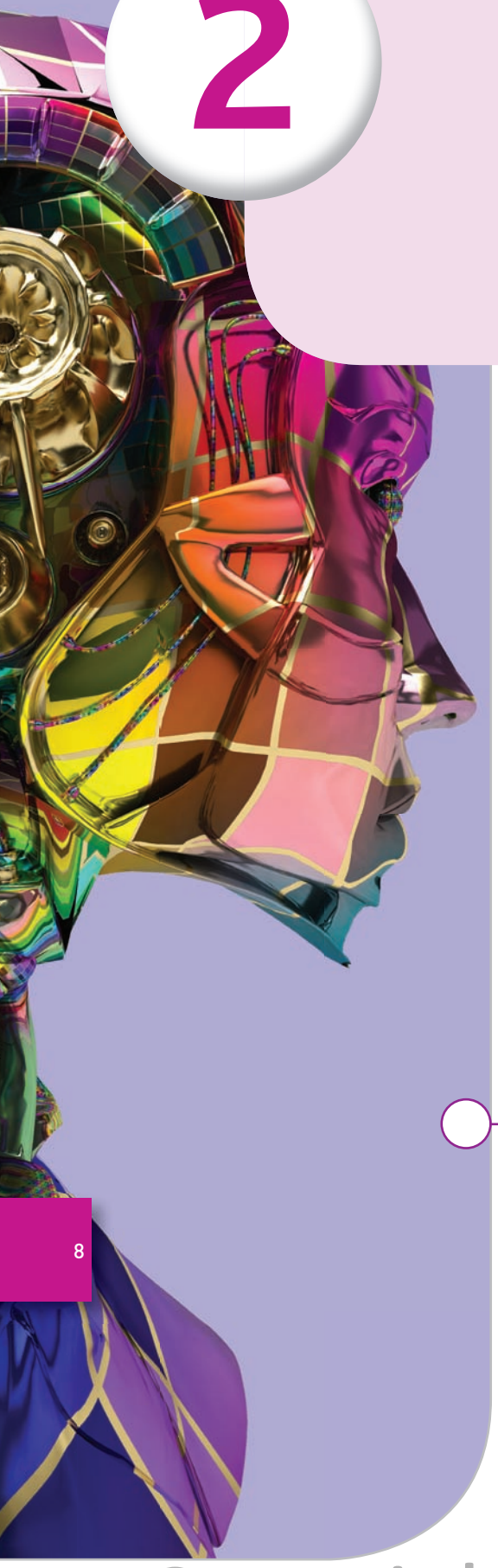
- Programming languages are used to write applications (apps).
- An algorithm is a sequence of instructions that can be followed to complete a task. Algorithms always terminate.
- Programming code is made up of algorithms that are implemented within a programming language.
- Instructions are stored in memory along with the data required by the program.
- Data are stored and named according to certain conventions.
- Variables and constants are used to store data and must be declared in some languages.
- There are several data types built in to every programming language and the programmer can also define their own.

TASKS

1. Give two reasons why it is a good idea to use meaningful variable names.
2. Use examples to explain the difference between a constant and a variable.
3. Why is it important to declare all variables and constants at the beginning of a program?
4. Explain the difference between a value and a variable.
5. Suggest suitable data types and variable names for:
   a) the current rate of VAT
   b) today's date
   c) the total takings from a shop
   d) a person's date of birth
   e) which wrist a person wears a watch on.

STUDY / RESEARCH TASKS

1. A list of data is also known as a one-dimensional array. Find out what two- and three-dimensional arrays are and give examples of where you might use each.
2. Identify the built-in data types for the main programming language that is used in your school or college.
3. Research data types that are specifically used to store sound and video data. How do they differ from other data types?



2 Programming concepts

SPECIFICATION COVERAGE

3.1.1.2 Programming concepts

LEARNING OBJECTIVES

In this chapter you will learn how to:

- put lines of code in the correct sequence
- write an assignment statement
- write a selection statement
- write an iterative (repeat) statement
- use loops.

INTRODUCTION

In simple terms, programming is just a case of writing a series of instructions in order to complete a task. Depending on which programming language you use, this is done in different ways. However, there are certain constructs that are common to all high-level languages. These building blocks of programming are sequence, selection and repetition (also known as iteration). There is also a further fundamental principle called assignment.

Sequencing

Sequencing instructions correctly is critical when programming. In simple terms this means making sure that each line of code is executed in the right order. For example, a DVD recorder may have a simple program to record a TV channel at a certain time. The sequence of events would be:

```
Set time to record = 15:00
Set channel to record = Channel 4
Check time
If time = 15:00 Then Record
```

If any of these instructions were wrong, missed out or executed in the wrong order, then the program would not work correctly.

