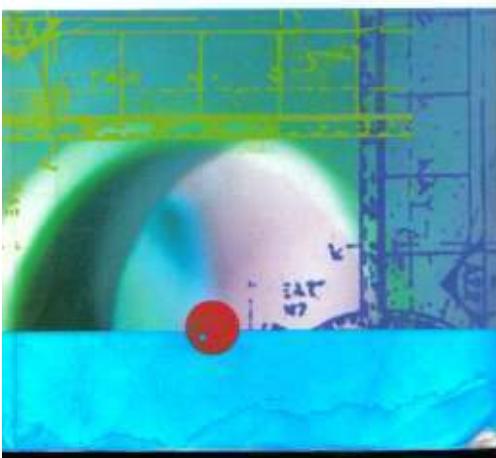


Simple Program Design

A Step-by-Step Approach

FOURTH EDITION

Lesley Anne Robertson





simple program design

Simple Program Design: A Step-by-Step Approach

Fourth Edition

Lesley Anne Robertson

Contributing Authors

Wendy Doube

Kim Styles

THOMSON
—
COURSE TECHNOLOGY

Australia • Canada • Mexico • Singapore • Spain • United Kingdom • United States



Simple Program Design: A Step-by-Step Approach, Fourth Edition
is published by Course Technology.

Executive Vice President, Publisher
Kristen Duerr

Managing Editor
Jennifer Muroff

Product Manager
Tricia Boyle

Production Editor
Aimee Poirier

Senior Product Marketing Manager
Jason Sakos

Associate Product Manager
Nick Lombardi

Cover Designer
Abigail Scholz

Editor
Robyn Flemming

Illustrations
Rick Noble

Text Designer
Polar Design

**COPYRIGHT © 2004 Lesley Anne
Robertson.**
Thomson Learning™ is a trademark
used herein under license.

ALL RIGHTS RESERVED. No part of this
work may be reproduced, transcribed,
or used in any form or by any means –
graphic, electronic, or mechanical,
including photocopying, recording,
taping, Web distribution, or information
storage and retrieval systems –
without prior written permission of
the publisher.

Disclaimer
Course Technology reserves the right
to revise this publication and make
changes from time to time as neces-
sary without notice. The Web
addresses in this book are subject to
change from time to time as necessary
without notice.

For more information, contact Course
Technology, 25 Thomson Place,
Boston, MA 02210; or find us on the
World Wide Web at www.course.com

For permission to use material from
this text or products, contact us by

- Web: www.thomsonrights.com
- Phone: 1-800-730-2214
- Fax: 1-800-730-2215

ISBN 0-619-16046-2

Printed and bound in Hong Kong by
C & C Offset Printing Co., Ltd.

Contents

Preface

xi

1 Program design

Describes the steps in the program development process, and introduces current program design methodologies, procedural and object-oriented programming, algorithms, pseudocode and program data.

1.1	Steps in program development	2
1.2	Program design methodology	4
1.3	Procedural versus object-oriented programming	5
1.4	An introduction to algorithms and pseudocode	6
1.5	Program data	7
	Chapter summary	10

2 Pseudocode

Introduces common words, keywords and meaningful names when writing pseudocode. The Structure Theorem is introduced, and the three basic control structures are established. Pseudocode is used to represent each control structure.

2.1	How to write pseudocode	12
2.3	Meaningful names	15
2.3	The Structure Theorem	15
	Chapter summary	17

3 Developing an algorithm

Introduces methods of analysing a problem and developing a solution. Simple algorithms that use the sequence control structure are developed, and methods of manually checking the algorithm are determined.

3.1	Defining the problem	19
3.2	Designing a solution algorithm	23
3.3	Checking the solution algorithm	25
	Chapter summary	33
	Programming problems	33

4 Selection control structures

Expands the selection control structure by introducing multiple selection, nested selection and the case construct in pseudocode. Several algorithms, using variations of the selection control structure, are developed.

4.1	The selection control structure	36
4.2	Algorithms using selection	40
4.3	The case structure	47
	Chapter summary	51
	Programming problems	51

5 Repetition control structures

Develops algorithms that use the repetition control structure in the form of DOWHILE, REPEAT...UNTIL, and counted repetition loops.

5.1	Repetition using the DOWHILE structure	55
5.2	Repetition using the REPEAT...UNTIL structure	64
5.3	Counted repetition	67
	Chapter summary	69
	Programming problems	70

6 Pseudocode algorithms using sequence, selection and repetition

Develops algorithms to solve eight simple programming problems using combinations of sequence, selection and repetition constructs. Each problem is properly defined; the control structures required are established; a pseudocode algorithm is developed; and the solution is manually checked for logic errors.

6.1	Eight solution algorithms	73
	Chapter summary	84
	Programming problems	85

7 Array processing

Introduces arrays, operations on arrays, and algorithms that manipulate arrays. Algorithms for single and two-dimensional arrays, which initialise the elements of an array, search an array and write out the contents of an array, are presented.

7.1	Array processing	88
7.2	Initialising the elements of an array	91
7.3	Searching an array	94
7.4	Writing out the contents of an array	96
7.5	Programming examples using arrays	97

7.6	Two-dimensional arrays	101
	Chapter summary	105
	Programming problems	105

8

First steps in modularisation

Introduces modularisation as a means of dividing a problem into subtasks. Hierarchy charts and parameter passing are introduced, and several algorithms that use a modular structure are developed.

8.1	Modularisation	110
8.2	Hierarchy charts or structure charts	113
8.3	Further modularisation	115
8.4	Communication between modules	116
8.5	Using parameters in program design	120
8.6	Steps in modularisation	124
8.7	Programming examples using modules	124
	Chapter summary	132
	Programming problems	133

9

Further modularisation, cohesion and coupling

Develops modularisation further, using a more complex problem. Module cohesion and coupling are introduced, several levels of cohesion and coupling are described, and pseudocode examples of each level are provided.

9.1	Steps in modularisation	137
9.2	Module cohesion	143
9.3	Module coupling	149
	Chapter summary	154
	Programming problems	154

10

General algorithms for common business problems

Develops a general pseudocode algorithm for four common business applications. All problems are defined; a hierarchy chart is established; and a pseudocode algorithm is developed, using a mainline and several subordinate modules. The topics covered include report generation with page break, a single-level control break, a multiple-level control break and a sequential file update program.

10.1	Program structure	161
10.2	Report generation with page break	162
10.3	Single-level control break	164

10.4	Multiple-level control break	168
10.5	Sequential file update	173
	Chapter summary	179
	Programming problems	180

1 1

Detailed object-oriented design

Introduces object-oriented design, classes and objects, attributes, responsibilities, operations, accessors and mutators, and information hiding. The steps required to create an object-oriented solution to a problem are provided and solution algorithms developed.

11.1	Introduction to object-oriented design	188
11.2	Steps in creating an object-oriented solution	193
11.3	Programming example using object-oriented design	203
11.4	Interface and GUI objects	207
	Chapter summary	210
	Programming problems	210

1 2

Simple object-oriented design for multiple classes

Introduces the concept of multiple classes, relationships between classes and polymorphism in object-oriented design. Discusses the relationship between classes, including aggregation, composition and generalisation, and lists the steps required to create an object-oriented design to a problem with multiple classes.

12.1	Object-oriented design with multiple classes	213
12.2	Programming example with multiple classes	217
	Chapter summary	232
	Programming problems	233

1 3

Conclusion

A revision of the steps involved in good top-down program design.

13.1	Simple program design	235
	Chapter summary	236

Appendix 1 Flowcharts

Introduces flowcharts for those students who prefer a more graphic approach to program design. Algorithms that use a combination of sequence, selection and repetition are developed in some detail.

	The three basic control structures	239
	Simple algorithms that use the sequence control structure	242
	Flowcharts and the selection control structure	246

Simple algorithms that use the selection control structure	248
The case structure expressed as a flowchart	254
Flowcharts and the repetition control structure	256
Simple algorithms that use the repetition control structure	257
Further examples using flowcharts	265
Flowcharts and modules	289

Appendix 2 Nassi-Schneiderman diagrams

Introduces Nassi-Schneiderman diagrams for those students who prefer a more diagrammatic approach to program design. Algorithms that use a combination of sequence, selection and repetition constructs are developed in some detail.

The three basic control structures	303
Simple algorithms that use the sequence control structure	305
N-S diagrams and the selection control structure	307
Simple algorithms that use the selection control structure	309
The case structure expressed as an N-S diagram	313
N-S diagrams and the repetition control structure	314
Simple algorithms that use the repetition control structure	315

Appendix 3 Special algorithms

Contains a number of algorithms that are not included in the body of the textbook and yet may be required at some time in a programmer's career.

Sorting algorithms	321
Dynamic data structures	324

Appendix 4 Translating pseudocode into computer languages: quick reference chart

329

Glossary	336
-----------------	-----

Index	343
--------------	-----

Preface

W

ith the increased popularity of programming courses in our universities, colleges and technical institutions, there is a need for an easy-to-read textbook on computer program design. There are already dozens of introductory programming texts using specific languages such as C++, Visual Basic, Pascal and COBOL, but they usually gloss over the important step of designing a solution to a given programming problem.

This textbook tackles the subject of program design by using modern programming techniques and pseudocode to develop a solution algorithm. The recommended pseudocode has been chosen because of its closeness to written English, its versatility and ease of manipulation, and its similarity to the syntax of most structured programming languages.

Simple Program Design, Fourth Edition is designed for programmers who want to develop good programming skills for solving common business problems. Too often, programmers, when are faced with a problem, launch straight into the code of their chosen programming language, instead of concentrating on the actual problem at hand. They become bogged down with the syntax and format of the language, and often spend many hours getting the program to work. Using this textbook, the programmer will learn how to properly define the problem, how to divide it into modules, how to design a solution algorithm, and how to prove the algorithm's correctness, before commencing any program coding. By using pseudocode and modern programming techniques, the programmer can concentrate on developing a well-designed and correct solution, and thus eliminate many frustrating hours at the testing phase.

The content of the book covers program design in two distinct sections. Chapters 1 to 10 cover algorithm design in the context of traditional programming languages. The section begins with a basic introduction to program design methodology, and the steps in developing a solution algorithm. Then, concept by concept, the student is introduced to the syntax of pseudocode; methods of defining the problem; the application of basic control structures in the development of the solution algorithm; desk-checking techniques; arrays; module design; hierarchy charts; communication between modules; parameter passing; and module cohesion and coupling.

Chapters 11 and 12 have been designed for students who want to develop skills in object-oriented analysis and design. These chapters were written by

Kim Styles and Wendy Doube, lecturers in computing at the Gippsland School of Computing and Information Technology, Monash University. This section introduces the concepts of object-oriented design and the steps involved in creating an object-oriented solution to a problem. Step-by-step algorithms using object-oriented design are provided, as well as material on multiple classes and interfaces.

Each chapter thoroughly covers the topic at hand, giving practical examples relating to business applications, and a consistently structured approach when representing algorithms and hierarchy charts.

This fourth edition has been thoroughly revised, in keeping with modern program design techniques. It includes a discussion on program design methodology, an improved method for desk checking the solution algorithm, more information and examples on communication between modules, a complete rewrite of the two chapters on object-oriented design and a much more comprehensive section on flowcharts.

Ten programming problems, of increasing complexity, are provided at the end of each chapter, so that teachers have a choice of exercises that matches the widely varying abilities of their students.

I would like to thank Kim Styles and Wendy Doube, lecturers in Computing at Monash University, for their wonderful input on object-oriented design methodology; Victor Cockrell from Curtin University, for his enthusiastic suggestions and his Quick Reference Chart for translating pseudocode into several computer languages; and my brother, Rick Noble, for his amusing cartoons at the beginning of each chapter.

Lesley Anne Robertson

The Author

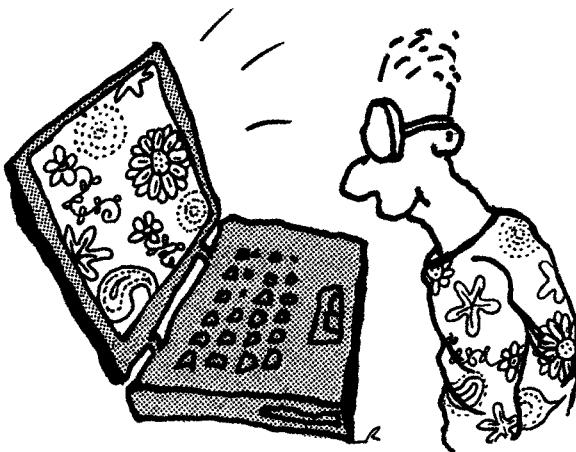
Lesley Anne Robertson was introduced to pseudocode and program design methodology when she joined IBM, Australia, in 1973 as a trainee programmer. Since then, she has consistently used these techniques as a programmer, systems analyst, and Lecturer in Computing at the University of Western Sydney, NSW, where she taught computer program design for 11 years.

Lesley now lives on a vineyard and winery in Mudgee, Australia, with her daughters Lucy and Sally.

Chapter

Program design

1



Objectives

- To describe the steps in the program development process
- To introduce current program design methodology
- To introduce procedural and object-oriented programming
- To introduce algorithms and pseudocode
- To describe program data

Outline

- 1.1 Steps in program development
 - 1.2 Program design methodology
 - 1.3 Procedural versus object-oriented programming
 - 1.4 An introduction to algorithms and pseudocode
 - 1.5 Program data
- Chapter summary

1.1 Steps in program development

Computer programming is an art. Many people believe that a programmer must be good at mathematics, have a memory for figures and technical information, and be prepared to spend many hours sitting at a computer, typing programs. However, given the right tools, and steps to follow, anyone can write well-designed programs. It is a task worth doing, as it is both stimulating and fulfilling.

Programming can be defined as the development of a solution to an identified problem, and the setting up of a related series of instructions which, when directed through computer hardware, will produce the desired results. It is the first part of this definition that satisfies the programmer's creative needs: that is, to design a solution to an identified problem. Yet this step is so often overlooked. Leaping straight into the coding phase without first designing a proper solution usually results in a program that contains a lot of errors. Often the programmer needs to spend a significant amount of time finding these errors and correcting them. A more experienced programmer will design a solution to the program first, desk check this solution, and then code the program in a chosen programming language.

There are seven basic steps in the development of a program, as follows.

1 Define the problem

This step involves carefully reading and rereading the problem until you understand completely what is required. To help with this initial analysis, the problem should be divided into three separate components:

- the inputs
- the outputs
- the processing steps to produce the required outputs.

A defining diagram, as described in Chapter 3, is recommended in this analysis phase, as it helps to separate and define the three components.

2 Outline the solution

Once the problem has been defined, you may decide to decompose it into smaller tasks or steps, and establish an outline solution. This initial outline is usually a rough draft of the solution and may include:

- the major processing steps involved
- the major subtasks (if any)
- the user interface (if any)
- the major control structures (e.g. repetition loops)
- the major variables and record structures
- the mainline logic.

The solution outline may also include a hierarchy or structure chart. The steps involved in developing this outline solution are detailed in Chapters 2 to 6.

3 Develop the outline into an algorithm

The solution outline developed in Step 2 is then expanded into an algorithm: a set of precise steps that describe exactly the tasks to be performed and the order in which they are to be carried out. This book uses pseudocode (a form of structured English) to represent the solution algorithm. Flowcharts and Nassi-Schneiderman diagrams are also provided in Appendixes 1 and 2 for those who prefer a more pictorial method of algorithm representation.

4 Test the algorithm for correctness

This step is one of the most important in the development of a program, and yet it is the step most often forgotten. The main purpose of desk checking the algorithm is to identify major logic errors early, so that they may be easily corrected. Test data needs to be walked through each step in the algorithm to check that the instructions described in the algorithm will actually do what they are supposed to. The programmer walks through the logic of the algorithm, exactly as a computer would, keeping track of all major variables on a sheet of paper. Chapter 3 recommends the use of a desk check table to desk check the algorithm, and many examples of its use are provided.

5 Code the algorithm into a specific programming language

Only after all design considerations have been met in the previous four steps should you actually start to code the program into your chosen programming language.

6 Run the program on the computer

This step uses a program compiler and programmer-designed test data to machine test the code for syntax errors (those detected at compile time) and logic errors (those detected at run time). This is usually the most rewarding step in the program development process. If the program has been well designed, the time-wasting frustration and despair often associated with program testing are reduced to a minimum. This step may need to be performed several times until you are satisfied that the program is running as required.

7 Document and maintain the program

Program documentation should not be listed as the last step in the program development process, as it is really an ongoing task from the initial definition of the problem to the final test result.

Documentation involves both external documentation (such as hierarchy charts, the solution algorithm and test data results) and internal documentation that may have been coded in the program. Program maintenance refers to changes that may need to be made to a program throughout its life. Often, these changes are performed by a different programmer from the one who

initially wrote the program. If the program has been well designed using structured programming techniques, the code will be seen as self-documenting, resulting in easier maintenance.

1.2 Program design methodology

The fundamental principle of program design is based on the fact that a program accepts input data, processes that data, and then delivers that data to the program user as output. Recently, a number of different approaches to program design have emerged, the most common being:

- procedure-driven
- event-driven
- data-driven.

Procedure-driven program design



The procedure-driven approach to program design is based on the idea that the most important feature of a program is 'what' it does – that is, its processes or functions. The flow of data into and out of each process or function is then considered and a strategy developed to break each function into smaller and more specific flows of data. The details about the actual structure of the data are not considered until all the high-level processes or functions of the program have been defined.

Event-driven program design



The event-driven approach to program design is based on the idea that an event or interaction with the outside world can cause a program to change from one known state to another. The initial state of a program is identified, then all the triggers that represent valid events for that state are established. Each of these events results in the program changing to a new defined state, where it stays until the next event occurs. For example, when a program user decides to click the left mouse button, click the right mouse button, drag the mouse or double click the mouse, each action could trigger a different 'event' within the program and thus result in a different program state.

Data-driven program design



The data-driven approach to program design is based on the idea that the data in a program is more stable than the processes involved. It begins with an analysis of the data and the relationships between the data, in order to determine the fundamental data structures. Once these data structures have been defined, the required data outputs are examined in order to establish what processes are required to convert the input data to the required output.

The choice between procedure-driven, event-driven or data-driven program design methodologies is usually determined by the selection of a programming language. However, regardless of the program design method

chosen, the programmer must develop the necessary basic skills to be able to design a solution algorithm to a given problem. These basic skills include a well-defined and disciplined approach to designing the solution algorithm and adherence to the recommended program development process, as follows:

- Step 1: Define the problem.
- Step 2: Outline the solution (or user interface).
- Step 3: Develop the outline into a solution algorithm.
- Step 4: Test the algorithm for correctness.
- Step 5: Code the algorithm into a specific programming language.
- Step 6: Run the program on the computer.
- Step 7: Document and maintain the program.

1.3

Procedural versus object-oriented programming

Procedural programming is based on a structured, top-down approach to writing effective programs. The approach concentrates on ‘what’ a program has to do and involves identifying and organising the ‘processes’ in the program solution. The problem is usually decomposed into separate tasks or functions and includes top-down development and modular design.

Top-down development

In the top-down development of a program design, a general solution to the problem is outlined first. This is then broken down gradually into more detailed steps until finally the most detailed levels have been completed. It is only after this process of ‘functional decomposition’ (or ‘stepwise refinement’) that the programmer starts to code. The result of this systematic, disciplined approach to program design is a higher precision of programming than was previously possible.

Modular design



Procedural programming also incorporates the concept of modular design, which involves grouping tasks together because they all perform the same function (for example, calculating sales tax or printing report headings). Modular design is connected directly to top-down development, as the steps or subtasks, into which the programmer divides the program solution, actually form the future modules of the program. Good modular design also assists in the reading and understanding of the program.

Object-oriented programming



Object-oriented programming is also based on decomposing the problem; however, the primary focus is on the things (or objects) that make up the

program. The program is concerned with how the objects behave, so it breaks the problem into a set of separate objects that perform actions and relate to each other. These objects have definite properties, and each object is responsible for carrying out a series of related tasks.

This book looks at both approaches to program design, procedural and object-oriented. It is then left to the programmer to decide which methodology he or she will use. It must be noted, however, that, regardless of design methodology or programming language, all programmers must have the basic skills to design solution algorithms. It is the intention of this book to provide these skills.

1.4

An introduction to algorithms and pseudocode

A program must be systematically and properly designed before coding begins. This design process results in the construction of an algorithm.

What is an algorithm?

An algorithm is like a recipe; it lists the steps involved in accomplishing a task. It can be defined in programming terms as a set of detailed, unambiguous and ordered instructions developed to describe the processes necessary to produce the desired output from a given input. The algorithm is written in simple English and is not a formal document. However, to be useful, there are some principles which should be adhered to. An algorithm must:

- be lucid, precise and unambiguous
- give the correct solution in all cases
- eventually end.

For example, if you want to instruct someone to add up a list of prices on a pocket calculator, you might write an algorithm such as the following:

```
Turn on calculator
Clear calculator
Repeat the following instructions
    Key in dollar amount
    Key in decimal point (.)
    Key in cents amount
    Press addition (+) key
Until all prices have been entered
Write down total price
Turn off calculator
```

Notice that in this algorithm the first two steps are performed once, before the repetitive process of entering the prices. After all the prices have been entered and summed, the total price can be written down and the cal-

culator turned off. These final two activities are also performed only once. This algorithm satisfies the desired list of properties: it lists all the steps in the correct order from top to bottom, in a definite and unambiguous fashion, until a correct solution is reached. Notice that the steps to be repeated (entering and summing the prices) are indented, both to separate them from those steps performed only once and to emphasise the repetitive nature of their action. It is important to use indentation when writing solution algorithms because it helps to differentiate between the different control structures.

What is pseudocode?



Pseudocode, flowcharts and Nassi-Schneiderman diagrams are all popular ways of representing algorithms. Flowcharts and Nassi-Schneiderman diagrams are covered in Appendixes 1 and 2, while pseudocode has been chosen as the primary method of representing an algorithm because it is easy to read and write and allows the programmer to concentrate on the logic of the problem. Pseudocode is really structured English. It is English that has been formalised and abbreviated to look like high-level computer languages.