KEYWORD

Syntax: the rules of how words are used within a given language.

The actual process of writing statements varies from one programming language to another. This is because all languages use different **syntax**. Common usage of the word syntax refers to the way that sentences are structured to create well-formed sentences. For example, the sentence 'Birds south fly in the winter' is syntactically incorrect because the verb needs to come after the noun. Programming languages work in the same way and have certain rules that programmers need to stick to otherwise the code will not work.

Assignment

We met the concept of an assignment statement in Chapter 1. Assignment gives a value to a variable or constant. For example you might be using a variable called `Age` so the code:

```
Age ← 34
```

will set the variable `Age` to have the value `34`.

The value stored in the variable could change as the program is run. For example, a computer game might use a variable called `Score`. At the beginning of the game the value is set to 0. Each time the player scores a point, the assignment process takes place again to reset the value of `Score` to 1 and so on.

Assigning values will take place over and over again while a program is being run. Initially, the programmer will assign a value to the variable. Then as the program runs, the algorithms in the program code will calculate and then return (re-assign) the latest value. Assignments are the fundamental building blocks of any computer program because they define the data the program is going to be using.

Selection

KEYWORD

Selection: the principle of choosing what action to take based on certain criteria.

The **selection** process allows a computer to compare values and then decide what course of action to take. For example you might want your program to decide if someone is old enough to drive a car. The selection process for this might look something like this:

```
If Age < 17 Then
    Output = "Not old enough to drive"
Else
    Output = "Old enough to drive"
End If
```

In this case, the computer is making a decision based on the value of the variable `Age`. If the value of `Age` is less than 17 it will output the text string `"Not old enough to drive"`. For any other age it will output the text string `"Old enough to drive"`. The `If` statement is a very common construct. In this case it is used to tell the program what to do if the statement is true using the `If...Then` construct. If the statement is false, it uses the `Else` part of the code. This is a very simple selection statement with only two outcomes.

KEYWORD

Nesting: placing one set of instructions within another set of instructions.

Nested selection

You can carry out more complex selections by using a **nested** statement. For example, a program could be written to work out how much to charge to send parcels of different weights. This could be achieved using the following sequence of selection statements:

```
If Weight >= 2000 Then
  Price = £10
Else If Weight >= 1500 Then
  Price = £7.50
Else If Weight >= 1000 Then
  Price = £5
Else
  Price = £2.50
End If
```

When the weight is input, it works through the lines of code in the `If` statement and returns the correct value. For example, if the parcel weighs 1700 g it will cost £7.50 as it is between 1500 g and 1999 g. If it weighed 2000 g or more, the `If` statement would return £10.

In some languages complex selections can be implemented using constructs such as this `Case` statement. The following example shows a section of code that allows the user to type in a country code to identify where a parcel is being sent to:

```
Select Case ParcelDestination
  Case 1
      WriteLine ("Mainland UK")
  Case 2
      WriteLine ("Europe")
  Case 3
      WriteLine ("USA")
  Case Else
      WriteLine ("Rest of the World")
End Select
```

This routine takes the value of the variable `ParcelDestination` and compares it against the different criteria. So if `ParcelDestination` is `1` then `Mainland UK` will be printed to the screen.

KEYWORD

Iteration: the principle of repeating processes.

Repetition (Iteration)

It is useful to be able to repeat a process in a program. This is usually called **iteration**. For example you might want to count the number of words in a block of text or you may want to keep a device moving forward until it reaches a wall. Both these routines involve repeating something until a

KEYWORDS

Definite iteration: a process that repeats a set number of times.

Indefinite iteration: a process that repeats until a certain condition is met.

Loop: a repeated process.

condition is met – either you run out of words to count or the device comes to a wall. An iterative process has two parts – a pair of commands that show the start and finish of the process to be repeated and some sort of condition.

There are two basic forms of iteration – definite and indefinite. **Definite iteration** means that the instructions are repeated in a loop a certain number of times. **Indefinite iteration** means that the instructions are repeated in a loop until some other event stops it. Iteration statements are often referred to as **loops**, as they go round and round. Let's look at an example of each.

Definite iteration

If you want a process to be carried out a set number of times you will need to use definite iteration. For example, the following code could be used to operate a robotic device. It will move a device forward 40 units:

```
For Counter = 1 To 40
  Move forward 1 unit
Next
```

After it has moved forward 40 units it will stop. It will try and move irrespective of whether it meets an obstacle or not. This is known as a `For...Next` loop as it will carry out the instruction for a set number of times.

Indefinite iteration

In this case the loop is repeated until a specified condition is met – so it uses a selection process to decide whether or not to carry on (or even whether to start) a process.

This routine moves a device forward until the sensor detects an obstacle:

```
Repeat
  Move forward 1 unit
Until Sensors locate an obstacle
```

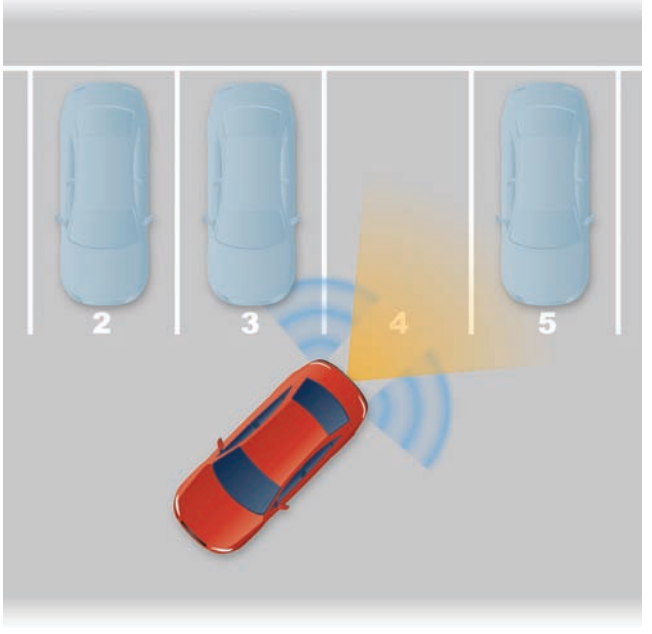


Figure 2.1 Parking sensor

There is no way of knowing how many times this loop will be repeated so potentially it could go on forever – a so-called infinite loop. This example is also known as a `Repeat...Until` loop as it repeats the instruction until a condition is met.

To check for a condition before the code is run, you can use what is commonly called a `While` or `Do while` loop. For example, a program that converts marks to grades might use the following line of code:

```
While Mark <=100
  Convert Mark to Grade
End While
```

In this case, it checks the condition before the code is run. If the mark is over 100, then the code inside the `While` loop will not even start.

Nested loops

In the same way that you can nest selection statements together, it is also possible to have a loop within a loop. For example, an algorithm to create a web counter on a web page may have 8 digits allowing for numbers up to 10 million. Starting with the units, the program counts from 0 to 9. When it reaches 9, it starts again from 0, but it also has to increment the value in the tens column by 1. The units will move round 10 times before the tens, then moves once. The tens column moves around 10 times and then the hundreds increments by 1 and so on.

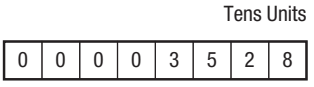


Figure 2.2 A web counter

KEYWORD

Sequence: the principle of putting the correct instructions in the right order within a program.

The same algorithm can therefore be used for each digit and can be nested together so that the code is carried out in the correct **sequence**. The code below shows a nested loop just for the units and tens:

```
Tens = 0
Units = 0
While Tens < 10
   While Units < 10
      Output Tens and Units to web counter
      Units = Units + 1
   End While
   Tens = Tens + 1
   Units = 0
End While
```

Notice that the way the code is indented indicates the sequence of events. This shows that for every iteration of the outer loop, the inner loop will be completed.

Structures such as those mentioned in this chapter are one of the characteristics of a high-level language. They are easy to understand when they are viewed in isolation, but the problems start when you try to put a series of constructs together to do something more useful than deciding if someone is old enough to drive a car or to move a device forwards. In order to create larger, more useful programs, you need to plan ahead and organise your code.

Practice questions can be found at the end of the section on pages 46 and 47.

KEY POINTS

- Programming statements are built up using four main constructs: sequence, selection, repetition (also known as iteration) and assignment.
- Sequence is putting the instructions in the correct order to perform a task.
- Selection statements choose what action to take based on specified criteria. For example, `If...Then` statements.
- Iteration is where a particular step or steps are repeated in order to achieve a certain task. For example, `For... Next` statements.
- Assignment is the process of giving values to variables and constants. For example, `Age = 25`.

TASKS

1. Write examples of the three main types of programming statement: assignment, selection, iteration.
2. Give two examples where an iterative process might be used.
3. Explain the difference between definite and indefinite iteration.
4. Explain the concept of a nested statement.
5. Why is the sequence of programming statements so important? Use an example to explain.
6. What is syntax and why is it important? Use an example to explain.