There is no standard pseudocode at present. Authors seem to adopt their own special techniques and sets of rules, which often resemble a particular programming language. This book attempts to establish a standard pseudocode for use by all programmers, regardless of the programming language they choose. Like many versions of pseudocode, this version has certain conventions, as follows:

- 1 Statements are written in simple English.
- 2 Each instruction is written on a separate line.
- 3 Keywords and indentation are used to signify particular control structures.
- 4 Each set of instructions is written from top to bottom, with only one entry and one exit.
- 5 Groups of statements may be formed into modules, and that group given a name.

1.5 Program data

Because programs are written to process data, you must have a good understanding of the nature and structure of the data being processed. Data within a program may be a single variable, such as an integer or a character; or a group item (sometimes called an aggregate), such as an array, or a file.

Variables, constants and literals

A variable is the name given to a collection of memory cells, designed to store a particular data item. It is called a variable because the value stored in those memory cells may change or vary as the program executes. For example, the variable `total_amount` may contain several values during the execution of the program.

A constant is a data item with a name and a value that remain the same during the execution of the program. For example, the name 'fifty' may be given to a data item that contains the value 50.

A literal is a constant whose name is the written representation of its value. For example, the program may contain the literal '50'.

Data types

At the beginning of a program, the programmer must clearly define the form or type of the data to be collected. The data types can be elementary data items or data structures.

Elementary data items

An elementary data item is one containing a single variable that is always treated as a unit. These data items are usually classified into data types. A data type consists of a set of data values and a set of operations that can be performed on those values. The most common elementary data types are:

integer:

representing a set of whole numbers, positive, negative or zero

e.g. 3, 576, -5

real:

representing a set of numbers, positive or negative, which may include values before or after a decimal point. These are sometimes referred to as floating point numbers

e.g. 19.2, 1.92E+01, -0.01

character:

representing the set of characters on the keyboard, plus some special characters

e.g. 'A', 'b', '\$'

Boolean:

representing a control flag or switch, which may contain one of only two possible values; true or false.

Data structures

A data structure is an aggregate of other data items. The data items that it contains are its components, which may be elementary data items or another data structure. In a data structure, data is grouped together in a particular way, which reflects the situation with which the program is concerned. The most common data structures are:

record:

a collection of data items or fields that all bear some relationship to one another. For example, a student record may contain the student's number, name, address and enrolled subjects.

file:

a collection of records. For example, a student file may contain a collection of the above student records.

array:

a data structure that is made up of a number of variables or data items that all have the same data type and are accessed by the same name. For example, an array called 'scores' may contain a collection of students' exam scores. Access to the individual items in the array is made by the use of an index or subscript beside the name of the array – for example, scores (3).

string:

a collection of characters that can be fixed or variable. For example, the string 'Jenny Parker' may represent a student's name.

Files

A popular method of storing information is to enter and store data on a file. The major advantages of using files are:

- Several different programs can access the same data.
- The data can be entered and reused several times.
- The data can be easily updated and maintained.
- The accuracy of the data is easier to enforce.

There are two different methods of storing data on files:

- sequential or text files, where data is stored and retrieved sequentially
- direct or random-access files, where data is stored and retrieved randomly, using a key or index.

Sequential files may be opened to read or to write, but not both operations on the same file. Random-access files can be opened to read and write on the same file.

Data validation

Data should always undergo a validation check before it is processed by a program. Different types of data require different checks – for example:

- *Correct type*: the input data should match the data type definition stated at the beginning of the program.
- *Correct range*: the input data should be within a required set of values.
- *Correct length*: the input data – for example, string – should be the correct length.
- *Completeness*: all required fields should be present.
- *Correct date*: an incoming date should be acceptable.

Chapter summary

In this chapter, the steps in program development were introduced and briefly described. These seven steps are:

- 1** Define the problem.
- 2** Outline the solution.
- 3** Develop the outline into an algorithm.
- 4** Test the algorithm for correctness.
- 5** Code the algorithm into a specific programming language.
- 6** Run the program on the computer.
- 7** Document and maintain the program.

Three different approaches to program design were introduced, namely procedure-driven, event-driven and data-driven. Procedural programming and object-oriented programming were introduced, along with top-down development and modular design.

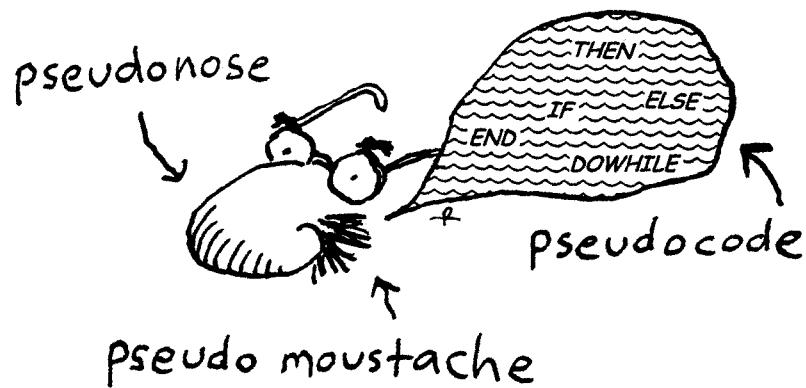
An algorithm was defined as a set of detailed, unambiguous and ordered instructions developed to describe the processes necessary to produce the desired output from the given input. Pseudocode is an English-like way of representing the algorithm; its advantages and some conventions for its use were listed.

Programmers need to have a good understanding of the data to be processed; therefore, data variables, constants and literals were defined, as well as elementary data items, data structures, files and data validation.

Chapter

Pseudocode

2



Objectives

- To introduce common words, keywords and meaningful names when writing pseudocode
- To define the three basic control structures as set out in the Structure Theorem
- To illustrate the three basic control structures using pseudocode

Outline

- 2.1 How to write pseudocode
 - 2.2 Meaningful names
 - 2.3 The Structure Theorem
- Chapter summary

2.1 How to write pseudocode

When designing a solution algorithm, you need to keep in mind that a computer will eventually perform the set of instructions written. That is, if you use words and phrases in the pseudocode which are in line with basic computer operations, the translation from the pseudocode algorithm to a specific programming language becomes quite simple.

This chapter establishes six basic computer operations and introduces common words and keywords used to represent these operations in pseudocode. Each operation can be represented as a straightforward English instruction, with keywords and indentation to signify a particular control structure.

Six basic computer operations

1 A computer can receive information

When a computer is required to receive information or input from a particular source, whether it be a terminal, a disk or any other device, the verbs Read and Get are used in pseudocode. Read is usually used when the algorithm is to receive input from a record on a file, while Get is used when the algorithm is to receive input from the keyboard. For example, typical pseudocode instructions to receive information are:

```
Read student name  
Get system date  
Read number_1, number_2  
Get tax_code
```

Each example uses a single verb, Read or Get, followed by one or more nouns to indicate what data is to be obtained.

2 A computer can put out information

When a computer is required to supply information or output to a device, the verbs Print, Write, Put, Output or Display are used in pseudocode. Print is usually used when the output is to be sent to the printer, while Write is used when the output is to be written to a file. If the output is to be written to the screen, the words Put, Output or Display are used in pseudocode. Typical pseudocode examples are:

```
Print 'Program Completed'  
Write customer record to master file  
Put out name, address and postcode  
Output total_tax  
Display 'End of data'
```

Usually an output Prompt instruction is required before an input Get instruction. The Prompt verb causes a message to be sent to the screen, which requires the user to respond, usually by providing input – for example:

```
Prompt for student_mark  
Get student_mark
```

3 A computer can perform arithmetic

Most programs require the computer to perform some sort of mathematical calculation, or formula, and for these, a programmer may use either actual mathematical symbols or the words for those symbols. For instance, the same pseudocode instruction can be expressed as either of the following:

Add number to total
total = total + number

Both expressions clearly instruct the computer to add one value to another, so either is acceptable in pseudocode. The equal symbol '=' has been used to indicate assignment of a value as a result of some processing.

To be consistent with high-level programming languages, the following symbols can be written in pseudocode:

+ for Add
- for Subtract
* for Multiply
/ for Divide
() for Parentheses

The verbs Compute and Calculate are also available. Some pseudocode examples to perform a calculation are:

Divide total_marks by student_count
sales_tax = cost_price * 0.10
Compute C = (F - 32) * 5/9

When writing mathematical calculations for the computer, standard mathematical 'order of operations' applies to pseudocode and most computer languages. The first operation carried out will be any calculation contained within parentheses. Next, any multiplication or division, as it occurs from left to right, will be performed. Then, any addition or subtraction, as it occurs from left to right, will be performed.

4 A computer can assign a value to a variable or memory location

There are three cases where you may write pseudocode to assign a value to a variable or memory location:

- 1 To give data an initial value in pseudocode, the verbs Initialise or Set are used.
- 2 To assign a value as a result of some processing, the symbols '=' or ' \leftarrow ' are written.
- 3 To keep a variable for later use, the verbs Save or Store are used.

Some typical pseudocode examples are:

Initialise total_price to zero
Set student_count to 0
total_price = cost_price + sales_tax
total_price \leftarrow cost_price + sales_tax
Store customer_num in last_customer_num

Note that the '=' symbol is used to assign a value to a variable as a result of some processing and is not equivalent to the mathematical '=' symbol. For this reason, some programmers prefer to use the ' \leftarrow ' symbol to represent the assign operation.

5 A computer can compare two variables and select one of two alternate actions

An important computer operation available to the programmer is the ability to compare two variables and then, as a result of the comparison, select one of two alternate actions. To represent this operation in pseudocode, special keywords are used: IF, THEN and ELSE. The comparison of data is established in the IF clause, and the choice of alternatives is determined by the THEN or ELSE options. Only one of these alternatives will be performed. A typical pseudocode example to illustrate this operation is:

```
IF student_attendance_status is part_time THEN
    add 1 to part_time_count
ELSE
    add 1 to full_time_count
ENDIF
```

In this example the attendance status of the student is investigated, with the result that either the part_time_count or the full_time_count accumulator is incremented. Note the use of indentation to emphasise the THEN and ELSE options, and the use of the delimiter ENDIF to close the operation.

6 A computer can repeat a group of actions

When there is a sequence of processing steps that need to be repeated, two special keywords, DOWHILE and ENDDO, are used in pseudocode. The condition for the repetition of a group of actions is established in the DOWHILE clause, and the actions to be repeated are listed beneath it. For example:

```
DOWHILE student_total < 50
    Read student record
    Print student name, address to report
    Add 1 to student_total
ENDDO
```

In this example it is easy to see the statements that are to be repeated, as they immediately follow the DOWHILE statement and are indented for added emphasis. The condition that controls and eventually terminates the repetition is established in the DOWHILE clause, and the keyword ENDDO acts as a delimiter. As soon as the condition for repetition is found to be false, control passes to the next statement after the ENDDO.

2.2

Meaningful names

When designing a solution algorithm, a programmer must introduce some unique names, which will be used to represent the variables or objects in the problem. All names should be meaningful. A name given to a variable is simply a method of identifying a particular storage location in the computer.

The uniqueness of a name will differentiate it from other locations. Often a name describes the type of data stored in a particular variable. For instance, a variable may be one of the three simple data types, an integer, a real number or a character. The name itself should be transparent enough to adequately describe the variable – for example, number1, number2 and number3 are more meaningful names for three numbers than A, B and C.

If you need to use more than one word in a variable name, then underscores are useful as word separators – for example, sales_tax and word_count. Most programming languages do not tolerate a space in a variable name, as a space would signal the end of the variable name and thus imply that there were two variables. If you don't want to use an underscore, then words can be joined together with the use of a capital letter as a word separator – for example, salesTax and wordCount. For readability, it is not advisable to string words together all in lower case, as a name like carregistration is much harder to read than carRegistration.

2.3

The Structure Theorem

The Structure Theorem revolutionised program design by establishing a structured framework for representing a solution algorithm. The Structure Theorem states that it is possible to write any computer program by using only three basic control structures that are easily represented in pseudocode: sequence, selection and repetition.

The three basic control structures

1 Sequence



The sequence control structure is the straightforward execution of one processing step after another. In pseudocode, we represent this construct as a sequence of pseudocode statements.

```
statement a  
statement b  
statement c
```

The sequence control structure can be used to represent the first four basic computer operations listed previously: to receive information, put out information, perform arithmetic, and assign values. For example, a typical sequence of statements in an algorithm might read:

```
Add 1 to pageCount  
Print heading line1  
Print heading line2  
Set lineCount to zero  
Read customer record
```

These instructions illustrate the sequence control structure as a straightforward list of steps written one after the other, in a top-to-bottom fashion. Each instruction will be executed in the order in which it appears.

2 Selection

The selection control structure is the presentation of a condition and the choice between two actions, the choice depending on whether the condition is true or false. This construct represents the decision-making abilities of the computer and is used to illustrate the fifth basic computer operation, namely to compare two variables and select one of two alternate actions.

In pseudocode, selection is represented by the keywords IF, THEN, ELSE and ENDIF:

```
IF condition p is true THEN  
    statement(s) in true case  
ELSE  
    statement(s) in false case  
ENDIF
```

If condition p is true, then the statement or statements in the true case will be executed, and the statements in the false case will be skipped. Otherwise (the ELSE statement) the statements in the true case will be skipped and statements in the false case will be executed. In either case, control then passes to the next processing step after the delimiter ENDIF. A typical pseudocode example might read:

```
IF student_attendance_status is part_time THEN  
    add 1 to part_time_count  
ELSE  
    add 1 to full_time_count  
ENDIF
```

The selection control structure is discussed fully in Chapter 4.

3 Repetition

The repetition control structure can be defined as the presentation of a set of instructions to be performed repeatedly, as long as a condition is true. The basic idea of repetitive code is that a block of statements is executed again and again, until a terminating condition occurs. This construct represents the sixth basic computer operation, namely to repeat a group of actions. It is written in pseudocode as:

```
DOWHILE condition p is true  
    statement block  
ENDDO
```

The DOWHILE loop is a leading decision loop – that is, the condition is tested before any statements are executed. If the condition in the DOWHILE statement is found to be true, the block of statements following that statement is executed once. The delimiter ENDDO then triggers a return of control to the retesting of the condition. If the condition is still true, the statements are repeated, and so the repetition process continues until the condition is found to be false. Control then passes to the statement that follows the ENDDO statement. It is imperative that at least one statement within the statement block alters the condition and eventually renders it false, because otherwise the logic may result in an endless loop.

Here is a pseudocode example that represents the repetition control structure:

```
Set student_total to zero
DOWHILE student_total < 50
    Read student record
    Print student name, address to report
    Add 1 to student_total
ENDDO
```

This example illustrates a number of points:

- 1 The variable student_total is initialised before the DOWHILE condition is executed.
- 2 As long as student_total is less than 50 (that is, the DOWHILE condition is true), the statement block will be repeated.
- 3 Each time the statement block is executed, one instruction within that block will cause the variable student_total to be incremented.
- 4 After 50 iterations, student_total will equal 50, which causes the DOWHILE condition to become false and the repetition to cease.

It is important to realise that the initialising and subsequent incrementing of the variable tested in the condition is an essential feature of the DOWHILE construct. The repetition control structure is discussed fully in Chapter 5.

Chapter summary

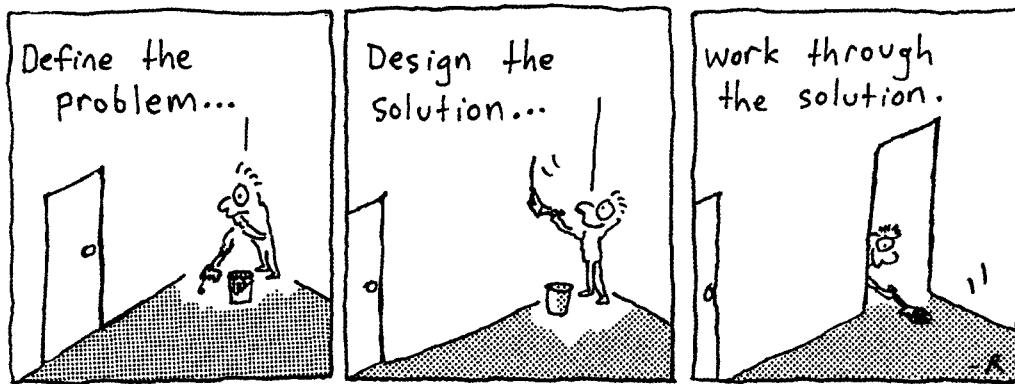
In this chapter, six basic computer operations were listed, along with pseudocode words and keywords to represent them. These operations were: to receive information, put out information, perform arithmetic, assign a value to a variable, decide between two alternate actions, and repeat a group of actions. Typical pseudocode examples were given as illustrations, and the importance of using meaningful names was discussed.

The Structure Theorem was introduced. It states that it is possible to write any computer program by using only three basic control structures: sequence, selection and repetition. Each control structure was defined, and its association with each of the six basic computer operations was indicated. Pseudocode examples for each control structure were provided.

Chapter

Developing an algorithm

3



Objectives

- To introduce methods of analysing a problem and developing a solution
- To develop simple algorithms using the sequence control structure
- To introduce methods of manually checking the developed solution

Outline

- 3.1 Defining the problem
 - 3.2 Designing a solution algorithm
 - 3.3 Checking the solution algorithm
- Chapter summary
Programming problems

3.1 Defining the problem

Chapter 1 described seven steps in the development of a computer program. The very first step, and one of the most important, is defining the problem. This involves carefully reading and rereading the problem until you understand completely what is required. Quite often, additional information will need to be sought to help resolve any ambiguities or deficiencies in the problem specifications. To help with this initial analysis, the problem should be divided into three separate components:

- 1 *Input*: a list of the source data provided to the problem.
- 2 *Output*: a list of the outputs required.
- 3 *Processing*: a list of actions needed to produce the required outputs.

When reading the problem statement, the input and output components are easily identified, because they use descriptive words such as nouns and adjectives. The processing component is also identified easily. The problem statement usually describes the processing steps as actions, using verbs and adverbs.

When dividing a problem into its three different components, you should simply analyse the actual words used in the specification, and divide them into those that are descriptive and those that imply actions. It may help to underline the nouns, adjectives and verbs used in the specification.

In some programming problems, the inputs, processes and outputs may not be clearly defined. In such cases, it is best to concentrate on the outputs required. Doing this will then decide most inputs, and the way will be set for determining the processing steps required to produce the desired output.

At this stage, the processing section should be a list of what actions need to be performed, not how they will be accomplished. Do not attempt to find a solution until the problem has been completely defined. Let's look at a simple example.

EXAMPLE 3.1 Add three numbers

A program is required to read three numbers, add them together and print their total.

Tackle this problem in two stages. First, underline the nouns and adjectives used in the specification. This will establish the input and output components, as well as any objects that are required. With the nouns and adjectives underlined, our example would look like this:

A program is required to read three numbers, add them together and print their total.

By looking at the underlined nouns and adjectives, you can see that the input for this problem is three numbers and the output is the total. It is helpful to write down these first two components in a simple diagram, called a defining diagram.

Input	Processing	Output
number1		total
number2		
number3		

Second, underline (in a different colour) the verbs and adverbs used in the specification. This will establish the actions required. Example 3.1 should now look like this:

A program is required to read three numbers, add them together and print their total.

By looking at the underlined words, you can see that the processing verbs are 'read', 'add together' and 'print'. These steps can now be added to our defining diagram to make it complete. Note that when writing down each processing verb, also include the objects or nouns associated with each verb. The defining diagram now becomes:

Input	Processing	Output
number1	Read three numbers	total
number2	Add numbers together	
number3	Print total number	

Now that all the nouns and verbs in the specification have been considered and the defining diagram is complete, the problem has been properly defined. That is, we now understand the input to the problem, the output to be produced, and the processing steps required to convert the input to the output.

When it comes to writing down the processing steps in an algorithm, you should use words that describe the work to be done in terms of single, specific tasks or functions. For example:

Read three numbers
Add numbers together
Print total number

There is a pattern in the words chosen to describe these steps. Each action is described as a single verb followed by a two-word object. Studies have shown that if you follow this convention to describe a processing step, two benefits will result. First, you are using a disciplined approach to defining the problem; and second, the processing is being divided into separate tasks or functions. This simple operation of dividing a problem into separate functions and choosing a proper name for each function is extremely important later, when considering algorithm modules.

EXAMPLE 3.2 Find average temperature

A program is required to prompt the terminal operator for the maximum and minimum temperature readings on a particular day, accept those readings as integers, and calculate and display to the screen the average temperature, calculated by (maximum temperature + minimum temperature)/2.

First establish the input and output components by underlining the nouns and adjectives in the problem statement.

A program is required to prompt the terminal operator for the maximum and minimum temperature readings on a particular day, accept those readings as integers, and calculate and display to the screen the average temperature, calculated by (maximum temperature + minimum temperature)/2.

The input components are the maximum and minimum temperature readings, and the output is the average temperature. Using meaningful names, these components can be set up in a defining diagram as follows:

Input	Processing	Output
max_temp		avg_temp
min_temp		

Now establish the processing steps by underlining the verbs in the problem statement.

A program is required to prompt the terminal operator for the maximum and minimum temperature readings on a particular day, accept those readings as integers, and calculate and display to the screen the average temperature, calculated by (maximum temperature + minimum temperature)/2.

The processing verbs are 'prompt', 'accept', 'calculate' and 'display'. By finding the associated objects of these verbs, the defining diagram can now be completed, as follows:

Input	Processing	Output
max_temp	Prompt for temperatures	avg_temp
min_temp	Get temperatures Calculate average temperature Display average temperature	

EXAMPLE 3.3 Compute mowing time

A program is required to read from the screen the length and width of a rectangular house block, and the length and width of the rectangular house that has been built on the block. The algorithm should then compute and display the mowing time required to cut the grass around the house, at the rate of two square metres per minute.

To establish the input and output components in this problem, the nouns or objects have been underlined. By reading these words, you can see that the input components are the length and width of the block, and the length and width of the house. The output is the mowing time to cut the grass.

The input and output components can be set up in a defining diagram, as follows:

Input	Processing	Output
block_length		mowing_time
block_width		
house_length		
house_width		

Now the verbs and adverbs in the problem statement can be underlined.