STUDY / RESEARCH TASKS

1. Identify a real-life situation where it might be useful to use the following constructs within a program:
   a) iteration
   b) selection.
2. Write a program that reads in a file of test marks and then converts them to grades.
3. Write a program that works out the postage charges for parcels of different weights.
4. Write a program that simulates the odometer on a car.

Glossary

Abstract data type: a conceptual model of how data can be stored and the operations that can be carried out on the data.
Abstraction by generalisation/categorisation: the concept of reducing problems by putting similar aspects of a problem into hierarchical categories.
Accepting state: the state that identifies whether an input string has been accepted. Also known as the goal state.
Address bus: used to specify a physical address in memory so that the data bus can access it.
Addressable memory: the concept that data and instructions are stored in memory using discrete addresses.
Addressing mode: the way in which the operand is interpreted.
Adjacency list: a data structure that stores a list of nodes with their adjacent nodes.
Adjacency matrix: a data structure set up as a two-dimensional array or grid that shows whether there is an edge between each pair of nodes.
Algorithm: a sequence of steps that can be followed to complete a task and that always terminates.
Alphabet: the acceptable symbols (characters, numbers) for a given Turing machine.
Analysis: the first stage of system development where the problem is identified, researched and alternative solutions proposed.
AND: Boolean operation that outputs true if both inputs are true.
AND gate: result is true if both inputs are true.
Application program interface (API): a set of subroutines that enable one program to interface with another program.
Application software: programs that perform specific tasks that would need doing even if computers didn't exist, e.g. editing text, carrying out calculations.
Arc: a join or relationship between two nodes – also known as an edge.
Argument: a value that is passed into a function or subroutine.
Arithmetic Logic Unit (ALU): part of the processor that processes and manipulates data.
Arithmetic operation: instructions that perform basic maths such as +, –, /, ×.
Array: a set of related data items stored under a single identifier and are accessed based on their position. Can work on one or more dimensions.
ASCII: a standard binary coding system for characters and numbers.
Assembler: a program that translates a program written in assembly language into machine code.
Assembly language: a way of programming using mnemonics.
Assignment: the process of giving a value to a variable or constant.
Association aggregation: creating an object that contains other objects, which can continue to exist even if the containing object is destroyed.
Associative array: a two-dimensional structure containing key/value pairs of data.
Asymmetric encryption: where a public and private key are used to encrypt and decrypt data.
Asynchronous data transmission: data is transmitted between two devices that do not share a common clock signal.
Attribute: a characteristic or piece of information about an entity, which would be stored as a field in a relational database.
Automation: creating a computer model of a real-life situation and putting it into action.
Backus–Naur Form (BNF): a form of notation for describing the syntax used by a programming language.
Bandwidth: a measure of the capacity of the channel down which the data is being sent. Measured in hertz (Hz).
Barcode reader: a device that uses lasers or LEDs to read the black and white lines of a barcode.
Baudot code: a five-digit character code that predates ASCII and Unicode.
Big data: a generic term for large or complex datasets that are difficult to store and analyse.
Binary file: stores data as sequences of 0s and 1s.
Binary search: a technique for searching data that works by splitting datasets in half repeatedly until the search data is found.
Binary tree search: a technique for searching a binary tree that traversed the tree until the search term is found.
Binary tree: a structure where each node can only have up to two child nodes attached to it.
Bit: a single binary digit from a binary number – either a zero or a one.
Bit rate: the rate at which data is actually being transmitted. Measured in bits per second.
Bit-mapped graphic: an image made up of individual pixels.
Black box testing: using test data to test for an expected outcome.
Block: in data storage it is the concept of storing data into set groups of bits and bytes of a fixed length.
Block interface: code that describes the data being passed from one subroutine to another.
BODMAS: a methodology for evaluating mathematical expressions in a particular sequence.
Boolean expression: an equation made up of Boolean operations.
Boolean operation: a single Boolean function that results in a TRUE or FALSE value.
Boundary test data: test data on or close to the boundary of the acceptable range.
Branch operations: operations within an instruction set that allow you to move from one part of the program to another.
Breadth first: a method for traversing a graph that explores nodes closest to the starting node first before progressively exploring nodes that are further away.
Bubble sort: a technique for putting data in order by repeatedly stepping through an array, comparing adjacent elements and swapping them if necessary until the array is in order.
Bus: microscopic parallel wires that transmit data between internal components.