A program is required to read from the screen the length and width of a rectangular house block, and the length and width of the rectangular house that has been built on the block. The algorithm should then compute and display the mowing time required to cut the grass around the house, at the rate of two square metres per minute.

The processing steps can now be added to the defining diagram:

Input	Processing	Output
block_length	Prompt for block measurements	mowing_time
block_width	Get block measurements	
house_length	Prompt for house measurements	
house_width	Get house measurements Calculate mowing area Calculate mowing time	

Remember that at this stage you are only concerned with the fact that the mowing time is to be calculated, not how the calculation will be performed. That will come later, when the solution algorithm is established. You must be absolutely confident of *what* is to be done in the program before you attempt to establish *how* it is done.

3.2 Designing a solution algorithm

Designing a solution algorithm is the most challenging task in the life cycle of a program. Once the problem has been properly defined, you usually begin with a rough sketch of the steps required to solve the problem. The first attempt at designing a particular algorithm usually does not result in a finished product. Steps may be left out, or some that are included may later be altered or deleted. Pseudocode is useful in this trial-and-error process, since it is relatively easy to add, delete or alter an instruction. Do not hesitate to alter algorithms, or even to discard one and start again, if you are not completely satisfied with it. If the algorithm is not correct, the program will never be.

There is some argument that the work of a programmer ends with the algorithm design. After that, a coder or trainee programmer could take over and code the solution algorithm into a specific programming language. In practice, this usually does not happen. However, it is important that you are not too anxious to start coding until the necessary steps of defining the problem and designing the solution algorithm have been completed.

Here are solution algorithms for the preceding three examples. All involve sequence control structures only; there are no decisions or loops, so the solution algorithms are relatively simple.

EXAMPLE 3.4 Solution algorithm for Example 3.1

A program is required to read three numbers, add them together and print their total.

A Defining diagram

Input	Processing	Output
number1	Read three numbers	
number2	Add numbers together	
number3	Print total number	

This diagram shows what is required, and a simple calculation will establish how. Using pseudocode, and the sequence control structure, establish the solution algorithm as follows:

B Solution algorithm

```
Add_three_numbers  
    Read number1, number2, number3  
    total = number1 + number2 + number3  
    Print total  
END
```

There are a number of points to consider in this solution algorithm:

- 1 A name has been given to the algorithm, namely Add_three_numbers. Algorithm names should briefly describe the function of the algorithm, and are usually expressed as a single verb followed by a two-word object.
- 2 An END statement at the end of the algorithm indicates that the algorithm is complete.
- 3 All processing steps between the algorithm name and the END statement have been indented for readability.
- 4 Each processing step in the defining diagram relates directly to one or more statements in the algorithm. For instance, 'Read three numbers' in the defining diagram becomes 'Read number1, number2, number3' in the algorithm; and 'Add number together' becomes 'total = number1 + number2 + number3'.

Now that the algorithm is complete, you should desk check the solution and then translate it into a programming language. (Desk checking is covered in Section 3.3.)

EXAMPLE 3.5 Solution algorithm for Example 3.2

A program is required to prompt the terminal operator for the maximum and minimum temperature readings on a particular day, accept those readings as integers, and calculate and display to the screen the average temperature, calculated by (maximum temperature + minimum temperature)/2.

A Defining diagram

Input	Processing	Output
max_temp	Prompt for temperatures	avg_temp
min_temp	Get temperatures Calculate average temperature Display average temperature	

Using pseudocode, a simple calculation and the sequence control structure, the algorithm can be expressed as follows:

B Solution algorithm

```
Find_average_temperature
  Prompt operator for max_temp, min_temp
  Get max_temp, min_temp
  avg_temp = (max_temp + min_temp)/2
  Output avg_temp to the screen
END
```

EXAMPLE 3.6 Solution algorithm for Example 3.3

A program is required to read from the screen the length and width of a rectangular house block, and the length and width of the rectangular house that has been built on the block. The algorithm should then compute and display the mowing time required to cut the grass around the house, at the rate of two square metres per minute.

A Defining diagram

Input	Processing	Output
block_length	Prompt for block measurements	mowing_time
block_width	Get block measurements	
house_length	Prompt for house measurements	
house_width	Get house measurements Calculate mowing area Calculate mowing time	

The actions to be carried out in this algorithm are listed sequentially in the processing component of the defining diagram. At this stage, the processing steps still only describe the steps to be performed, in their correct order.

B Solution algorithm

```
Calculate_mowing_time
    Prompt operator for block_length, block_width
    Get block_length, block_width
    block_area = block_length * block_width
    Prompt operator for house_length, house_width
    Get house_length, house_width
    house_area = house_length * house_width
    mowing_area = block_area - house_area
    mowing_time = mowing_area/2
    Output mowing_time to screen
```

```
END
```

3.3 Checking the solution algorithm

After a solution algorithm has been established, it must be tested for correctness. This step is necessary because most major logic errors occur during the development of the algorithm, and if not detected, these errors can be passed on to the program. It is much easier to detect errors in pseudocode than in the corresponding program code. This is because once programming begins, you usually assume that the logic of the algorithm is correct. Then, when you

detect errors, your attention is focused on the individual lines of code to identify the problems, rather than on the initial logic expressed in the algorithm. It is often too difficult to step back and analyse the program as a whole. As a result, many frustrating hours can be wasted during testing, which could have been avoided by spending a few minutes desk checking the solution algorithm.

Desk checking involves tracing through the logic of the algorithm with some chosen test data. That is, you walk through the logic of the algorithm exactly as a computer would, keeping track of all major variable values on a sheet of paper. This playing computer not only helps to detect errors early, but also helps you to become familiar with the way the program runs. The closer you are to the execution of the program, the easier it is to detect errors.

Selecting test data

When selecting test data to desk check an algorithm, you must look at the program specification and choose simple test cases only, based on the requirements of the specification, not the algorithm. By doing this, you will still be able to concentrate on *what* the program is supposed to do, not *how*.

To desk check the algorithm, you need only a few simple test cases that will follow the major paths of the algorithm logic. A much more comprehensive test will be performed once the algorithm has been coded into a programming language.

Steps in desk checking an algorithm

There are six simple steps to follow when desk checking an algorithm:

- 1 Choose simple input test cases that are valid. Two or three test cases are usually sufficient.
- 2 Establish what the expected result should be for each test case. This is one of the reasons for choosing simple test data in the first place: it is much easier to determine the total of 10, 20 and 30 than 3.75, 2.89 and 5.31!
- 3 Make a table on a piece of paper of the relevant variable names within the algorithm.
- 4 Walk the first test case through the algorithm, line by line, keeping a step-by-step record of the contents of each variable in the table as the data passes through the logic.
- 5 Repeat the walk-through process using the other test data cases, until the algorithm has reached its logical end.
- 6 Check that the expected result established in Step 2 matches the actual result developed in Step 5.

By desk checking an algorithm, you are attempting to detect early errors. It is a good idea for someone other than the author of the solution algorithm to design the test data for the program, as they are not influenced by the program logic. Desk checking will eliminate most errors, but it still cannot prove that the algorithm is 100% correct!

Now let us desk check each of the algorithms developed in this chapter.

EXAMPLE 3.7 Desk check of Example 3.1

A Solution algorithm

```
Add_three_numbers  
1   Read number1, number2, number3  
2   total = number1 + number2 + number3  
3   Print total  
END
```

B Desk checking

- 1 Choose two sets of input test data. The three numbers selected will be 10, 20 and 30 for the first test case and 40, 41 and 42 for the second.

	First data set	Second data set
number1	10	40
number2	20	41
number3	30	42

- 2 Establish the expected result for each test case.

	First data set	Second data set
total	60	123

- 3 Set up a table of relevant variable names, and pass each test data set through the solution algorithm, statement by statement. Line numbers have been used to identify each statement within the program.

Statement number	number1	number2	number3	total
First pass				
1 	10	20	30	
2				60
3 				print
Second pass				
1	40	41	42	
2				123
3				print

- 4** Check that the expected results (60 and 123) match the actual results (the total column in the table).

This desk check, which should take no more than a few minutes, indicates that the algorithm is correct. You can now proceed to code the algorithm into a programming language. Note that if, at the end of a desk check, the actual results do not match the expected results, the solution algorithm probably contains a logic error. In this case, the programmer needs to go back to the solution algorithm, fix the error, then desk check the algorithm again. See Example 3.10.

EXAMPLE 3.8 Desk check of Example 3.2

A *Solution algorithm*

```
Find_average_temperature  
1   Prompt operator for max_temp, min_temp  
2   Get max_temp, min_temp  
3   avg_temp = (max_temp + min_temp)/2  
4   Output avg_temp to the screen  
END
```

B *Desk checking*

- 1** Choose two sets of input test data. The max_temp and min_temp values will be 30 and 10 for the first case, and 40 and 20 for the second.

	First data set	Second data set
max_temp	30	40
min_temp	10	20

- 2** Establish the expected result for each test case.

	First data set	Second data set
avg_temp	20	30

- 3** Set up a table of variable names, and pass each test data set through the solution algorithm, statement by statement, using the algorithm line numbers as indicated.

Statement number	max_temp	min_temp	avg_temp
First pass			
1.2 	30	10	
3			20
4			output
Second pass			
1.2	40	20	
3			30
4			output

- 4 Check that the expected results in Step 2 match the actual results in Step 3.

EXAMPLE 3.9 Desk check of Example 3.3

A Solution algorithm

```

Calculate_mowing_time
1   Prompt operator for block_length, block_width
2   Get block_length, block_width
3   block_area = block_length * block_width
4   Prompt operator for house_length, house_width
5   Get house_length, house_width
6   house_area = house_length * house_width
7   mowing_area = block_area - house_area
8   mowing_time = mowing_area/2
9   Output mowing_time to screen
END

```

B Desk checking

- 1 Input data:

	First data set	Second data set
block_length	30	40
block_width	30	20
house_length	20	20
house_width	20	10

2 Expected results:

	First data set	Second data set
mowing_time	250 minutes	300 minutes

3 Set up a table of variable names and pass each test data set through the solution algorithm, statement by statement.

Statement number	block_length	block_width	house_length	house_width	block_area	house_area	mowing_area	mowing_time
First pass								
1,2	30	30						
3					900			
4,5			20	20				
6						400		
7							500	
8								250
9								output
Second pass								
1,2	40	20						
3					800			
4,5			20	10				
6						200		
7							600	
8								300
9								output

4 Check that the expected results match the actual results. Yes, the expected result for each set of data matches the calculated result.

EXAMPLE 3.10 Desk check of Example 3.3, which now contains a logic error

A Solution algorithm

```
Calculate_mowing_time
1   Prompt operator for block_length, block_width
2   Get block_length, block_width
3   block_area = block_length * block_width
4   Prompt operator for house_length, house_width
5   Get house_length, house_width
6   house_area = block_length * block_width
7   mowing_area = block_area - house_area
8   mowing_time = mowing_area / 2
9   Output mowing_time to screen
END
```

B Desk checking

- 1 Input data:

	First data set	Second data set
block_length	30	40
block_width	30	20
house_length	20	20
house_width	20	10

- 2 Expected results:

	First data set	Second data set
mowing_time	250 minutes	300 minutes

- 3 Set up a table of variable names and pass each test data set through the solution algorithm, statement by statement.

Statement number	block_length	block_width	house_length	house_width	block_area	house_area	mowing_area	mowing_time
First pass								
1,2	30	30						
3					900			
4,5			20	20				
6						900		
7							0	
8								0
9								output
Second pass								
1,2	40	20						
3					800			
4,5			20	10				
6						800		
7							0	
8								0
9								output

- 4 Check that the expected results match the actual results. Here, you can see that the calculation for house_area in line 6 is incorrect, because when house_area is subtracted from block_area in line 7, the result is zero, which cannot be right. The algorithm needs to be adjusted, so that the statement

house_area = block_length * block_width

is changed to

house_area = house_length * house_width

Another desk check would establish that the algorithm is now correct.

Chapter summary

The first section of this chapter was devoted to methods of analysing and defining a programming problem. You must fully understand a problem before you can attempt to find a solution. The method suggested was to analyse the actual words used in the specification with the aim of dividing the problem into three separate components: **input**, output and processing. Several examples were explored and the use of a defining diagram was established. It was emphasised that the processing steps should list **what** tasks need to be performed, rather than *how* they are to be accomplished.

The second section was devoted to the establishment of a solution algorithm. After the initial analysis of the problem, you must attempt to find a solution and express this solution as an algorithm. To do this, you must use the defining diagram, correct pseudocode statements and the three basic control structures. Only algorithms using the sequence control structure were used as examples.

The third section was concerned with checking the algorithm for correctness. A method of playing computer by tracing through the algorithm step by step was introduced, with examples to previous problems given.

Programming problems

To solve the following problems, you will need to:

- define the problem by constructing a defining diagram
- create a solution algorithm using pseudocode
- desk check the solution algorithm using two valid test cases.

- 1 Construct an algorithm that will prompt an operator to input three characters, receive those three characters, and display a welcoming message to the screen such as 'Hello xxx! We hope you have a nice day'.
- 2 You require an algorithm that will receive two integer items from a terminal operator, and display to the screen their sum, difference, product and quotient.
- 3 You require an algorithm that will receive an integer from the screen, add 5 to it, double it, subtract 7 from it, and display the final number to the screen.
- 4 You require an algorithm that will read in a tax rate (as a percentage) and the prices of five items. The program is to calculate the total price, before tax, of the items, then the tax payable on those items. The tax payable is calculated by applying the tax rate percentage to the total price. Print the total price and the tax payable as output.
- 5 You require an algorithm to read in one customer's account balance at the beginning of the month, a total of all withdrawals for the month, and a total of all

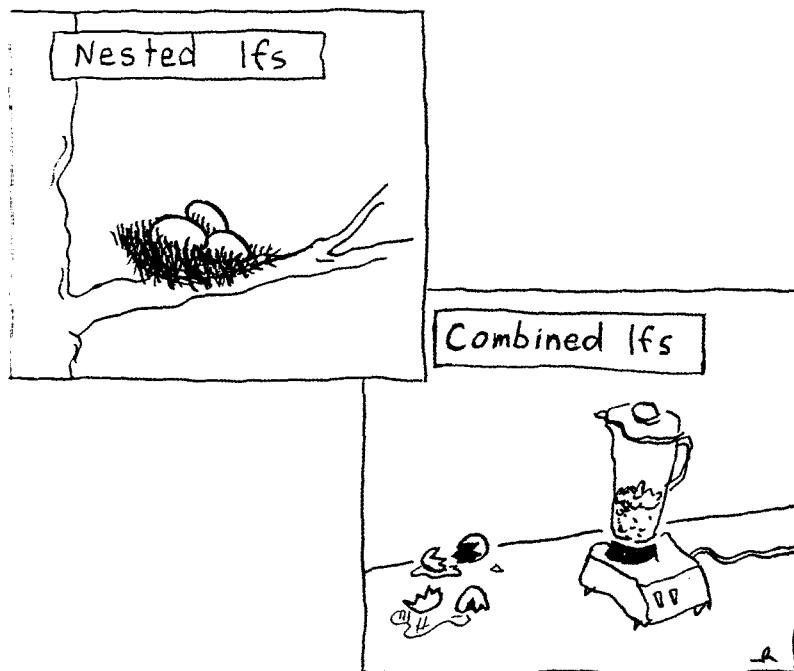
deposits made during the month. A federal tax charge of 1% is applied to all transactions made during the month. The program is to calculate the account balance at the end of the month by (1) subtracting the total withdrawals from the account balance at the beginning of the month, (2) adding the total deposits to this new balance, (3) calculating the federal tax (1% of total transactions – that is, total withdrawals + total deposits), and (4) subtracting this federal tax from the new balance. After these calculations, print the final end-of-month balance.

- 6** You require a program to read in the values from an employee's time sheet, and calculate and print the weekly pay owing to that employee. The values read in are the total number of regular hours worked, the total overtime hours and the hourly wage rate. Weekly pay is calculated as payment for regular hours worked, plus payment for overtime hours worked. Payment for regular hours worked is calculated as (wage rate times regular hours worked); payment for overtime hours worked is calculated as (wage rate times overtime hours worked times 1.5).

Chapter

Selection control structures

4



Objectives

- To elaborate on the uses of simple selection, multiple selection and nested selection in algorithms
- To introduce the case construct in pseudocode
- To develop algorithms using variations of the selection control structure

Outline

- ↳ 1 The selection control structure
 - ↳ 2 Algorithms using selection
 - ↳ 3 The case structure
- Chapter summary
Programming problems

4.1 The selection control structure

The selection control structure was introduced in Chapter 2, as the second construct in the Structure Theorem. This structure represents the decision-making abilities of the computer. That is, you can use the selection control structure in pseudocode to illustrate a choice between two or more actions, depending on whether a condition is true or false. The condition in the IF statement is based on a comparison of two items, and is usually expressed with one of the following relational operators:

- < less than
- > greater than
- = equal to
- <= less than or equal to
- >= greater than or equal to
- <> not equal to

There are a number of variations of the selection structure, as follows.

1 Simple selection (simple IF statement)

Simple selection occurs when a choice is made between two alternate paths, depending on the result of a condition being true or false. The structure is represented in pseudocode using the keywords IF, THEN, ELSE and ENDIF. For example:

```
IF account_balance < $300 THEN
    service_charge = $5.00
ELSE
    service_charge = $2.00
ENDIF
```

Only one of the THEN or ELSE paths will be followed, depending on the result of the condition in the IF clause.

2 Simple selection with null false branch (null ELSE statement)

The null ELSE structure is a variation of the simple IF structure. It is used when a task is performed only when a particular condition is true. If the condition is false, then no processing will take place and the IF statement will be bypassed. For example:

```
IF student_attendance = part_time THEN
    add 1 to part_time_count
ENDIF
```

In this case, the part_time_count field will be altered only if the student's attendance pattern is part-time.

3 Combined selection (combined IF statement)

A combined IF statement is one that contains multiple conditions, each connected with the logical operators AND or OR. If the connector AND is used to combine the conditions, then *both* conditions must be true for the combined condition to be true. For example:

```
IF student_attendance = part_time  
AND student_gender = female THEN  
    add 1 to female_part_time_count  
ENDIF
```

In this case, each student record will undergo two tests. Only those students who are female *and* who attend part-time will be selected, and the variable `female_part_time_count` will be incremented. If either condition is found to be false, the counter will remain unchanged.

If the connector OR is used to combine any two conditions, then only *one* of the conditions needs to be true for the combined condition to be considered true. If neither condition is true, the combined condition is considered false. Changing the AND in the above example to OR dramatically changes the outcome from the processing of the IF statement.

```
IF student_attendance = part_time  
OR student_gender = female THEN  
    add 1 to female_part_time_count  
ENDIF
```

In this example, if either or both conditions are found to be true, the combined condition will be considered true. That is, the counter will be incremented:

- 1 if the student is part-time, regardless of gender
or
- 2 if the student is female, regardless of attendance pattern.

Only those students who are not female and not part-time will be ignored. So, `female_part_time_count` will contain the total count of female part-time students, male part-time students and female full-time students. As a result, `female_part_time_count` is no longer a meaningful name for this variable. You must fully understand the processing that takes place when combining conditions with the AND or OR logical operators.

More than two conditions can be linked together with the AND or OR operators. However, if both operators are used in the one IF statement, parentheses must be used to avoid ambiguity. Look at the following example:

```
IF record_code = '23'  
OR update_code = delete  
AND account_balance = zero THEN  
    delete customer record  
ENDIF
```

The logic of this statement is confusing. It is uncertain whether the first two conditions should be grouped together and operated on first, or the second two conditions should be grouped together and operated on first. Pseudocode algorithms should never be ambiguous. There are no precedence rules for logical operators in pseudocode, but there are precedence rules in most programming languages. Therefore, parentheses must be used in pseudocode to avoid ambiguity as to the meaning intended, as follows:

```
IF (record_code = '23'  
    OR update_code = delete)  
    AND account_balance = zero THEN  
        delete customer record  
    ENDIF
```

The IF statement is now no longer ambiguous, and it is clear as to what conditions are necessary for the customer record to be deleted: the record will only be deleted if the account balance equals zero and either the record code = 23 or the update code = delete.

The NOT operator

The NOT operator can be used for the logical negation of a condition, as follows:

```
IF NOT (record_code = '23') THEN  
    update customer record  
ENDIF
```

Here, the IF statement will be executed for all record codes other than code '23' – that is, for record codes *not* equal to '23'.

Note that the AND and OR operators can also be used with the NOT operator, but great care must be taken and parentheses used to avoid ambiguity, as follows:

```
IF NOT (record_code = '23'  
    AND update_code = delete) THEN  
    update customer record  
ENDIF
```

Here, the customer record will only be updated if the record code is not equal to '23' *and* the update code is not equal to delete.

4 Nested selection (nested IF statement)

Nested selection occurs when the word IF appears more than once within an IF statement. Nested IF statements can be classified as linear or non-linear.

Linear nested IF statements

The linear nested IF statement is used when a field is being tested for various values and a different action is to be taken for each value.

This form of nested IF is called linear, because each ELSE immediately follows the IF condition to which it corresponds. Comparisons are made until a true condition is encountered, and the specified action is executed until the

first two
e second
udocode
rules for
host pro-
ocode to

to what
e record
e record

follows:

n code
oper-
uity, as

equal

in an
ear.

rious

y fol-
ntil a
il the

~~text~~ ELSE statement is reached. Linear nested IF statements should be indented for readability, with each IF, ELSE and corresponding ENDIF aligned. For example:

```
IF record_code = 'A' THEN
    increment counter_A
ELSE
    IF record_code = 'B' THEN
        increment counter_B
    ELSE
        IF record_code = 'C' THEN
            increment counter_C
        ELSE
            increment error_counter
        ENDIF
    ENDIF
ENDIF
```

Note that there are an equal number of IF, ELSE and ENDIF statements, that each ELSE and ENDIF statement is positioned so that it corresponds with its matching IF statement, and that the correct indentation makes it easy to read and understand. A block of nested IF statements like this is sometimes referred to as 'cascading IF statements', as they cascade like a waterfall across the page.

Non-linear nested IF statements

A non-linear nested IF occurs when a number of different conditions need to be satisfied before a particular action can occur. It is termed non-linear because the ELSE statement may be separated from the IF statement with which it is paired. Indentation is once again important when expressing this form of selection in pseudocode. Each ELSE and ENDIF statement should be aligned with the IF condition to which it corresponds.

For instance:

```
IF student_attendance = part_time THEN
    IF student_gender = female THEN
        IF student_age > 21 THEN
            add 1 to mature_female_pt_students
        ELSE
            add 1 to young_female_pt_students
        ENDIF
    ELSE
        add 1 to male_pt_students
    ENDIF
ELSE
    add 1 to full_time_students
ENDIF
```

Note that there are an equal number of IF conditions as ELSE and ENDIF statements. Using correct indentation helps to see which pair of IF, ELSE and

ENDIF statements match. However, non-linear nested IF statements may contain logic errors that are difficult to correct, so they should be used sparingly in pseudocode. If possible, replace a series of non-linear nested IF statements with a combined IF statement. This replacement is possible in pseudocode because two consecutive IF statements act like a combined IF statement which uses the AND operator. Take as an example the following non-linear nested IF statement:

```
IF student_attendance = part_time THEN
    IF student_age > 21 THEN
        increment mature_pt_student
    ENDIF
ENDIF
```

This can be written as a combined IF statement:

```
IF student_attendance = part_time
AND student_age > 21 THEN
    increment mature_pt_student
ENDIF
```

The same outcome will occur for both pseudocode expressions, but the format of the latter is preferred, if the logic allows it, simply because it is easier to understand.

4.2 Algorithms using selection

Let us look at some programming examples that use the selection control structure. In each example, the problem will be defined, a solution algorithm will be developed and the algorithm will be manually tested. To help define the problem, the processing verbs in each example have been underlined.

EXAMPLE 4.1 Read three characters

Design an algorithm that will prompt a terminal operator for three characters, accept those characters as input, sort them into ascending sequence and output them to the screen.

A Defining diagram

Input	Processing	Output
char_1	Prompt for characters	char_1
char_2	Accept three characters	char_2
char_3	Sort three characters Output three characters	char_3

B Solution algorithm

The solution algorithm requires a series of IF statements to sort the three characters into ascending sequence.

```
Read_three_characters
1   Prompt the operator for char_1, char_2, char_3
2   Get char_1, char_2, char_3
3   IF char_1 > char_2 THEN
        temp = char_1
        char_1 = char_2
        char_2 = temp
    ENDIF
4   IF char_2 > char_3 THEN
        temp = char_2
        char_2 = char_3
        char_3 = temp
    ENDIF
5   IF char_1 > char_2 THEN
        temp = char_1
        char_1 = char_2
        char_2 = temp
    ENDIF
6   Output to the screen char_1, char_2, char_3
END
```



In this solution, most of the logic of the algorithm is concerned with the sorting of the three characters into alphabetic sequence. This sorting is carried out with the use of pseudocode that 'swaps' two items, as follows:

```
temp = char_1
char_1 = char_2
char_2 = temp
```



Here, the values in the variables char_1 and char_2 are 'swapped', with the use of the temporary variable, temp. Pseudocode such as this must be written carefully to ensure that items are not lost in the shuffle.

To make the algorithm easier to read, this sorting logic could have been performed in a module, as demonstrated in Chapter 8.

C Desk checking

Two sets of valid characters will be used to check the algorithm; the characters k, b and g as the first set, and z, s and a as the second.

1 Input data

	First data set	Second data set
char_1	k	z
char_2	b	s
char_3	g	a

2 Expected results

	First data set	Second data set
char_1	b	a
char_2	g	s
char_3	k	z

3 Desk check table

Line numbers have been used to identify each statement within the program. Note that when desk checking the logic, each IF statement is treated as a single statement.

Statement number	char_1	char_2	char_3	temp
First pass				
1,2	k	b	g	
3	b	k		k
4		g	k	k
5				
6	output	output	output	
Second pass				
1,2	z	s	a	
3	s	z		z
4		a	z	z
5	a	s		s
6	output	output	output	

EXAMPLE 4.2 Process customer record

A program is required to read a customer's name, a purchase amount and a tax code. The tax code has been validated and will be one of the following:

- 0 tax exempt (0%)
- 1 state sales tax only (3%)
- 2 federal and state sales tax (5%)
- 3 special sales tax (7%)

The program must then compute the sales tax and the total amount due, and print the customer's name, purchase amount, sales tax and total amount due.

A Defining diagram

Input	Processing	Output
cust_name	Read customer details	cust_name
purch_amt	Compute sales tax	purch_amt
tax_code	Compute total amount	sales_tax
	Print customer details	total_amt

B Solution algorithm

The solution algorithm requires a linear nested IF statement to calculate the sales tax.

```

Process_customer_record
1   Read cust_name, purch_amt, tax_code
2   IF tax_code = 0 THEN
        sales_tax = 0
    ELSE
        IF tax_code = 1 THEN
            sales_tax = purch_amt * 0.03
        ELSE
            IF tax_code =2 THEN
                sales_tax = purch_amt * 0.05
            ELSE
                sales_tax = purch_amt * 0.07
            ENDIF
        ENDIF
    ENDIF
3   total_amt = purch_amt + sales_tax
4   Print cust_name, purch_amt, sales_tax, total_amt
END

```

C Desk checking

Two sets of valid input data for purchase amount and tax code will be used to check the algorithm.

1 Input data

	First data set	Second data set
purch_amt	\$10.00	\$20.00
tax_code	0	2

2 Expected results

	First data set	Second data set
sales_tax	0	\$1.00
total_amt	\$10.00	\$21.00

Note that when desk checking the logic, the whole linear nested IF statement (13 lines of pseudocode) is counted as a single pseudocode statement.

3 Desk check table

Statement number	purch_amt	tax_code	sales_tax	total_amt
First pass				
1	\$10.00	0		
2			0	
3				\$10.00
4	print		print	print
Second pass				
1	\$20.00	2		
2			\$1.00	
3				\$21.00
4	print		print	print

As the expected result for the two test cases matches the calculated result, the algorithm is correct.

EXAMPLE 4.3 Calculate employee's pay

A program is required by a company to read an employee's number, pay rate and the number of hours worked in a week. The program is then to validate the pay rate and the hours worked fields and, if valid, compute the employee's weekly pay and print it along with the input data.

Validation: According to the company's rules, the maximum hours an employee can work per week is 60 hours, and the maximum hourly rate is \$25.00 per hour. If the hours worked field or the hourly rate field is out of range, the input data and an appropriate message is to be printed and the employee's weekly pay is not to be calculated.

Weekly pay calculation: Weekly pay is calculated as hours worked times pay rate. If more than 35 hours are worked, payment for the overtime hours worked is calculated at time-and-a-half.

A Defining diagram

Input	Processing	Output
emp_no	Read employee details	emp_no
pay_rate	Validate input fields	pay_rate
hrs_worked	Calculate employee pay Print employee details	hrs_worked emp_weekly_pay error_message

B Solution algorithm

The solution to this problem will require a series of simple IF and nested IF statements. First, the variables 'pay_rate' and 'hrs_worked' must be validated, and if either is found to be out of range, an appropriate message is to be placed into a variable called 'error_message'.

The employee's weekly pay is only to be calculated if the input variables 'pay_rate' and 'hrs_worked' are valid, so another variable, 'valid_input_fields', will be used to indicate to the program whether or not these input fields are valid.

Boolean variables

The variable valid_input_fields is a Boolean variable – that is, it may contain only one of two possible values (true or false). When using the IF statement with a Boolean variable, the IF statement can be simplified in pseudocode. For example, the following pseudocode

```
IF valid_input_fields = true THEN
    statement
ENDIF
```

can be simplified to imply '= true', and so can be written as:

```
IF valid_input_fields THEN
    statement
ENDIF
```

Similarly, if we want to test if valid_input_fields is false, we can say in pseudocode:

```
IF NOT valid_input_fields THEN
    statement
ENDIF
```

The variable valid_input_fields acts as an internal switch or flag to the program. It will initially be set to true, and will be assigned the value false if one of the input fields is found to be invalid. The employee's weekly pay will be calculated only if valid_input_fields is true.

```
Compute_employee_pay
1      Set valid_input_fields to true
2      Set error_message to blank
3      Read emp_no, pay_rate, hrs_worked
4      IF pay_rate > $25 THEN
            error_message = 'Pay rate exceeds $25.00'
            Print emp_no, pay_rate, hrs_worked, error_message
            valid_input_fields = false
        ENDIF
5      IF hrs_worked > 60 THEN
            error_message = 'Hours worked exceeds 60'
            Print emp_no, pay_rate, hrs_worked, error_message
            valid_input_fields = false
        ENDIF
6      IF valid_input_fields THEN
            IF hrs_worked <= 35 THEN
                emp_weekly_pay = pay_rate * hrs_worked
            ELSE
                overtime_hrs = hrs_worked - 35
                overtime_pay = overtime_hrs * pay_rate * 1.5
                emp_weekly_pay = (pay_rate * 35) + overtime_pay
            ENDIF
            Print emp_no, pay_rate, hrs_worked, emp_weekly_pay
        ENDIF
    END
```

In this solution, there are two separate functions to be performed in the algorithm: the validation of the input data, and the calculation and printing of the employee's weekly pay. These two tasks could have been separated into modules before the algorithm was developed in pseudocode (see Chapter 8).

C Desk checking

Two sets of valid input data for pay rate and hours worked will be used to check this algorithm.

1 Input data

	First data set	Second data set
pay_rate	\$10.00	\$40.00
hrs_worked	40	35

to the
also if
y will

algo-
of the
mod-
l.

ed to

2 Expected results

	First data set	Second data set
pay_rate	\$10.00	\$40.00
hrs_worked	40	35
emp_weekly_pay	\$425.00	-
error_message	blank	Pay rate exceeds \$25.00

3 Desk check table

Statement number	pay_rate	hrs_worked	over_time hrs	over_time pay	emp_weekly_pay	valid_input_fields	error_message	Print
First pass								
1						true		
2							blank	
3	\$10.00	40						
4								
5								
6			5	75.00	425.00			Print fields
Second pass								
1						true		
2							blank	
3	\$40.00	35						
4						false	Pay rate exceeds \$25.00	Print message
5								
6								

4.3 The case structure

The case control structure in pseudocode is another way of expressing a linear nested IF statement. It is used in pseudocode for two reasons: it can be directly translated into many high-level languages, and it makes the pseudocode easier to write and understand. Nested IFs often look cumbersome in pseudocode and depend on correct structure and indentation for readability. Let us look at the example used earlier in this chapter:

```

IF record_code = 'A' THEN
    increment counter_A
ELSE
    IF record_code = 'B' THEN
        increment counter_B
    ELSE
        IF record_code = 'C' THEN
            increment counter_C
        ELSE
            increment error_counter
        ENDIF
    ENDIF
ENDIF

```

This linear nested IF structure can be replaced with a case control structure. Case is not really an additional control structure. It simplifies the basic selection control structure and extends it from a choice between two values to a choice from multiple values. In one case structure, several alternative logical paths can be represented. In pseudocode, the keywords CASE OF and ENDCASE serve to identify the structure, with the multiple values indented, as follows:

```

CASE OF single variable
    value_1 : statement block_1
    value_2 : statement block_2

    .
    .
    .

    value_n : statement block_n
    value_other : statement block_other
ENDCASE

```

The path followed in the case structure depends on the value of the variable specified in the CASE OF clause. If the variable contains value_1, statement block_1 is executed; if it contains value_2, statement block_2 is executed, and so on. The value_other is included in the event that the variable contains none of the listed values. We can now rewrite the above linear nested IF statement with a case statement, as follows:

```

CASE OF record_code
    'A'      : increment counter_A
    'B'      : increment counter_B
    'C'      : increment counter_C
    other   : increment error_counter
ENDCASE

```

In both forms of pseudocode, the processing logic is exactly the same. However, the case solution is much more readable.

Let us now look again at Example 4.2. The solution algorithm for this example was earlier expressed as a linear nested IF statement, but it could equally have been expressed as a CASE statement.

EXAMPLE 4.4 Process customer record

A program is required to read a customer's name, a purchase amount and a tax code. The tax code has been validated and will be one of the following:

- 0 tax exempt (0%)
- 1 state sales tax only (3%)
- 2 federal and state sales tax (5%)
- 3 special sales tax (7%)

The program must then compute the sales tax and the total amount due, and print the customer's name, purchase amount, sales tax and total amount due.

A Defining diagram

Input	Processing	Output
cust_name	Read customer details	cust_name
purch_amt	Compute sales tax	purch_amt
tax_code	Compute total amount	sales_tax
	Print customer details	total_amt

B Solution algorithm

The solution algorithm will be expressed using a CASE statement.

```
Process_customer_record
1      Read cust_name, purch_amt, tax_code
2      CASE OF tax_code
          0 : sales_tax = 0
          1 : sales_tax = purch_amt * 0.03
          2 : sales_tax = purch_amt * 0.05
          3 : sales_tax = purch_amt * 0.07
      ENDCASE
3      total_amt = purch_amt + sales_tax
4      Print cust_name, purch_amt, sales_tax, total_amt
END
```

C Desk checking

Two sets of valid input data for purchase amount and tax code will be used to check the algorithm. Note that the case structure serves as a single pseudocode statement.

1 Input data

	First data set	Second data set
purch_amt	\$10.00	\$20.00
tax_code	0	2

2 Expected results

	First data set	Second data set
sales_tax	0	\$1.00
total_amt	\$10.00	\$21.00

3 Desk check table

Statement number	purch_amt	tax_code	sales_tax	total_amt
First pass				
1	\$10.00	0		
2			0	
3				\$10.00
4	print		print	print
Second pass				
1	\$20.00	2		
2			\$1.00	
3				\$21.00
4	print		print	print

As the expected result matches the actual result, the algorithm is shown to be correct.

Chapter summary

This chapter covered the selection control structure in detail. Descriptions and pseudocode examples were given for simple selection, null ELSE, combined IF and nested IF statements. Several solution algorithms that used the selection structure were developed.

The case structure was introduced as a means of expressing a linear nested IF statement in a simpler and more concise form. Case is available in many high-level languages, and so is a useful construct in pseudocode.

Programming problems

Construct a solution algorithm for the following programming problems. Your solution should contain:

- a defining diagram
 - a pseudocode algorithm
 - a desk check of the algorithm.
- 1 Design an algorithm that will receive two integer items from a terminal operator, and display to the screen their sum, difference, product and quotient. Note that the quotient calculation (first integer divided by second integer) is only to be performed if the second integer does not equal zero.
 - 2 Design an algorithm that will read two numbers and an integer code from the screen. The value of the integer code should be 1, 2, 3 or 4. If the value of the code is 1, compute the sum of the two numbers. If the code is 2, compute the difference (first minus second). If the code is 3, compute the product of the two numbers. If the code is 4, and the second number is not zero, compute the quotient (first divided by second). If the code is not equal to 1, 2, 3 or 4, display an error message. The program is then to display the two numbers, the integer code and the computed result to the screen.
 - 3 Design an algorithm that will prompt an operator for a student's serial number and the student's exam score out of 100. Your program is then to match the exam score to a letter grade and print the grade to the screen. Calculate the letter grade as follows:

Exam score	Assigned grade
90 and above	A
80-89	B
70-79	C
60-69	D
below 60	F

- 4** Design an algorithm that will receive the weight of a parcel and determine the delivery charge for that parcel. Calculate the charges as follows:

Parcel weight (kg)	Cost per kg (\$)
<2.5 kg	\$3.50 per kg
2.5-5 kg	\$2.85 per kg
>5 kg	\$2.45 per kg

- 5** Design an algorithm that will prompt a terminal operator for the price of an article and a pricing code. Your program is then to calculate a discount rate according to the pricing code and print to the screen the original price of the article, the discount amount and the new discounted price. Calculate the pricing code and accompanying discount amount as follows:

Pricing code	Discount rate
H	50%
F	40%
T	33%
Q	25%
Z	0%

If the pricing code is Z, the words 'No discount' are to be printed on the screen. If the pricing code is not H, F, T, Q or Z, the words 'Invalid pricing code' are to be printed.

- 6** An architect's fee is calculated as a percentage of the cost of a building. The fee is made up as follows:

8% of the first \$5000.00 of the cost of a building and
3% on the remainder if the remainder is less than or equal to \$80 000.00 or
2% on the remainder if the remainder is more than \$80 000.00.

Design an algorithm that will accept the cost of a building and calculate and display the architect's fee.

- 7** A home mortgage authority requires a deposit on a home loan according to the following schedule:

Loan (\$)	Deposit
less than \$25 000	5% of loan value
\$25 000-\$49 999	\$1250 + 10% of loan over \$25 000
\$50 000-\$100 000	\$5000 + 25% of loan over \$50 000

Loans in excess of \$100 000 are not allowed. Design an algorithm that will read a loan amount and compute and print the required deposit.

- 8** Design an algorithm that will receive a date in the format dd/mm/yyyy (for example, 21/07/2003) and validate it as follows:

- i the month must be in the range 1-12, and
- ii the day must be in the range of 1-31 and acceptable for the corresponding month. (Don't forget a leap year check for February.)

- 9** The tax payable on taxable incomes for employees in a certain country is set out in the following table:

Taxable income	Tax payable
From \$1.00-\$4461.99	Nil
From \$4462.00-\$17 893.99	Nil plus 30 cents for each \$ in excess of \$4462.00
From \$17 894.00-\$29 499.99	\$4119.00 plus 35 cents for each \$ in excess of \$17 894.00
From \$29 500.00-\$45 787.99	\$8656.00 plus 46 cents for each \$ in excess of \$29 500.00
\$45 788.00 and over	\$11 179.00 plus 60 cents for each \$ in excess of \$45 788.00

Design an algorithm that will read as input the taxable income amount and calculate and print the tax payable on that amount.

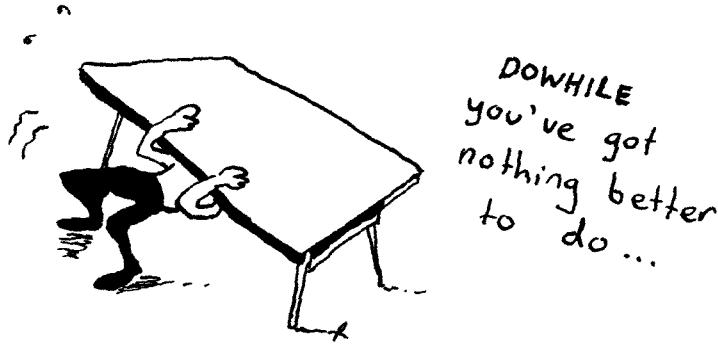
- 10** A transaction record on a sales commission file contains the retail price of an item sold, a transaction code that indicates the sales commission category to which an item can belong, and the employee number of the person who sold the item. The transaction code can contain the values S, M or L, which indicate that the percentage commission will be 5%, 7% or 10%, respectively. Construct an algorithm that will read a record on the file, calculate the commission owing for that record, and print the retail price, commission and employee number.

Chapter

5

Repetition control structures

" DESK CHECKING "



Objectives

- To develop algorithms that use the DOWHILE and REPEAT...UNTIL control structures
- To introduce a pseudocode structure for counted repetition loops
- To develop algorithms using variations of the repetition construct

Outline

- 5.1 Repetition using the DOWHILE structure
 - 5.2 Repetition using the REPEAT...UNTIL structure
 - 5.3 Counted repetition
- Chapter summary
Programming problems

5.1 Repetition using the DOWHILE structure

The solution algorithms developed so far have one characteristic in common: they show the program logic required to process just one set of input values. However, most programs require the same logic to be repeated for several sets of data. The most efficient way to deal with this situation is to establish a looping structure in the algorithm that will cause the processing logic to be repeated a number of times.

There are three different ways that a set of instructions can be repeated, and each way is determined by where the decision to repeat is placed:

- at the beginning of the loop (leading decision loop)
- at the end of the loop (trailing decision loop)
- a counted number of times (counted loop).

Leading decision loop

In Chapter 2, the DOWHILE construct was introduced as the pseudocode representation of a repetitive loop. Its format is:

```
DOWHILE condition p is true
    statement block
    ENDDO
```

The DOWHILE construct is a leading decision loop – that is, the condition is tested before any statements are executed. In the above DOWHILE loop, the following processing takes place:

- 1 The logical condition p is tested.
- 2 If condition p is found to be true, the statements within the statement block are executed once. The delimiter ENDDO then triggers a return of control to the retesting of condition p.
- 3 If condition p is still true, the statements are executed again, and so the repetition process continues until the condition is found to be false.
- 4 If condition p is found to be false, control passes to the next statement after the delimiter ENDDO and no further processing takes place within the loop.

There are two important considerations about which you must be aware before designing a DOWHILE loop. First, the testing of the condition is at the beginning of the loop. This means that the programmer may need to perform some initial processing to adequately set up the condition before it can be tested.

Second, the only way to terminate the loop is to render the DOWHILE condition false. This means you must set up some process within the statement block that will eventually change the condition so that the condition becomes false. Failure to do this results in an endless loop.

Using DOWHILE to repeat a set of instructions a known number of times

When a set of instructions is repeated a specific number of times, a counter can be used in pseudocode, which is initialised before the DOWHILE statement and incremented just before the ENDDO statement. Let's look at an example.

EXAMPLE 5.1 Fahrenheit–Celsius conversion

Every day, a weather station receives 15 temperatures expressed in degrees Fahrenheit. A program is to be written that will accept each Fahrenheit temperature, convert it to Celsius and display the converted temperature to the screen. After 15 temperatures have been processed, the words 'All temperatures processed' are to be displayed on the screen.

A Defining diagram

Input	Processing	Output
f_temp (15 temperatures)	Get Fahrenheit temperatures Convert temperatures Display Celsius temperatures Display screen message	c_temp (15 temperatures)

Having defined the input, output and processing, you are ready to outline a solution to the problem. This can be done by writing down the control structures needed and any extra variables that are to be used in the solution algorithm. In this example, you need:

- a DOWHILE structure to repeat the necessary processing
- a counter, called temperature_count, initialised to zero, that will control the 15 repetitions.

B Solution algorithm

```
Fahrenheit_Celsius_conversion
1      Set temperature_count to zero
2      DOWHILE temperature_count < 15
3          Prompt operator for f_temp
4          Get f_temp
5          Compute c_temp = (f_temp - 32) * 5/9
6          Display c_temp
7          Add 1 to temperature_count
8      ENDDO
9      Display 'All temperatures processed' to the screen
END
```

This solution algorithm illustrates a number of points:

- 1 The temperature_count variable is initialised before the DOWHILE condition is executed.
- 2 As long as temperature_count is less than 15 (that is, the DOWHILE condition is true), the statements between DOWHILE and ENDDO will be executed.
- 3 The variable temperature_count is incremented once within the loop, just before the ENDDO delimiter (that is, just before it is tested again in the DOWHILE condition).
- 4 After 15 iterations, temperature_count will equal 15, which causes the DOWHILE condition to become false and control to be passed to the statement after ENDDO.

C Desk checking

Although the program will require 15 records to process properly, it is still only necessary to check the algorithm at this stage with two valid sets of data.

1 Input data

	First data set	Second data set
f_temp	32	50

2 Expected results

	First data set	Second data set
c_temp	0	10

3 Desk check table

Statement number	temperature_count	DOWHILE condition	f_temp	c_temp
1	0			
2		true		
3,4			32	
5				0
6				display
7	1			
2		true		
3,4			50	
5				10
6				display
7	2			

Using DOWHILE to repeat a set of instructions an unknown number of times

1 When a trailer record or sentinel exists

When there are an unknown number of items to process, you cannot use a counter, so another way of controlling the repetition must be used. Often, a trailer record or sentinel signifies the end of the data. This sentinel is a special record or value placed at the end of valid data to signify the end of that data. It must contain a value that is clearly distinguishable from the other data to be processed. It is referred to as a sentinel because it indicates that no more data follows. Let's look at an example.

EXAMPLE 5.2 Print examination scores

A program is required to read and print a series of names and exam scores for students enrolled in a mathematics course. The class average is to be computed and printed at the end of the report. Scores can range from 0 to 100. The last record contains a blank name and a score of 999 and is not to be included in the calculations.

A Defining diagram

Input	Processing	Output
name	Read student details	name
exam_score	Print student details Compute average score Print average score	exam_score average_score

You will need to consider the following when establishing a solution algorithm:

- a DOWHILE structure to control the reading of exam scores, until it reaches a score of 999
- an accumulator for total scores, namely total_score
- an accumulator for the total students, namely total_students.

B Solution algorithm

```
Print_examination_scores
1      Set total_score to zero
2      Set total_students to zero
3      Read name, exam_score
4      DOWHILE exam_score not = 999
5          Add 1 to total_students
6          Print name, exam_score
7          Add exam_score to total_score
8          Read name, exam_score
ENDDO
9      IF total_students not = zero THEN
        average_score = total_score/total_students
        Print average_score
ENDIF
END
```

In this example, the DOWHILE condition tests for the existence of the trailer record or sentinel (record 999). However, this condition cannot be tested until at least one exam mark has been read. Hence, the initial processing that sets up the condition is a Read statement immediately before the DOWHILE clause (Read name, exam_score). This is known as a priming read.

The algorithm will require another Read statement, this time within the body of the loop. Its position is also important. The trailer record or sentinel must not be included in the calculation of the average score, so each time an exam score is read, it must be tested for a 999 value, before further processing can take place. For this reason, the Read statement is placed at the end of the loop, immediately before ENDDO, so that its value can be tested when control returns to the DOWHILE condition. As soon as the trailer record has been read, control will exit from the loop to the next statement after ENDDO – that is, the calculation of average_score.

C Desk checking

Two valid records and a trailer record should be sufficient to desk check this algorithm.

1 Input data

	First record	Second record	Third record
score	50	100	999

2 Expected results

First name, and score of 50.

Second name, and score of 100.

Average score 75.

3 Desk check table

Statement number	total_score	total_students	exam_score	DOWHILE condition	average score
1,2	0	0			
3			50		
4				true	
5		1			
6			print		
7	50				
8			100		
4				true	
5		2			
6			print		
7	150				
8			999		
4				false	
9					75
					print

When a trailer record or sentinel does not exist

When there is no trailer record or sentinel to signify the end of the data, the programmer needs to check for an end-of-file marker (EOF). This EOF marker is added when the file is created, as the last character in the file. The check for EOF is positioned in the DOWHILE clause, using one of the following equivalent expressions:

DOWHILE more data
DOWHILE more records
DOWHILE records exist
DOWHILE NOT EOF

In this case, all statements between the words DOWHILE and ENDDO will be repeated until an attempt is made to read a record, but no more records exist. When this occurs, a signal is sent to the program to indicate that there are no more records and so the 'DOWHILE more records' condition is rendered false. Let's look at an example.

EXAMPLE 5.3 Process student enrolments

A program is required that will read a file of student records, and select and print only those students enrolled in a course unit named Programming I. Each student record contains student number, name, address, postcode, gender and course unit number. The course unit number for Programming I is 18500. Three totals are to be printed at the end of the report: total females enrolled in the course, total males enrolled in the course, and total students enrolled in the course.

A Defining diagram

Input	Processing	Output
student_record <ul style="list-style-type: none">• student_no• name• address• postcode• gender• course_unit	Read student records Select student records Print selected records Compute total females enrolled Compute total males enrolled Compute total students enrolled Print totals	selected student records total_females_enrolled total_males_enrolled total_students_enrolled

You will need to consider the following requirements when establishing a solution algorithm:

- a DOWHILE structure to perform the repetition
- IF statements to select the required students
- accumulators for the three total fields.

B Solution algorithm

```
Process_student_enrolments
1      Set total_females_enrolled to zero
2      Set total_males_enrolled to zero
3      Set total_students_enrolled to zero
4      Read student record
5      DOWHILE records exist
6          IF course_unit = 18500 THEN
              print student details
              increment total_students_enrolled
              IF student_gender = female THEN
                  increment total_females_enrolled
              ELSE
                  increment total_males_enrolled
              ENDIF
          ENDIF
          Read student record
      ENDDO
8      Print total_females_enrolled
9      Print total_males_enrolled
10     Print total_students_enrolled
END
```

C Desk checking

Three valid student records should be sufficient to desk check this algorithm. Since student_no, name, address and postcode are not operated upon in this algorithm, they do not need to be provided as input test data.

1 Input data

	First record	Second record	Third record
course_unit	20000	18500	18500
gender	F	F	M

2 Expected results

Student number, name, address, postcode, F (2nd student)

Student number, name, address, postcode, M (3rd student)

Total females enrolled 1

Total males enrolled 1

Total students enrolled 2

3 Desk check table

Statement number	course_unit	gender	DOWHILE condition	total_females_enrolled	total_males_enrolled	total_students_enrolled
1,2,3				0	0	0
4	20000	F				
5			true			
6						
7	18500	F				
5			true			
6	print	print		1		1
7	18500	M				
5			true			
6	print	print			1	2
7	EOF					
5			false			
8,9,10				print	print	print

The priming Read before the DOWHILE condition, together with the subsequent Read within the loop, immediately before the ENDDO statement, form the basic framework for DOWHILE repetitions in pseudocode. In general, all algorithms using a DOWHILE construct to process a sequential file should have the same basic pattern, as follows:

```

Process_sequential_file
  Initial processing
  Read first record
  DOWHILE more records exist
    Process this record
    Read next record
  ENDDO
  Final processing
END

```

5.2 Repetition using the REPEAT...UNTIL structure

Trailing decision loop

The REPEAT...UNTIL structure is similar to the DOWHILE structure, in that a group of statements are repeated in accordance with a specified condition. However, where the DOWHILE structure tests the condition at the *beginning* of the loop, a REPEAT...UNTIL structure tests the condition at the *end* of the loop. This means that the statements within the loop will be executed once before the condition is tested. If the condition is false, the statements will then be repeated UNTIL the condition becomes true.

The format of the REPEAT...UNTIL structure is:

```
REPEAT  
    statement  
    statement
```

UNTIL condition is true

You can see that REPEAT...UNTIL is a trailing decision loop; the statements are executed once before the condition is tested. There are two other considerations about which you need to be aware before using REPEAT...UNTIL.

First, REPEAT...UNTIL loops are executed when the condition is false; it is only when the condition becomes true that repetition ceases. Thus, the logic of the condition clause of the REPEAT...UNTIL structure is the opposite of DOWHILE. For instance, 'DOWHILE more records' is equivalent to 'REPEAT...UNTIL no more records', and 'DOWHILE number NOT = 99' is equivalent to 'REPEAT...UNTIL number = 99'.

Second, the statements within a REPEAT...UNTIL structure will always be executed at least once. As a result, there is no need for a priming Read when using REPEAT...UNTIL. One Read statement at the beginning of the loop is sufficient; however, an extra IF statement immediately after the Read statement must be included, to prevent the processing of the trailer record.

Let us now compare an algorithm that uses a DOWHILE structure with the same problem using a REPEAT...UNTIL structure. Consider the following DOWHILE loop:

```
Process_student_records
    Set student_count to zero
    Read student record
    DOWHILE student_number NOT = 999
        Write student record
        Increment student_count
        Read student record
    ENDDO
    Print student_count
END
```

This can be rewritten as a trailing decision loop, using the REPEAT...UNTIL structure as follows:

```
Process_student_records
    Set student_count to zero
    REPEAT
        Read student record
        IF student number NOT = 999 THEN
            Write student record
            Increment student_count
        ENDIF
    UNTIL student number = 999
    Print student_count
END
```

REPEAT...UNTIL loops are used less frequently in pseudocode than DOWHILE loops for sequential file processing because of this extra IF statement required within the loop. Let's look at an example.

EXAMPLE 5.4 Process inventory items

A program is required to read a series of inventory records that contain item number, item description and stock figure. The last record in the file has an item number of zero. The program is to produce a low stock items report, by printing only those records that have a stock figure of less than 20 items. A heading is to be printed at the top of the report and a total low stock item count printed at the end.

A Defining diagram

Input	Processing	Output
inventory record • item_number • item_description • stock_figure	Read inventory records Select low stock items Print low stock records Print total low stock items	heading selected records • item_number • item_description • stock_figure total_low_stock_items

You will need to consider the following requirements when establishing a solution algorithm:

- a REPEAT...UNTIL to perform the repetition
- an IF statement to select stock figures of less than 20
- an accumulator for total_low_stock_items
- an extra IF, within the REPEAT loop, to ensure the trailer record is not processed.

B Solution algorithm using REPEAT...UNTIL

```
Process_inventory_records
1      Set total_low_stock_items to zero
2      Print 'Low Stock Items' heading
REPEAT
3      Read inventory record
4      IF item_number > zero THEN
          IF stock_figure < 20 THEN
              print item_number, item_description, stock_figure
              increment total_low_stock_items
          ENDIF
      ENDIF
5      UNTIL item_number = zero
6      Print total_low_stock_items
END
```

The solution algorithm has a simple structure, with a single Read statement at the beginning of the REPEAT...UNTIL loop and an extra IF statement within the loop to ensure the trailer record is not incorrectly incremented into the total_low_stock_items accumulator.

C Desk checking

Two valid records and a trailer record (item number equal to zero) will be used to test the algorithm:

1 Input data

	First record	Second record	Third record
item_number	123	124	0
stock_figure	8	25	

2 Expected results

Low Stock Items

123 8 (first record)

Total Low Stock Items = 1

3 Desk check table

Statement number	item_number	stock_figure	REPEAT UNTIL	total_low_stock_items	heading
-				0	
2					print
3	123	8			
4	print	print		1	
5			false		
3	124	25			
4					
5			false		
3	0				
4					
5			true		
6					print

5.3 Counted repetition

Counted loop

Counted repetition occurs when the exact number of loop iterations is known in advance. The execution of the loop is controlled by a loop index, and instead of using DOWHILE, or REPEAT...UNTIL, the simple keyword DO is used as follows:

```
DO loop_index = initial_value to final_value  
    statement block  
ENDDO
```

The DO loop does more than just repeat the statement block. It will:

- 1 initialise the loop_index to the required initial_value
- 2 increment the loop_index by 1 for each pass through the loop
- 3 test the value of loop_index at the beginning of each loop to ensure that it is within the stated range of values
- 4 terminate the loop when the loop_index has exceeded the specified final_value.

In other words, a counted repetition construct will perform the initialising, incrementing and testing of the loop counter automatically. It will also terminate the loop once the required number of repetitions has been executed.

Let us look again at Example 5.1, which processes 15 temperatures at a weather station each day. The solution algorithm can be rewritten to use a DO loop.

EXAMPLE 5.5 Fahrenheit–Celsius conversion

Every day, a weather station receives 15 temperatures expressed in degrees Fahrenheit. A program is to be written that will accept each Fahrenheit temperature, convert it to Celsius and display the converted temperature to the screen. After 15 temperatures have been processed, the words 'All temperatures processed' are to be displayed on the screen.

A Defining diagram

Input	Processing	Output
f_temp (15 temperatures)	Get Fahrenheit temperatures Convert temperatures Display Celsius temperatures Display screen message	c_temp (15 temperatures)

B Solution algorithm

The solution will require a DO loop and a loop counter (temperature_count) to process the repetition.

```
Fahrenheit_Celsius_conversion
1   DO temperature_count = 1 to 15
2       Prompt operator for f_temp
3       Get f_temp
4       Compute c_temp = (f_temp - 32) * 5/9
5       Display c_temp
     ENDDO
6       Display 'All temperatures processed' to the screen
END
```

Note that the DO loop controls all the repetition:

- It initialises temperature_count to 1.
- It increments temperature_count by 1 for each pass through the loop.
- It tests temperature_count at the beginning of each pass to ensure that it is within the range 1 to 15.
- It automatically terminates the loop once temperature_count has exceeded 15.

C Desk checking

Two valid records should be sufficient to test the algorithm for correctness. It is not necessary to check the DO loop construct for all 15 records.

1 Input data

	First data set	Second data set
f_temp	32	50

2 Expected results

	First data set	Second data set
c_temp	0	10

3 Desk check table

Statement number	temperature_count	f_temp	c_temp
1	1		
2,3		32	
4			0
5			display
1	2		
2,3		50	
4			10
5			display

Desk checking the algorithm with the two input test cases indicates that the expected results have been achieved.

A requirement of counted repetition loops is that the exact number of input data items or records needs to be known before the algorithm can be written. Counted repetition loops are used extensively with arrays or tables, as seen in Chapter 7.

Chapter summary

This chapter covered the repetition control structure in detail. Descriptions and pseudocode examples were given for leading decision loops (DOWHILE), trailing decision loops (REPEAT...UNTIL) and counted loops (DO). Several solution algorithms that used each of the three control structures were defined, developed and desk checked.

We saw that most of the solution algorithms that used the DOWHILE structure had the same general pattern. This pattern consisted of:

- 1 some initial processing before the loop
- 2 some processing for each record within the loop
- 3 some final processing once the loop has been exited.

Expressed as a solution algorithm, this basic pattern was developed as a general solution:

```
Process_sequential_file
    Initial processing
    Read first record
    DOWHILE more records exist
        Process this record
        Read next record
    ENDDO
    Final processing
END
```

Programming problems

Construct a solution algorithm for the following programming problems. Your solution should contain:

- a defining diagram
- a pseudocode algorithm
- a desk check of the algorithm.

- 1 Design an algorithm that will output the seven times table, as follows:

7 x 1 = 7
7 x 2 = 14
7 x 3 = 21 . . .

- 2 Design an algorithm that will display to the screen the first 20 numbers, with their squares and cubes, as follows:

1	1	1
2	4	8
3	9	27 . . .

- 3 Design an algorithm that will prompt for, receive and total a collection of payroll amounts entered at the terminal until a sentinel amount of 999 is entered. After the sentinel has been entered, display the total payroll amount to the screen.

- 4 Design an algorithm that will read a series of integers at the terminal. The first integer is special, as it indicates how many more integers will follow. Your algorithm is to calculate and print the sum and average of the integers, excluding the first integer, and display these values to the screen.

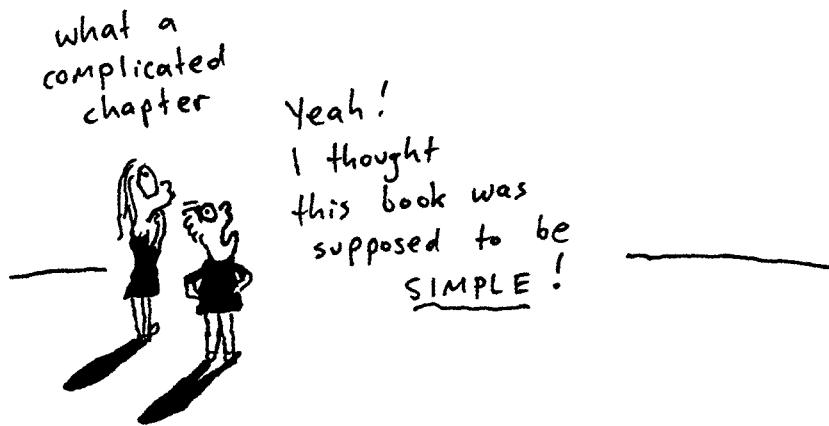
- 5 Design an algorithm that will prompt for and receive the time expressed in 2400 format (e.g. 2305 hours), convert it to 12-hour format (e.g. 11.05 p.m.) and display the new time to the screen. Your program is to repeat the processing until a sentinel time of 9999 is entered.

- 6** Design a program that will read a file of product records, each containing the item number, item name, the quantity sold this year and the quantity sold last year. The program is to produce a product list showing the item number, item name, and the increase or decrease in the quantity sold this year for each item.
- 7** The first record of a set of records contains a bank account number and an opening balance. Each of the remaining records in the set contains the amount of a cheque drawn on that bank account. The trailer record contains a zero amount. Design a program that will read and print the bank account number and opening balance on a statement of account report. The following cheque amounts are to be read and printed on the report, each with a new running balance. Print a closing balance at the end of the report.
- 8** Design a program that will read a file of employee records containing employee number, employee name, hourly pay rate, regular hours worked and overtime hours worked. The company pays its employees weekly, according to the following rules:
- regular pay = regular hours worked \times hourly rate of pay
overtime pay = overtime hours worked \times hourly rate of pay \times 1.5
total pay = regular pay + overtime pay
- Your program is to read the input data on each employee's record and compute and print the employee's total pay on the weekly payroll report. All input data and calculated amounts are to appear on the report. A total payroll amount is to appear at the end of the report.
- 9** Design an algorithm that will process the weekly employee time cards for all the employees of an organisation. Each employee time card will have three data items: an employee number, an hourly wage rate and the number of hours worked during a given week. Each employee is to be paid time-and-a-half for all hours worked over 35. A tax amount of 15% of gross salary is to be deducted. The output to the screen should display the employee's number and net pay. At the end of the run, display the total payroll amount and the average amount paid.
- 10** As a form of quality control, the Pancake Kitchen has recorded, on a Pancake file, two measurements for each of its pancakes made in a certain month: the thickness in mm (millimetres) and the diameter in cm (centimetres). Each record on the file contains two measurements for a pancake, thickness followed by diameter. The last record in the file contains values of 99 for each measurement. Design a program that will read the Pancake file, calculate the minimum, maximum and average for both the dimensions, and print these values on a report.

Chapter

6

Pseudocode algorithms using sequence, selection and repetition



Objectives

- To develop solution algorithms to eight typical programming problems using sequence, selection and repetition constructs

Outline

- 6.1 Eight solution algorithms
- Chapter summary
- Programming problems

6.1 Eight solution algorithms

This chapter develops solution algorithms to eight programming problems of increasing complexity. All the algorithms will use a combination of sequence, selection and repetition constructs. The algorithms have been designed to be interactive or to process sequential files. Reading these algorithms should consolidate the groundwork developed in the previous chapters.

Each programming problem will be defined, the control structures required will be determined and a solution algorithm will be devised.

1 Defining the problem

It is important that you divide the problem into its three components: input, output and processing. The processing component should list the tasks to be performed – that is, *what* needs to be done, not *how*. The verbs in each problem have been underlined to help identify the actions to be performed.

2 The control structures required

Once the problem has been defined, write down the control structures (sequence, selection and repetition) that may be needed, as well as any extra variables that the solution may require.

3 The solution algorithm

Having defined the problem and determined the required control structures, devise a solution algorithm and represent it using pseudocode. Each solution algorithm presented in this chapter is only one solution to the particular problem: other solutions could be equally correct.

4 Desk checking

You will need to desk check each of the algorithms with two or more test cases.

EXAMPLE 6.1 Process number pairs

Design an algorithm that will prompt for and receive pairs of numbers from an operator at a terminal and display their sum, product and average on the screen. If the calculated sum is over 200, an asterisk is to be displayed beside the sum. The program is to terminate when a pair of zero values is entered.

A Defining diagram

Input	Processing	Output
number1	Prompt for numbers	sum
number2	Get numbers Calculate sum Calculate product Calculate average Display sum, product, average	product average **
	Display '*'	

B Control structures required

- 1 A DOWHILE loop to control the repetition.
- 2 An IF statement to determine if an asterisk is to be displayed.
- 3 Note the use of the NOT operand with the AND logical operator.

C Solution algorithm

```
Process_number_pairs
    Set sum to zero
    Prompt for number1, number2
    Get number1, number2
    DOWHILE NOT (number1 = 0 AND number2 = 0)
        sum = number1 + number2
        product = number1 * number2
        average = sum / 2
        IF sum > 200 THEN
            Display sum, '*', product, average
        ELSE
            Display sum, product, average
        ENDIF
        Prompt for number1, number2
        Get number1, number2
    ENDDO
END
```

EXAMPLE 6.2 Print student records

A file of student records consists of 'S' records and 'U' records. An 'S' record contains the student's number, name, age, gender, address and attendance pattern; full-time (F/T) or part-time (P/T). A 'U' record contains the number and name of the unit or units in which the student has enrolled. There may be more than one 'U' record for each 'S' record. Design a solution algorithm that will read the file of student records and print only the student's number, name and address on a 'STUDENT LIST'.

A Defining diagram

Input	Processing	Output
's' records • number • name • address • age • gender • attendance_pattern 'u' records	Print heading Read student records Select 's' records Print selected records	Heading line selected student records • number • name • address

B Control structures required

- 1 A DOWHILE loop to control the repetition.
- 2 An IF statement to select 'S' records.

C Solution algorithm

```
Print_student_records
    Print 'STUDENT LIST' heading
    Read student record
    DOWHILE more records exist
        IF student record = 'S' record THEN
            Print student_number, name, address
        ENDIF
        Read student record
    ENDDO
END
```

EXAMPLE 6.3 Print selected students

Design a solution algorithm that will read the same student file as in Example 6.2, and produce a report of all female students who are enrolled part-time. The report is to be headed 'PART TIME FEMALE STUDENTS' and is to show the student's number, name, address and age.

A Defining diagram

Input	Processing	Output
's' records • number • name • address • age • gender • attendance_pattern 'u' records	Print heading Read student records Select P/T female students Print selected records	Heading line selected student records • number • name • address • age

B Control structures required

- 1 A DOWHILE loop to control the repetition.
- 2 An IF statement or statements to select 'S', female and part-time (P/T) students.

C Solution algorithm

Several algorithms for this problem will be presented, and all are equally correct. The algorithms only differ in the way the IF statement is expressed. It is interesting to compare the three different solutions.

Solution 1 uses a non-linear nested IF:

```
Produce_part_time_female_list
Print 'PART TIME FEMALE STUDENTS' heading
Read student record
DOWHILE more records
    IF student record = 'S' record THEN
        IF attendance_pattern = P/T THEN
            IF gender = female THEN
                Print student_number, name, address, age
            ENDIF
        ENDIF
    ENDIF
    Read student record
ENDDO
END
```

Solution 2 uses a nested and compound IF statement:

```
Produce_part_time_female_list
    Print 'PART TIME FEMALE STUDENTS' heading
    Read student record
    DOWHILE more records
        IF student record = 'S' record THEN
            IF (attendance_pattern = P/T
                AND gender = female) THEN
                    Print student_number, name, address, age
                ENDIF
            ENDIF
            Read student record
        ENDDO
    END
```

Solution 3 also uses a compound IF statement:

```
Produce_part_time_female_list
    Print 'PART TIME FEMALE STUDENTS' heading
    Read student record
    DOWHILE more records
        IF student record = 'S' record
            AND attendance_pattern = P/T
            AND gender = female THEN
                Print student_number, name, address, age
            ENDIF
            Read student record
        ENDDO
    END
```

EXAMPLE 6.4 Print and total selected students

Design a solution algorithm that will read the same student file as in Example 6.3 and produce the same 'PART TIME FEMALE STUDENTS' report. In addition, you are to print at the end of the report the number of students who have been selected and listed, and the total number of students on the file.

A Defining diagram

Input	Processing	Output
's' records • number • name • address • age • gender • attendance_pattern 'u' records	Print heading Read student records Select P/T female students Print selected records Compute total students Compute total selected students Print totals	Heading line selected student records • number • name • address • age total_students total_selected_students

B Control structures required

- 1 A DOWHILE loop to control the repetition.
- 2 IF statements to select 'S', female and P/T students.
- 3 Accumulators for total_selected_students and total_students.

C Solution algorithm

```
Produce_part_time_female_list
    Print 'PART TIME FEMALE STUDENTS' heading
    Set total_students to zero
    Set total_selected_students to zero
    Read student record
    DOWHILE records exist
        IF student record = 'S' record THEN
            increment total_students
            IF (attendance_pattern = P/T
                AND gender = female) THEN
                increment total_selected_students
                Print student_number, name, address, age
            ENDIF
        ENDIF
        Read student record
    ENDDO
    Print total_students
    Print total_selected_students
END
```

Note the positions where the total accumulators are incremented. If these statements are not placed accurately within their respective IF statements, the algorithm could produce erroneous results.

EXAMPLE 6.5 Print student report

Design an algorithm that will read the same student file as in Example 6.4 and, for each student, print the name, number and attendance pattern from the 'S' records (student records) and the unit number and unit name from the 'U' records (enrolled units records) as follows.

STUDENT REPORT

Student name
Student number
Attendance
Enrolled units

At the end of the report, print the total number of students enrolled.

A Defining diagram

Input	Processing	Output
's' records • number • name • attendance_pattern	Print heading Read student records Print 's' record details Print 'u' record details Compute total students Print total students	Heading line detail lines • name • number • attendance_pattern • unit_number • unit_name total_students
'u' records • unit_number • unit_name		

B Control structures required

- 1 A DOWHILE loop to control the repetition.
- 2 An IF statement to select 'S' or 'U' records.
- 3 An accumulator for total_students.

C Solution algorithm

```
Print_student_report
    Print 'STUDENT REPORT' heading
    Set total_students to zero
    Read student record
    DOWHILE records exist
        IF student record = 'S' THEN
            add 1 to total_students
            Print 'Student name', name
            Print 'Student number', number
            Print 'Attendance', attendance_pattern
            Print 'Enrolled units'
        ELSE
            IF student record = 'U' THEN
                Print unit_number, unit_name
            ELSE
                Print 'student record error'
            ENDIF
        ENDIF
        Read student record
    ENDDO
    Print 'Total students', total_students
END
```

EXAMPLE 6.6 Produce sales report

Design a program that will read a file of sales records and produce a sales report. Each record in the file contains a customer's number, name, a sales amount and a tax code. The tax code is to be applied to the sales amount to determine the sales tax due for that sale, as follows:

Tax code	Sales tax
0	tax exempt
1	3%
2	5%

The report is to print a heading 'SALES REPORT', and detail lines listing the customer number, name, sales amount, sales tax and the total amount owing.

A Defining diagram

Input	Processing	Output
sales record • customer_number • name • sales_amt • tax_code	Print heading Read sales records Calculate sales tax Calculate total amount Print customer details	Heading line detail lines • customer_number • name • sales_amt • sales_tax • total_amount

B Control structures required

- 1 A DOWHILE loop to control the repetition.
- 2 A case statement to calculate the sales_tax.

Assume that the tax_code field has been validated and will contain only a value of 0, 1 or 2.

C Solution algorithm

```
Produce_sales_report
    Print 'SALES REPORT' heading
    Read sales record
    DOWHILE not EOF
        CASE of tax_code
            0 : sales_tax = 0
            1 : sales_tax = sales_amt * 0.03
            2 : sales_tax = sales_amt * 0.05
        ENDCASE
        total_amt = sales_amt + sales_tax
        Print customer_number, name, sales_amt, sales_tax, total_amt
        Read sales record
    ENDDO
END
```

EXAMPLE 6.7 Student test results

Design a solution algorithm that will read a file of student test results and produce a student test grades report. Each test record contains the student number, name and test score (out of 50). The program is to calculate for each student the test score as a percentage and to print the student's number, name, test score (out of 50) and letter grade on the report. The letter grade is determined as follows:

A = 90-100%	D = 60-69%
B = 80-89%	F = 0-59%
C = 70-79%	

A Defining diagram

Input	Processing	Output
Student test records • student_number • name • test_score	Print heading Read student records Calculate test percentage Calculate letter grade Print student details	Heading line student details • student_number • name • test_score • grade

B Control structures required

- 1 A DOWHILE loop to control the repetition.
- 2 A formula to calculate the percentage.
- 3 A linear nested IF statement to calculate the grade.
(The case construct cannot be used here, as CASE is not designed to cater for a range of values – for example, 0-59%).)

C Solution algorithm

```
Print_student_results
    Print 'STUDENT TEST GRADES' heading
    Read student record
    DOWHILE not EOF
        percentage = test_score * 2
        IF percentage >= 90 THEN
            grade = 'A'
        ELSE
            IF percentage >= 80 THEN
                grade = 'B'
            ELSE
                IF percentage >= 70 THEN
                    grade = 'C'
                ELSE
                    IF percentage >= 60 THEN
                        grade = 'D'
                    ELSE
                        grade = 'F'
                    ENDIF
                ENDIF
            ENDIF
        ENDIF
        Print student_number, name, test_score, grade
        Read student record
    ENDDO
END
```

Note that the linear nested IF has been worded so that all alternatives have been considered.

EXAMPLE 6.8 Gas supply billing

The Domestic Gas Supply Company records its customers' gas usage figures on a customer usage file. Each record on the file contains the customer number, customer name, customer address, and gas usage expressed in cubic metres. Design a solution algorithm that will read the customer usage file, calculate the amount owing for gas usage for each customer, and print a report listing each customer's number, name, address, gas usage and the amount owing.

The company bills its customers according to the following rate: if the customer's usage is 60 cubic metres or less, a rate of \$2.00 per cubic metre is applied; if the customer's usage is more than 60 cubic metres, then a rate of \$1.75 per cubic metre is applied for the first 60 cubic metres and \$1.50 per cubic metre for the remaining usage.

At the end of the report, print the total number of customers and the total amount owing to the company.

A Defining diagram

Input	Processing	Output
customer usage records <ul style="list-style-type: none">• customer_number• name• address• gas_usage	Print heading Read usage records Calculate amount owing Print customer details Compute total customers Compute total amount owing Print totals	Heading line customer details <ul style="list-style-type: none">• customer_number• name• address• gas_usage• amount_owing total_customers total_amount_owing

B Control structures required

- 1 A DOWHILE loop to control the repetition.
- 2 An IF statement to calculate the amount_owing.
- 3 Accumulators for total_customers and total_amount_owing.

C Solution algorithm

```
Bill_gas_customers
    Print 'CUSTOMER USAGE FIGURES' heading
    Set total_customers to zero
    Set total_amount_owing to zero
    Read customer record
    DOWHILE more records
        IF usage <= 60 THEN
            amount_owing = usage * $2.00
        ELSE
            amount_owing = (60 * $1.75) + ((usage - 60) * $1.50)
        ENDIF
        Print customer_number, name, address, gas_usage, amount_owing
        Add amount_owing to total_amount_owing
        Add 1 to total_customers
        Read customer record
    ENDDO
    Print total_customers
    Print total_amount_owing
END
```

Chapter summary

This chapter developed solution algorithms to eight typical programming problems. The approach to all eight problems followed the same pattern:

- 1 The problem was defined, using a defining diagram.
- 2 The control structures required were written down, along with any extra variables required.
- 3 The solution algorithm was produced, using pseudocode and the three basic control structures: sequence, selection and repetition.

It was noted that the solution algorithms mostly followed the same basic pattern, although the statements within the pattern were different. This pattern was first introduced in Chapter 5, as follows:

```
Process_sequential_file
    Initial processing
    Read first record
    DOWHILE more records exist
        Process this record
        Read next record
    ENDDO
    Final processing
END
```

Programming problems

Construct a solution algorithm for the following programming problems. Your solution should contain:

- a defining diagram
- a list of control structures required
- a pseudocode algorithm
- a desk check of the algorithm.

- 1 Design an algorithm that will prompt for and receive your age in years and months and calculate and display your age in months. If the calculated months figure is more than 500, three asterisks should also appear beside the month figure. Your program is to continue processing until a sentinel of 999 is entered.
- 2 Design an algorithm that will prompt for and receive the measurement for the diameter of a circle, and calculate and display the area and circumference of that circle. Your program is to continue processing until a sentinel of 9999 is entered.
- 3 A file of student records contains name, gender (M or F), age (in years) and marital status (single or married) for each student. Design an algorithm that will read through the file and calculate the numbers of married men, single men, married women and single women. Print these numbers on a student summary report. If any single men are over 30 years of age, print their names and ages on a separate eligible bachelors report.
- 4 Design an algorithm that will read a file of employee records and produce a weekly report of gross earnings for those employees. Gross earnings are earnings before tax and other deductions have been deducted. Each input record contains the employee number, the hours worked and the hourly rate of pay. Each employee's gross pay is calculated as the product of the hours worked and the rate of pay. At the end of the report, print the total gross earnings for that week.
- 5 Design an algorithm that will read the same file as in Problem 4, and produce a weekly report of the net earnings for those employees. Net earnings are gross earnings minus deductions. Each employee has two deductions from their gross earnings each week: tax payable (15% of gross earnings) and medical levy (1% of gross earnings). Your report is to print the gross earnings, tax payable, medical levy and net earnings for each employee. At the end of the report, print the total gross earnings, total tax, total medical levy and total net earnings.
- 6 A parts inventory record contains the following fields:
 - record code (only code 11 is valid)
 - part number (six characters; two alpha and four numeric – for example, AA1234)
 - part description
 - inventory balance.

Design an algorithm that will read the file of parts inventory records, validate the record code and part number on each record, and print the details of all valid inventory records that have an inventory balance equal to zero.

7 Design a program that will read the same parts inventory file described in Problem 6, validate the record code and part number on each record, and print the details of all valid records whose part numbers fall within the values AA3000 and AA3999 inclusive. Also print a count of these selected records at the end of the parts listing.

8 Design a program that will produce the same report as in Problem 7, but, in addition, print at the end of the parts listing a count of all the records whose part number falls between AA3000 and AA3999, as well as a count of all records whose part numbers begin with the value 'AA'.

9 An electricity supply authority records the amount of electricity that each customer uses on an electricity usage file. This file consists of:

- a** a header record (first record), which provides the total kilowatt-hours used during the month by all customers
- b** a number of detail records, each containing the customer number, customer name and electricity usage (in kilowatt hours) for the month.

Design a solution algorithm that will read the electricity usage file and produce an electricity usage report showing the customer number, customer name, electricity usage and the amount owing. The amount owing is calculated at 11 cents for each kilowatt-hour used, up to 200 hours, and 8 cents for each kilowatt-hour used over 200 hours. The total electricity usage in kilowatt-hours is also to be accumulated.

At the end of the program, compare the total electricity usage accumulated in the program with the value provided in the header record, and print an appropriate message if the totals are not equal.

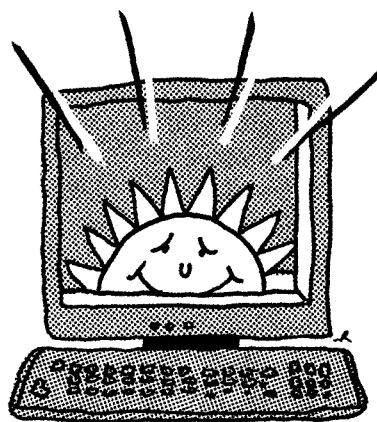
10 Design an algorithm that will read a file of customer records, showing the total amount owing on his or her credit card, and produce a report showing the customer's minimum amount due. Each customer record contains the customer number, name, address, postcode and total amount owing. The minimum amount due is calculated on the total amount owing, as follows:

If the total amount owing is less than \$5.00, the total amount owing becomes the minimum amount due. If the total amount owing is greater than \$5.00, the minimum amount due is calculated to be one-quarter of the total amount owing, provided this resulting amount is not less than \$5.00. If the resulting amount is less than \$5.00, the minimum amount due is \$5.00.

Chapter

Array processing

7



Array of sunshine

Objectives

- To introduce arrays and the uses of arrays
- To develop pseudocode algorithms for common operations on arrays
- To illustrate the manipulation of single- and two-dimensional arrays

Outline

- 7.1 Array processing
 - 7.2 Initialising the elements of an array
 - 7.3 Searching an array
 - 7.4 Writing out the contents of an array
 - 7.5 Programming examples using arrays
 - 7.6 Two-dimensional arrays
- Chapter summary
Programming problems

7.1 Array processing

Arrays are one of the most powerful programming tools available. They provide the programmer with a way of organising a collection of homogeneous data items (that is, items that have the same type and the same length) into a single data structure. An array, then, is a data structure that is made up of a number of variables all of which have the same data type – for example, all the exam scores for a class of 30 mathematics students. By using an array, a single variable name such as ‘scores’ can be associated with all 30 exam scores.

The individual data items that make up the array are referred to as the elements of the array. Elements in the array are distinguished from one another by the use of an index or subscript, enclosed in parentheses, following the array name – for example: ‘scores (3)’.

The subscript indicates the position of an element within the array; so, scores (3) refers to the third exam score, or the third element of the array scores, and scores (23) refers to the 23rd exam score.

The subscript or index may be a number or a variable, and may then be used to access any item within the valid bounds of an array – for example:

scores(6), or
element(index)

Arrays are an internal data structure – that is, they are required only for the duration of the program in which they are defined. They are a very convenient mechanism for storing and manipulating a collection of similar data items in a program, and a programmer should be familiar with the operations most commonly performed on them. Arrays are sometimes referred to as tables.

Operations on arrays

Arrays can be used:

- to load initial information into an array that is later required for processing
- to process the elements of the array
- to store the results of processing into the elements of an array
or
- to store information, which is then written to a report.

Thus, the most typical operations performed on arrays are:

- loading a set of initial values into the elements of an array
- processing the elements of an array
- searching an array, using a linear or binary search, for a particular element
- writing out the contents of an array to a report.

Usually, the elements of an array are processed in sequence, starting with the first element. This can be accomplished easily in pseudocode with either a DO loop or a DOWHILE loop.

Simple algorithms that manipulate arrays

The following algorithms involve the simple manipulation of arrays. Each algorithm is written using a DO loop. In each algorithm, the contents of the array and the number of elements have been established and the subscript is named 'index'. The number of elements in the array is stored in the variable `number_of_elements`.

EXAMPLE 7.1 Find the sum of the elements of an array

In this example, each element of the array is accumulated into a variable called `sum`. When all elements have been added, the variable `sum` is printed.

```
Find_sum_of_elements
    Set sum to zero
    DO index = 1 to number_of_elements
        sum = sum + array (index)
    ENDDO
    Print sum
END
```

EXAMPLE 7.2 Find the mean (average) of the elements of an array

In this example, each element of the array is accumulated into a variable called `sum`. When all elements have been added, the average of the elements is found and printed.

```
Find_element_average
    Set sum to zero
    DO index = 1 to number_of_elements
        sum = sum + array (index)
    ENDDO
    average = sum / number_of_elements
    Print average
END
```

EXAMPLE 7.3 Find the largest of the elements of an array

In this example, the elements of an array are searched to determine which element is the largest. The algorithm starts by putting the first element of the array into the variable largest_element, and then looks at the other elements of the array to see if a larger value exists. The largest value is then printed.

```
Find_largest_element
    Set largest_element to array (1)
    DO index = 2 to number_of_elements
        IF array (index) > largest_element THEN
            largest_element = array (index)
        ENDIF
    ENDDO
    Print largest_element
END
```

EXAMPLE 7.4 Find the smallest of the elements of an array

In this example, the elements of an array are searched to determine the smallest element. The algorithm starts by putting the first element of the array into the variable smallest_element, and then looks at the other elements of the array to see if a smaller value exists. The smallest value is then printed.

```
Find_smallest_element
    Set smallest_element to array (1)
    DO index = 2 to number_of_elements
        IF array (index) < smallest_element THEN
            smallest_element = array (index)
        ENDIF
    ENDDO
    Print smallest_element
END
```

EXAMPLE 7.5 Find the range of the elements of an array

In this example, the elements of an array are searched to determine the smallest and the largest elements. The algorithm starts by putting the first element of the array into the variables smallest_element and largest_element, and then looks at the other elements to see if a smaller or larger value exists. The two values are then printed.

```

Find_range_of_elements
    Set smallest_element to array (1)
    Set largest_element to array (1)
    DO index = 2 to number_of_elements
        IF array (index) < smallest_element THEN
            smallest_element = array (index)
        ELSE
            IF array (index) > largest_element THEN
                largest_element = array (index)
            ENDIF
        ENDIF
    ENDDO
    Print the range as smallest_element followed by largest_element
END

```

7.2 Initialising the elements of an array

Because an array is an internal data structure, initial values must be placed into the array before any information can be retrieved from it. These initial values can be assigned to the elements of the array as constants, or they can be read into the array from a file.

Loading constant values into an array

This method should only be used when the data in the array is unlikely to be changed – for example, the names of the 12 months of the year. To initialise such an array, establish an array called month_table, which contains 12 elements all of the same size. Then assign the elements of the array with the names of the months, one by one, as follows:

```

Initialise_month_table
    month_table(1) = 'January '
    month_table(2) = 'February '
    :
    :
    :
    month_table(12) = 'December '
END

```

Note that each element of the array must be the size of the largest month – that is, September – so, the shorter month names must be padded with blanks.

Loading initial values into an array from an input file

Defining array elements as constants in a program is not recommended if the values change frequently, as the program will need to be changed every time an array element changes. A common procedure is to read the input values into the elements of an array from an input file.

The reading of a series of values from a file into an array can be represented by a simple DOWHILE loop. The loop should terminate when either the array is full or the input file has reached end of file. Both of these conditions can be catered for in the condition clause of the DOWHILE loop.

In the following pseudocode algorithm, values are read from an input file and assigned to the elements of an array, starting with the first element, until there are no more input values or the array is full. The array name is 'array', the subscript is 'index', and the maximum number of elements that the array can hold is max_num_elements.

```
Read_values_into_array
    Set max_num_elements to required value
    Set index to zero
    Read first input value
    DOWHILE (input values exist) AND (index < max_num_elements)
        index = index + 1
        array (index) = input value
        Read next input value
    ENDDO
    IF (input values exist) AND index = max_num_elements THEN
        Print 'Array size too small'
    ENDIF
END
```

Note that the processing will terminate when either the input file has reached EOF or the array is full. An error message will be printed if there are more input data items than there are elements in the array.

Arrays of variable size

In some programs, the number of entries in an array can vary. In this case, a sentinel value (for example, 9999) is used to mark the last element of the array, both in the initialising file of data items and in the array itself. The sentinel record will indicate the end of input records during initial processing and the last element of the array during further processing. The algorithm for loading values into an array of variable size must include a check to ensure that no attempt is made to load more entries into the array than there are elements, as in the following example:

```

Read_values_into_variable_array
Set max_num_elements to required value
Set index to zero
Read first input value
DOWHILE (input value NOT = 9999) AND (index < max_num_elements)
    index = index + 1
    array (index) = input value
    Read next input value
ENDDO
IF index < max_num_elements THEN
    index = index + 1
    array (index) = 9999
ELSE
    Print 'Array size too small'
ENDIF
END

```

Note that the processing will terminate when either the sentinel record has been read or the array is full. An error message will be printed if there are more input data items than there are elements in the array.

Paired arrays

Many arrays in business applications are paired – that is, there are two arrays that have the same number of elements. The arrays are paired because the elements in the first array correspond to the elements in the same position in the second array. For example, a sales number array can be paired with a sales name array. Both arrays would have the same number of elements, with corresponding sales numbers and sales names. When you have determined where in the sales number array a particular salesperson's number appears, retrieve the salesperson's name from the corresponding position in the sales name array. In this way, the salesperson's number and name can appear on the same report, without any extra keying.

In the following example, an algorithm has been designed to read a file of product codes and corresponding selling prices for a particular company and to load them into two corresponding arrays, named product_codes and selling_prices. In the algorithm, the subscript is 'index', and the field max_num_elements contains the total number of elements in each array.

```

Read_values_into_paired_arrays
    Set max_num_elements to required value
    Set index to zero
    Read first input record
    DOWHILE (NOT EOF input record) AND (index < max_num_elements)
        index = index + 1
        product_codes (index) = input product_code
        selling_prices (index) = input selling_price
        Read next record
    ENDDO
    IF (NOT EOF input record) AND index = max_num_elements THEN
        Print 'Array size too small'
    ENDIF
END

```

7.3 Searching an array

A common operation on arrays is to search the elements of an array for a particular data item. The reasons for searching an array may be:

- to edit an input value – that is, to check that it is a valid element of an array
- to retrieve information from an array
- to retrieve information from a corresponding element in a paired array.

When searching an array, it is an advantage to have the array sorted into ascending sequence, so that, when a match is found, the rest of the array does not have to be searched. If you find an array element that is equal to an input entry, a match has been found and the search can be stopped. Also, if you find an array element that is greater than an input entry, no match has been found and the search can be stopped. Note that if the larger entries of an array are searched more often than the smaller entries, it may be an advantage to sort the array into descending sequence.

An array can be searched using either a linear search or a binary search.

A linear search of an array

A linear search involves looking at each of the elements of the array, one by one, starting with the first element. Continue the search until either you find the element being looked for or you reach the end of the array. A linear search is often used to validate data items.

The pseudocode algorithm for a linear search of an array will require a program flag named `element_found`. This flag, initially set to false, will be set to true once the value being looked for is found – that is, when the current element of the array is equal to the data item being looked for. In the following algorithm, the data item being searched for is stored in the variable `input_value`, and `max_num_elements` contains the total number of elements in the array.

```
Linear_search_of_an_array
Set max_num_elements to required value
Set element_found to false
Set index to 1
DOWHILE (NOT element_found) AND (index < = max_num_elements)
    IF array (index) = input_value THEN
        Set element_found to true
    ELSE
        index = index + 1
    ENDIF
ENDDO
IF element_found THEN
    Print array (index)
ELSE
    Print 'value not found', input_value
ENDIF
END
```

A binary search of an array

When the number of elements in an array exceeds 25, and the elements are sorted into ascending sequence, a more efficient method of searching the array is a binary search.

A binary search locates the middle element of the array first, and determines if the element being searched for is in the first or second half of the table. The search then points to the middle element of the relevant half table, and the comparison is repeated. This technique of continually halving the area under consideration is continued until the data item being searched for is found, or its absence is detected.

In the following algorithm, a program flag named element_found is used to indicate whether the data item being looked for has been found. The variable low_element indicates the bottom position of the section of the table being searched, and high_element indicates the top position. The maximum number of elements that the array can hold is placed in the variable max_num_elements.

The binary search will continue until the data item has been found, or there can be no more halving operations (that is, low_element is not less than high_element).

```

Binary_search_of_an_array
    Set element_found to false
    Set low_element to 1
    Set high_element to max_num_elements
    DOWHILE (NOT element_found) AND (low_element <= high_element)
        index = (low_element + high_element) / 2
        IF input_value = array (index) THEN
            Set element_found to true
        ELSE
            IF input_value < array (index) THEN
                high_element = index - 1
            ELSE
                low_element = index + 1
            ENDIF
        ENDIF
    ENDDO
    IF element_found THEN
        Print array (index)
    ELSE
        Print 'value not found', input_value
    ENDIF
END

```

7.4 Writing out the contents of an array

The elements of an array can be used as accumulators of data, to be written to a report. Writing out the contents of an array involves starting with the first element of the array and continuing until all elements have been written. This can be represented by a simple DO loop.

In the following pseudocode algorithm, the name of the array is 'array' and the subscript is 'index'. The number of elements in the array is represented by number_of_elements.

```

Write_values_of_array
    DO index = 1 to number_of_elements
        Print array (index)
    ENDDO
END

```

7.5 Programming examples using arrays

EXAMPLE 7.6 Process exam scores

Design a program that will prompt for and receive 18 examination scores from a mathematics test, compute the class average, and display all the scores and the class average to the screen.

A Defining diagram

Input	Processing	Output
18 exam scores	Prompt for scores Get scores Compute class average Display scores Display class average	18 exam scores class_average

B Control structures required

- 1 An array to store the exam scores – that is, ‘scores’.
- 2 An index to identify each element in the array.
- 3 A DO loop to accept the scores.
- 4 Another DO loop to display the scores to the screen.

C Solution algorithm

```
Process_exam_scores
    Set total_score to zero
    DO index = 1 to 18
        Prompt operator for score
        Get scores(index)
        total_score = total_score + scores(index)
    ENDDO
    Compute average_score = total_score / 18
    DO index = 1 to 18
        Display scores(index)
    ENDDO
    Display average_score
END
```

EXAMPLE 7.7 Process integer array

Design an algorithm that will read an array of 100 integer values, calculate the average integer value, and count the number of integers in the array that are greater than the average integer value. The algorithm is to display the average integer value and the count of integers greater than the average.

A Defining diagram

Input	Processing	Output
100 integer values	Read integer values Compute integer average Compute integer count Display integer average Display integer count	integer_average integer_count

B Control structures required

- 1 An array of integer values – that is, numbers.
- 2 A DO loop to calculate the average of the integers.
- 3 A DO loop to count the number of integers greater than the average.

C Solution algorithm

```
Process_integer_array
    Set integer_total to zero
    Set integer_count to zero
    DO index = 1 to 100
        integer_total = integer_total + numbers (index)
    ENDDO
    integer_average = integer_total / 100
    DO index = 1 to 100
        IF numbers (index) > integer_average THEN
            add 1 to integer_count
        ENDIF
    ENDDO
    Display integer_average, integer_count
END
```

EXAMPLE 7.8 Validate sales number

Design an algorithm that will read a file of sales transactions and validate the sales numbers on each record. As each sales record is read, the sales number on the record is to be verified against an array of 35 sales numbers. Any sales number not found in the array is to be flagged as an error.

A Defining diagram

Input	Processing	Output
sales records • sales_number	Read sales records Validate sales numbers Print error message	error_message

B Control structures required

- 1 A previously initialised array of sales numbers – that is, sales_numbers.
- 2 A DOWHILE loop to read the sales file.
- 3 A DOWHILE loop to perform a linear search of the array for the sales number.
- 4 A variable element_found that will stop the search when the sales number is found.

C Solution algorithm

```
Validate_sales_numbers
    Set max_num_elements to 35
    Read sales record
    DOWHILE sales records exist
        Set element_found to false
        Set index to 1
        DOWHILE (NOT element_found) AND (index <= max_num_elements)
            IF sales_numbers (index) = input sales number THEN
                Set element_found to true
            ELSE
                index = index + 1
            ENDIF
        ENDDO
        IF NOT element_found THEN
            Print 'invalid sales number', input sales number
        ENDIF
        Read sales record
    ENDDO
END
```

EXAMPLE 7.9 Calculate freight charge

Design an algorithm that will read an input weight for an item to be shipped, search an array of shipping weights and retrieve a corresponding freight charge. In this algorithm, two paired arrays, each containing six elements, have been established and initialised. The array, `shipping_weights`, contains a range of shipping weights in grams, and the array, `freight_charges`, contains a corresponding array of freight charges in dollars, as follows.

Shipping weights (grams)	Freight charges
1-100	3.00
101-500	5.00
501-1000	7.50
1001-3000	12.00
3001-5000	16.00
5001-9999	35.00

A Defining diagram

Input	Processing	Output
entry weight	Prompt for entry weight Get entry weight Search shipping weights array Compute freight charges Display freight charge	freight_charge error_message

B Control structures required

- 1 Two arrays, `shipping_weights` and `freight_charges`, already initialised.
- 2 A DOWHILE loop to search the `shipping_weights` array and hence retrieve the freight charge.
- 3 A variable `element_found` that will stop the search when the entry weight is found.

C Solution algorithm

```
Calculate_freight_charge
    Set max_num_elements to 6
    Set index to 1
    Set element_found to false
    Prompt for entry weight
    Get entry weight
    DOWHILE (NOT element_found) AND (index <= max_num_elements)
        IF shipping_weights (index) < entry weight THEN
            add 1 to index
        ELSE
            Set element_found to true
        ENDIF
    ENDDO
    IF element_found THEN
        freight_charge = freight_charges (index)
        Display 'Freight charge is', freight_charge
    ELSE
        Display 'invalid shipping weight', entry weight
    ENDIF
END
```

7.6 Two-dimensional arrays

So far, all algorithms in this chapter have manipulated one-dimensional arrays – that is, only one subscript is needed to locate an element in an array. In some business applications, there is a need for multidimensional arrays, where two or more subscripts are required to locate an element in an array. The following freight charges array is an example of a two-dimensional array, and is an extension of Example 7.9 above. It is a two-dimensional array because the calculation of the freight charges to ship an article depends on two values: the shipping weight of the article, and the geographical area or zone to which it is to be shipped, namely zones 1, 2, 3 or 4.

The range of shipping weights, in grams, is provided in the same one-dimensional array as in Example 7.9, as follows.

Shipping weights (grams)
1-100
101-500
501-1000
1001-3000
3001-5000
5001-9999

Freight charges (\$) (by shipping zone)			
1	2	3	4
2.50	3.50	4.00	5.00
3.50	4.00	5.00	6.50
4.50	6.50	7.50	10.00
10.00	11.00	12.00	13.50
13.50	16.00	20.00	27.50
32.00	34.00	35.00	38.00

In the freight charges array, any one of four freight charges may apply to a particular shipping weight, depending on the zone to which the shipment is to be delivered. Thus, the array is set out as having rows and columns, where the six rows represent the six shipping weight ranges, and the four columns represent the four geographical zones.

The number of elements in a two-dimensional array is calculated as the product of the number of rows and the number of columns – in this case, $6 \times 4 = 24$.

An element of a two-dimensional array is specified using the name of the array, followed by two subscripts, enclosed in parentheses, separated by a comma. The row subscript is specified first, followed by the column subscript. Thus, an element of the above freight charges array would be specified as `freight_charges (row_index, column_index)`. So, `freight_charges (5, 3)` refers to the freight charge listed in the array where the fifth row and the third column intersect – that is, a charge of \$20.00.

Loading a two-dimensional array

A two-dimensional array is loaded in columns within row order – that is, all the columns for row one are loaded before moving to row two and loading the columns for that row, and so on.

In the following pseudocode algorithm, values are read from an input file of freight charges and assigned to a two-dimensional `freight_charges` array. The array has six rows, representing the six shipping weight ranges, and four columns, representing the four geographical shipping zones, as in the above example.

The reading of a series of values from a file into a two-dimensional array can be represented by a DO loop within a DOWHILE loop.

```
Read_values_into_array
    Set max_num_elements to 24
    Set row_index to zero
    Read input file
    DOWHILE (input values exist) AND (row_index < 6)
        row_index = row_index + 1
        DO column_index = 1 to 4
            freight_charges (row_index, column_index) = input value
            Read input file
        ENDDO
    ENDDO
    IF (input values exist) AND row_index = 6 THEN
        Print 'Array size too small'
    ENDIF
END
```

Searching a two-dimensional array

Search method 1

In the following pseudocode algorithm, the freight charges for an article are to be calculated by searching a previously initialised two-dimensional array. The input values for the algorithm are the shipping weight of the article, and the geographical zone to which it is to be shipped.

First, the one-dimensional `shipping_weights` array is searched for the correct weight category (`row_index`) and then the two-dimensional `freight_charges` array is looked up using that weight category (`row_index`) and geographical zone (`column_index`).

```

Calculate_Freight_Charges
    Set row_index to 1
    Set element_found to false
    Prompt for shipping_weight, zone
    Get shipping_weight, zone
    DOWHILE (NOT element_found) AND (row_index <= 6)
        IF shipping_weights (row_index) < input shipping_weight THEN
            add 1 to row_index
        ELSE
            Set element_found to true
        ENDIF
    ENDDO
    IF element_found THEN
        IF zone = (1 or 2 or 3 or 4) THEN
            freight_charge = freight_charges (row_index, zone)
            Display 'Freight charge is', freight_charge
        ELSE
            Display 'invalid zone', zone
        ENDIF
    ELSE
        Display 'invalid shipping weight', input shipping_weight
    ENDIF
END

```

Search method 2

In the following algorithm, an input employee number is validated against a two-dimensional array of employee numbers, which has 10 rows and five columns. The array is searched sequentially, by columns within rows, using two DOWHILE loops until a match is found. If no match is found, an error message is printed.

```

Search_employee_numbers
    Set row_index to 1
    Set employee_found to false
    Read input employee_number
    DOWHILE (NOT employee_found) AND (row_index <=10)
        Set column_index to 1
        DOWHILE (NOT employee_found) AND (column_index <= 5)
            IF employee_numbers (row_index, column_index) = input
                employee_number THEN
                Set employee_found to true
            ENDIF
            column_index = column_index + 1
        ENDDO
        row_index = row_index + 1
    ENDDO
    IF NOT employee_found THEN
        Print 'invalid employee number', input employee_number
    ENDIF
END

```

Writing out the contents of a two-dimensional array

To write out the contents of a two-dimensional array, write out the elements in the columns within a row, before moving on to the next row. This is represented in pseudocode by a DO loop within another DO loop.

In the following pseudocode algorithm, the elements of a two-dimensional array are printed to a report, by column within row, using two subscripts.

```
Write_values_of_array
    Set number_of_rows to required value
    Set number_of_columns to required value
    DO row_index = 1 to number_of_rows
        DO column_index = 1 to number_of_columns
            Print array (row_index, column_index)
        ENDDO
    ENDDO
END
```

Chapter summary

This chapter defined an array as a data structure made up of a number of variables or data items that all have the same data type and are accessed by the same name. The individual elements that make up the array are accessed by the use of an index or subscript beside the name of the array – for example, scores (3).

Algorithms were developed for the most common operations on arrays, namely:

- loading a set of initial values into the elements of an array
- processing the elements of an array
- searching an array, using a linear or binary search, for a particular element
- writing out the contents of an array to a report.

Programming examples using both one- and two-dimensional arrays were developed.

Programming problems

Construct a solution algorithm for the following programming problems. Your solution should contain:

- a defining diagram
- a list of control structures required
- a pseudocode algorithm
- a desk check of the algorithm.

- 1 Design an algorithm that will read an array of 200 characters and display to the screen a count of the occurrences of each of the five vowels (a, e, i, o, u) in the array.

- 2** Design an algorithm that will accept a person's name from the screen entered as surname, first name, separated by a comma. Your program is to display the name as first name, followed by three blanks, followed by the surname.
- 3** Design an algorithm that will prompt for and receive 10 integers from an operator at a terminal, and count the number of integers whose value is less than the average value of the integers. Your program is to display the average integer value and the count of integers less than the average.
- 4** Design an algorithm that will prompt for and receive up to 20 integers from an operator at a terminal and display to the screen the average of the integers. The operator is to input a sentinel of 999 after the required number of integers (up to 20) have been entered.
- 5** Design an algorithm that will read a file of student letter grades and corresponding grade points and load them into two paired arrays, as follows:

Letter grade	Grade points
A	12
B	9
C	6
D	3
F	0

Your program is to read each record on the file (which contains a letter grade followed by a grade point), validate the letter grade (which must be A, B, C, D or F), check that the grade point is numeric, and load the values into the parallel arrays. Your program is to stop processing when the file reaches EOF or the arrays are full. Print an error message if there are more records on the file than elements in the array.

- 6** Design an algorithm that will read a file of student records containing the student's number, name, subject number and letter grade. Your program is to use the letter grade on the student record to retrieve the corresponding grade points for that letter grade from the paired arrays that were established in Problem 5. Print a report showing each student's number, name, subject number, letter grade and grade point. At the end of the report, print the total number of students and the grade point average (total grade points divided by the number of students).
- 7** Design an algorithm that will prompt for and receive an employee number from an operator at a terminal. Your program is to search an array of valid employee numbers to check that the employee number is valid, look up a parallel array to retrieve the corresponding employee name for that number, and display the name to the screen. If the employee number is not valid, an error message is to be displayed.
- 8** An employee file contains records that show an employee's number, name, job code and pay code. The job codes and pay codes are three-digit codes that refer to corresponding job descriptions and pay rates, as in the following tables:

Job code	Job description
A80	Clerk
A90	Word processor
B30	Accountant
B50	Programmer
B70	Systems analyst
C20	Engineer
C40	Senior engineer
D50	Manager

Pay code	Pay rate
01	\$9.00
02	\$9.50
03	\$12.00
04	\$20.00
05	\$23.50
06	\$27.00
07	\$33.00

Your program is to read the employee file, use the job code to retrieve the job description from the job table, use the pay code to retrieve the pay rate from the pay rate table, and print for each record the employee's number, name, job description and pay rate. At the end of the report, print the total number of employees.

- 9 The ACME Oil and Gas Company needs a personnel salary report for its employees, showing their expected increase in salary for the next year. Each record contains the employee's number, name, gross salary, peer performance rating and supervisor performance rating. The percentage increase in salary to be applied to the gross salary is based on two factors: the peer performance rating and the supervisor performance rating, as specified in the following two-dimensional array:

Salary increase percentage table					
Peer performance rating	Supervisor performance rating				
	1	2	3	4	5
1	.013	.015	.017	.019	.021
2	.015	.017	.019	.021	.023
3	.017	.019	.021	.023	.027
4	.019	.021	.023	.025	.030
5	.021	.023	.025	.027	.040

Your program is to retrieve the percentage increase in salary, using the peer performance rating and the supervisor performance rating as indexes to look up the salary increase percentage table. Then calculate the new salary by applying the percentage increase to the gross salary figure. For each employee, print the employee's number, name, this year's gross salary and next year's gross salary. At the end of the report, print the two total gross salary figures.

- 10** Fred's Auto Dealership requires a program that will calculate the sales discount to be applied to a vehicle, based on its year of manufacture and type. The discount is extracted from a two-dimensional table as follows: the year of manufacture of the vehicle is divided into six categories (2003, 2002, 2001, 2000, 1999 and 1998), and the type of car is divided into five categories (mini, small, medium, full-size and luxury). No discount is given for a vehicle older than 1998.

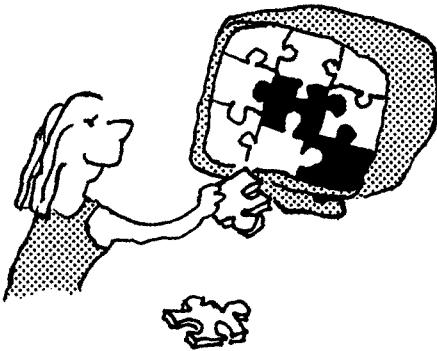
Year of manufacture	Discount percentage				
	Mini	Small	Medium	Full-size	Luxury
	1	2	3	4	5
2003	.050	.055	.060	.065	.070
2002	.040	.045	.050	.055	.060
2001	.030	.035	.040	.045	.050
2000	.020	.025	.030	.035	.040
1999	.010	.015	.020	.025	.030
1998	.005	.010	.015	.020	.025

Your program is to read the vehicle file, which contains the customer number, customer name, make of car, year of manufacture, car type code (1, 2, 3, 4 or 5) and sales price. Use the year of manufacture and the car type code as indexes to retrieve the discount percentage from the discount percentage table. Then apply the discount percentage to the sales price to determine the discounted price of the vehicle. Print all vehicle details, including discounted price.

Chapter

8

First steps in modularisation



Objectives

- To introduce modularisation as a means of dividing a problem into subtasks
- To present hierarchy charts as a pictorial representation of modular program structure
- To discuss intermodule communication, local and global variables, and the passing of parameters between modules
- To develop programming examples that use a simple modularised structure

Outline

- 8.1 Modularisation
 - 8.2 Hierarchy charts or structure charts
 - 8.3 Further modularisation
 - 8.4 Communication between modules
 - 8.5 Using parameters in program design
 - 8.6 Steps in modularisation
 - 8.7 Programming examples using modules
- Chapter summary
Programming problems

8.1 Modularisation

Many solution algorithms have been presented in this book, and all were relatively simple – that is, the finished algorithm was less than one page in length. As programming problems increase in complexity, however, it becomes more and more difficult to consider the solution as a whole. There is a method of looking at complex problems, which involves dividing the problem into smaller parts.

You must first identify the major tasks to be performed in the problem, then divide the problem into sections that represent those tasks. These sections can be considered subtasks or functions. Once the major tasks in the problem have been identified, the programmer may then need to look at each of the subtasks and identify within them further subtasks, and so on. This process of identifying first the major tasks, then further subtasks within them, is known as top-down design or functional decomposition.

By using this top-down design methodology, the programmer is adopting a modular approach to program design. That is, each of the subtasks or functions will eventually become a module within a solution algorithm or program. A module, then, can be defined as a section of an algorithm that is dedicated to a single function. The use of modules makes the algorithm simpler, more systematic, and more likely to be free of errors. Since each module represents a single task, the programmer can develop the solution algorithm task by task, or module by module, until the complete solution has been devised.

Modularisation is the process of dividing a problem into separate tasks, each with a single purpose. Top-down design methodology allows the programmer to concentrate on the overall design of the algorithm without getting too involved with the details of the lower-level modules.

The modularisation process

The division of a problem into smaller subtasks, or modules, is a relatively simple process. When you are defining the problem, write down the activities or processing steps to be performed. These activities are then grouped together to form more manageable tasks or functions, which will eventually form modules. The emphasis when defining the problem must still be on the tasks or functions that need to be performed. Each function will be made up of a number of activities, all of which contribute to the performance of a single task.

A module must be large enough to perform its task, and must include only the operations that contribute to the performance of that task. It should have a single entry, and a single exit with a top-to-bottom sequence of instructions. The name of the module should describe the work to be done as a single specific function. The convention of naming a module by using a verb, followed by a two-word object, is particularly important here, as it helps to identify the separate task or function that the module is to perform. Also, the careful naming of modules using this convention makes the algorithm and resultant code easier to follow. For example, typical module names might be:

```
Print_page_headings  
Calculate_sales_tax  
Validate_input_date
```

By using meaningful module names, you automatically describe the task that the module is to perform, and anyone reading the algorithm can easily see what the module is supposed to do.

The mainline

Since each module performs a single specific task, a mainline routine must provide the master control that ties all the modules together and coordinates their activity. This program mainline should show the main processing functions, and the order in which they are to be performed. It should also show the flow of data and the major control structures. The mainline should be easy to read, be of manageable length and show sound logic structure. Generally, you should be able to read a pseudocode mainline and see exactly what it is being done in the program.

Benefits of modular design

There are a number of benefits from using modular design.

- *Ease of understanding*: each module should perform just one function.
- *Reusable code*: modules used in one algorithm can also be used in other algorithms.
- *Elimination of redundancy*: using modules can help to avoid the repetition of writing out the same segment of code more than once.
- *Efficiency of maintenance*: each module should be self-contained and have little or no effect on other modules within the algorithm.

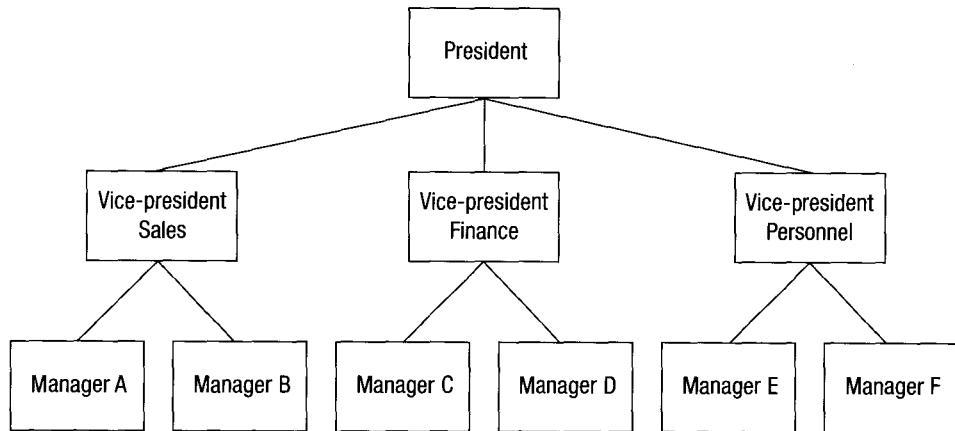
Let us now look at an example, introduced previously, where the solution is made simpler by modularisation.

EXAMPLE 8.1 Process three characters

Design a solution algorithm that will prompt a terminal operator for three characters, accept those characters as input, sort them into ascending sequence and output them to the screen. The algorithm is to continue to read characters until 'XXX' is entered.

A Defining diagram

Input	Processing	Output
char_1	Prompt for characters	char_1
char_2	Accept three characters	char_2
char_3	Sort three characters Output three characters	char_3

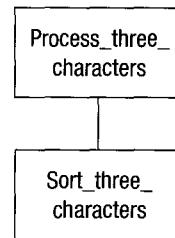


A hierarchy chart may also be referred to as a structure chart or a visual table of contents. The hierarchy chart uses a tree-like diagram of boxes; each box represents a module in the program and the lines connecting the boxes represent the relationship of the modules to others in the program hierarchy. The chart shows no particular sequence for processing the modules; only the modules themselves in the order in which they first appear in the algorithm.

At the top of the hierarchy chart is the controlling module, or mainline. On the next level are the modules that are called directly from the mainline – that is, the modules immediately subordinate to the mainline. On the next level are the modules that are subordinate to the modules on the first level, and so on. This diagrammatic form of hierarchical relationship appears similar to an organisational chart of personnel within a large company.

The mainline will pass control to each module when it is ready for that module to perform its task. The controlling module is said to invoke or call the subordinate module. The controlling module is therefore referred to as the calling module, and the subordinate module the called module. On completion of its task, the called module returns control to the calling module.

The hierarchy chart for Example 8.1 would be relatively simple. It would show a calling module (`Process_three_characters`) and a called module (`Sort_three_characters`), as follows:



8.3 Further modularisation

The above example could also have been designed to use a mainline and *three* modules, one for each of the main processing steps in the defining diagram, as follows.

EXAMPLE 8.2 Process three characters

Design a solution algorithm that will prompt a terminal operator for three characters, accept those characters as input, sort them into ascending sequence and output them to the screen. The algorithm is to continue to read characters until 'XXX' is entered.

A Defining diagram

Input	Processing	Output
char_1	Prompt for characters	char_1
char_2	Accept three characters	char_2
char_3	Sort three characters	char_3
	Output three characters	

B Solution algorithm

The processing steps in the above diagram can be divided into three separate tasks, each task becoming a module in the algorithm. These tasks are: Read three characters, Sort three characters and Print three characters. The solution algorithm would now look like this:

```
Process_three_characters
    Read_three_characters
    DOWHILE NOT (char_1 = 'X' AND char_2 = 'X' AND char_3 = 'X')
        Sort_three_characters
        Print_three_characters
        Read_three_characters
    ENDDO
    END

    Read_three_characters
    Prompt the operator for char_1, char_2, char_3
    Get char_1, char_2, char_3
END
```

8.3 Further modularisation

The above example could also have been designed to use a mainline and *three* modules, one for each of the main processing steps in the defining diagram, as follows.

EXAMPLE 8.2 Process three characters

Design a solution algorithm that will prompt a terminal operator for three characters, accept those characters as input, sort them into ascending sequence and output them to the screen. The algorithm is to continue to read characters until 'XXX' is entered.

A Defining diagram

Input	Processing	Output
char_1	Prompt for characters	char_1
char_2	Accept three characters	char_2
char_3	Sort three characters	char_3
	Output three characters	

B Solution algorithm

The processing steps in the above diagram can be divided into three separate tasks, each task becoming a module in the algorithm. These tasks are: Read three characters, Sort three characters and Print three characters. The solution algorithm would now look like this:

```
Process_three_characters
    Read_three_characters
    DOWHILE NOT (char_1 = 'X' AND char_2 = 'X' AND char_3 = 'X')
        Sort_three_characters
        Print_three_characters
        Read_three_characters
    ENDDO
    END

    Read_three_characters
    Prompt the operator for char_1, char_2, char_3
    Get char_1, char_2, char_3
END
```

```

Sort_three_characters
IF char_1 > char_2 THEN
    temp = char_1
    char_1 = char_2
    char_2 = temp
ENDIF
IF char_2 > char_3 THEN
    temp = char_2
    char_2 = char_3
    char_3 = temp
ENDIF
IF char_1 > char_2 THEN
    temp = char_1
    char_1 = char_2
    char_2 = temp
ENDIF
END

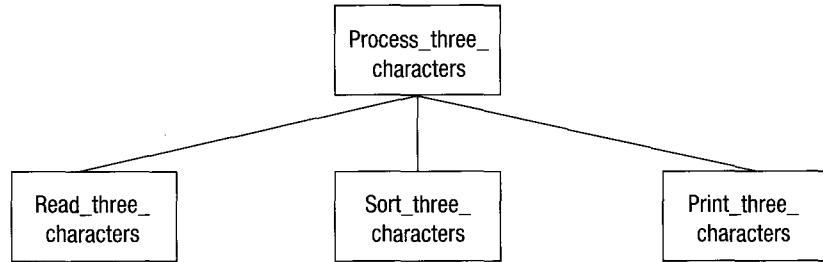
```

```

Print_three_characters
Output to the screen char_1, char_2, char_3
END

```

The hierarchy chart will be made up of a mainline module and three sub-modules, as follows:



B.4 Communication between modules

When designing solution algorithms, you should consider not only the division of the problem into modules but also the flow of information between the modules. The fewer and simpler the communications between modules, the easier it is to understand and maintain one module without reference to other modules. This flow of information, called intermodule communication, can be accomplished by the scope of the variable (local or global data) or the passing of parameters.

Scope of a variable

The scope of a variable is the portion of a program in which that variable has been defined and to which it can be referred. If a list is created of all the modules in which a variable can be referenced, that list defines the scope of the variable. Variables can be global, where the scope of the variable is the whole program, and local, where the scope of the variable is simply the module in which it is defined.

Global data

Global data is data that can be used by all the modules in a program. The scope of a global variable is the whole program, because every module in the program can access the data. The lifetime of a global variable spans the execution of the whole program.

Local data

Variables that are defined within a subordinate module are called local variables. These local variables are not known to the calling module, or to any other module. The scope of a local variable is simply the module in which it is defined. The lifetime of a local variable is limited to the execution of the single subroutine in which it is defined. Using local variables can reduce what is known as program side effects.

Side effects

A side effect is a form of cross-communication of a module with other parts of a program. It occurs when a subordinate module alters the value of a global variable inside a module. Side effects are not necessarily detrimental. However, they do tend to decrease the manageability of a program. A programmer should be aware of their impact.

If a program is amended at any time by a programmer other than the person who wrote it, a change may be made to a global variable. This change could cause side effects or erroneous results because the second programmer is unaware of other modules that also alter that global variable.

Passing parameters

A particularly efficient method of intermodule communication is the passing of parameters or arguments between modules. Parameters are simply data items transferred from a calling module to its subordinate module at the time of calling. When the subordinate module terminates and returns control to its caller, the values in the parameters are transferred back to the calling module. This method of communication avoids any unwanted side effects, as the only interaction between a module and the rest of the program is via parameters.

When a calling module calls a subordinate module in pseudocode, it must consist of the name of the called module with a list of the parameters to be passed to the called module enclosed in parentheses – for example:

```
Print_page_headings (page_count, line_count)
```

The called module will have, following its name, a list of parameters that it expects to receive from the calling module. The names that the respective modules give to their parameters need not be the same – in fact, they often differ because a different programmer has written them – but their number, type and order must be identical.

Formal and actual parameters

Parameter names that appear when a submodule is defined are known as *formal parameters*. Variables and expressions that are passed to a submodule in a particular call are called *actual parameters*. A call to a submodule will include an *actual* parameter list, one variable for each formal parameter name. There is a one-to-one correspondence between formal and actual parameters, which is determined by the relative position in each parameter list. Also, the actual parameter corresponding to a formal parameter must have the same data type as that specified in the declaration of the formal parameter.

For example, a mainline may call a module with an actual parameter list, as follows:

```
Print_page_headings (page_count, line_count)
```

while the module may have been declared with the following formal parameter list:

```
Print_page_headings (page_counter, line_counter)
```

Although the parameter names are different, the actual and formal parameters will correspond.

Value and reference parameters

Parameters may have one of three functions:

- 1 To pass information from a calling module to a subordinate module. The subordinate module would then use that information in its processing, but would not need to communicate any information back to the calling module.
- 2 To pass information from a subordinate module to its calling module. The calling module would then use that parameter in subsequent processing.
- 3 To fulfil a two-way communication role. The calling module may pass information to a subordinate module, where it is amended in some fashion, then passed back to the calling module.

Value parameters

Value parameters only pass data one way – that is, the *value* of the parameter is passed to the called module. When a submodule is called, each actual parameter is assigned into the corresponding formal parameter, and from then on, the two parameters are independent. The called module may not modify the value of the parameter in any way, and, when the submodule has finished processing, the value of the parameter returns to its original value.

Reference parameters

Reference parameters can pass data to a called module where that data may be changed and then passed back to the calling module – that is, the *reference address* of the parameter is passed to the called module. Each actual parameter is an ‘alias’ for the corresponding formal parameter; the two parameters refer to the same object, and changes made through one are visible through the other. As a result, the value of the parameter may be referenced and changed during the processing of the submodule.

There are two principal reasons why you might choose one parameter over the other. First, if the called module is designed to change the value of the actual parameter, then you would pass the parameter by reference. Conversely, to ensure the called routine cannot modify the parameter, you would pass the parameter by value.

Let’s look at an example that illustrates the use of formal and actual parameters and passing by reference.

EXAMPLE 8.3 Increment two counters

Design an algorithm that will increment two counters from 1 to 10 and then output those counters to the screen. Your program is to use a module to increment the counters.

A Defining diagram

Input	Processing	Output
counter1	Increment counters	counter1
counter2	Output counters	counter2

B Solution algorithm

```
Increment_two_counters
  Set counter1, counter2 to zero
  DO I = 1 to 10
    Increment_counter(counter1)
    Increment_counter(counter2)
    Output to the screen counter1, counter2
  ENDDO
  END

  Increment_counter(counter)
  counter = counter + 1
END
```

In this example, the module Increment_counter is called once with the actual parameter, counter1, and once with the actual parameter, counter2. At the time of the call, the value of the actual parameter is associated with the formal parameter, counter, and the module Increment_counter is executed. On completion of the execution of that module, the value of the formal parameter, counter, is associated with the actual parameter, which has changed during the execution of the module. Each time the module Increment_counter is called, the value of the actual parameter is increased by 1. This is an example of passing a parameter by reference.

Hierarchy charts and parameters

Parameters, which pass between modules, can be incorporated into a hierarchy chart or structure chart using the following symbols:



Data parameters contain the actual variables or data items that will be passed between modules.

Status parameters act as program flags and should contain just one of two values: true or false. These program flags or switches are set to true or false, according to a specific set of conditions. They are then used to control further processing.

When designing modular programs, the programmer should avoid using data parameters to indicate status as well, because this can affect the program in two ways:

- 1 It may confuse the reader of the program because a variable has been overloaded – that is, it has been used for more than one purpose.
- 2 It may cause unpredictable errors when the program is amended at some later date, as the maintenance programmer may be unaware of the dual purpose of the variable.

3.5 Using parameters in program design

Let us look again at Example 8.2 and change the solution algorithm so that parameters are used to communicate between modules.

EXAMPLE 8.4 Process three characters

Design a solution algorithm that will prompt a terminal operator for three characters, accept those characters as input, sort them into ascending sequence and output them to the screen. The algorithm is to continue to read characters until 'XXX' is entered.

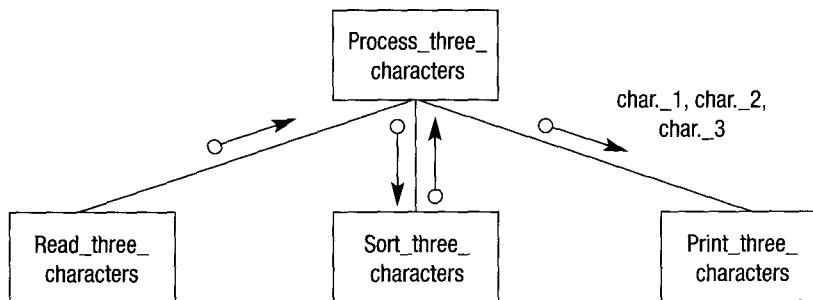
A Defining diagram

Input	Processing	Output
char_1	Prompt for characters	char_1
char_2	Accept three characters	char_2
char_3	Sort three characters	char_3
	Output three characters	

B Group the activities into modules

A module will be created for each processing step in the defining diagram.

C Construct a hierarchy chart



D Establish the logic of the solution algorithm using pseudocode

The mainline will call the module `Read_three_characters`, which will get the three input characters and send them to the mainline as parameters. These parameters will then be passed to the module `Sort_three_characters`, which will sort the three characters and send these sorted values back to the mainline as parameters. The three characters will then be passed to the module `Print_three_characters`, which will print them.

```

Process_three_characters
  Read_three_characters (char_1, char_2, char_3)
  DOWHILE NOT (char_1 = 'X' AND char_2 = 'X' AND char_3 = 'X')
    Sort_three_characters (char_1, char_2, char_3)
    Print_three_characters (char_1, char_2, char_3)
    Read_three_characters (char_1, char_2, char_3)
  ENDDO
END
  
```

```

Read_three_characters (char_1, char_2, char_3)
    Prompt the operator for char_1, char_2, char_3
    Get char_1, char_2, char_3
END

Sort_three_characters (char_1, char_2, char_3)
    IF char_1 > char_2 THEN
        temp = char_1
        char_1 = char_2
        char_2 = temp
    ENDIF
    IF char_2 > char_3 THEN
        temp = char_2
        char_2 = char_3
        char_3 = temp
    ENDIF
    IF char_1 > char_2 THEN
        temp = char_1
        char_1 = char_2
        char_2 = temp
    ENDIF
END

Print_three_characters (char_1, char_2, char_3)
    Output to the screen char_1, char_2, char_3
END

```

Further modularisation

The module Sort_three_characters above contains some repeated code, which looks cumbersome. Further modularisation could be achieved by the introduction of a new module, called Swap_two_characters, which is called by the module Sort_three_characters. The calling module will pass two characters at a time to the submodule, which will swap the position of the parameters and return them, as follows:

```

Process_three_characters
    Read_three_characters (char_1, char_2, char_3)
    DOWHILE NOT (char_1 = 'X' AND char_2 = 'X' AND char_3 = 'X')
        Sort_three_characters (char_1, char_2, char_3)
        Print_three_characters (char_1, char_2, char_3)
        Read_three_characters (char_1, char_2, char_3)
    ENDDO
END

```

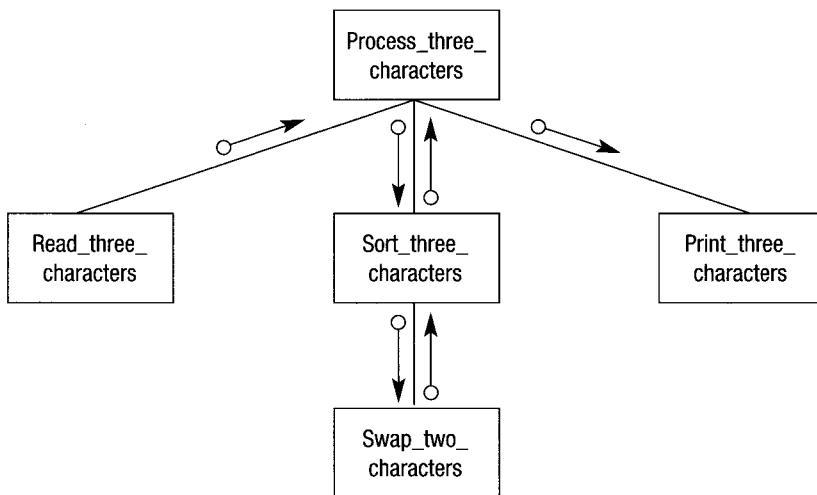
```
Read_three_characters (char_1, char_2, char_3)
    Prompt the operator for char_1, char_2, char_3
    Get char_1, char_2, char_3
END
```

```
Sort_three_characters (char_1, char_2, char_3)
    IF char_1 > char_2 THEN
        Swap_two_characters (char_1, char_2)
    ENDIF
    IF char_2 > char_3 THEN
        Swap_two_characters (char_2, char_3)
    ENDIF
    IF char_1 > char_2 THEN
        Swap_two_characters (char_1, char_2)
    ENDIF
END
```

```
Print_three_characters (char_1, char_2, char_3)
    Output to the screen char_1, char_2, char_3
END
```

```
Swap_two_characters (first_char, second_char)
    temp = first_char
    first_char = second_char
    second_char = temp
END
```

The hierarchy chart for the above solution would look like this:



```

Compute_employee_pay
1   Read_employee_details (employee_details)
2   DOWHILE more records
3   Validate_input_fields (employee_details, valid_input_fields)
4   IF valid_input_fields THEN
      Calculate_employee_pay (employee_details)
      Print_employee_details (employee_details)
    ENDIF
5   Read_employee_details (employee_details)
ENDDO
END

```

E Develop the pseudocode for each successive module in the hierarchy chart

```

Read_employee_details (employee_details)
6   Read emp_no, pay_rate, hrs_worked
END

Validate_input_fields (employee_details, valid_input_fields)
7   set valid_input_fields to true
8   Set error_message to blank
9   IF pay_rate > $25 THEN
      error_message = 'Pay rate exceeds $25.00'
      Print emp_no, pay_rate, hrs_worked, error_message
      valid_input_fields = false
    ENDIF
10  IF hrs_worked > 60 THEN
      error_message = 'Hours worked exceeds 60'
      Print emp_no, pay_rate, hrs_worked, error_message
      valid_input_fields = false
    ENDIF
END

Calculate_employee_pay (employee_details)
11  IF hrs_worked <= 35 THEN
      emp_weekly_pay = pay_rate * hrs_worked
    ELSE
      overtime_hrs = hrs_worked - 35
      overtime_pay = overtime_hrs * pay_rate * 1.5
      emp_weekly_pay = (pay_rate * 35) + overtime_pay
    ENDIF
END

```

```

Print_employee_details (employee_details)
12      Print emp_no, pay_rate, hrs_worked, emp_weekly_pay
        END

```

F Desk check the solution algorithm

The desk checking of an algorithm with modules is no different to the method developed for our previous examples.

- 1 Create some valid input test data.
- 2 List the output that the input data is expected to produce.
- 3 Use a desk check table to walk the data through the mainline of the algorithm to ensure that the expected output is achieved. When a submodule is called, walk the data through each line of that module as well.

1 Input data

Three test cases will be used to test the algorithm.

Record	pay_rate	hrs_worked
employee 1	\$20.00	35
employee 2	\$20.00	40
employee 3	\$50.00	65
EOF		

2 Expected results

employee 1	\$20.00	35	\$700.00
employee 2	\$20.00	40	\$850.00
employee 3	\$50.00	65	Pay rate exceeds \$25.00
employee 3	\$50.00	65	Hours worked exceeds 60

3 Desk check table

Statement number	valid_input_fields	pay_rate	hrs_worked	DOWHILE condition	error_message	emp_weekly_pay	ovt hrs	ovt pay
1, 6		\$20.00	35					
2				true				
3, 7-10	true				blank			
4								
11						\$700		
12		print	print			print		
5, 6		\$20.00	40					
2				true				
3, 7-10	true				blank			
4								
11						\$850	5	\$150
12		print	print			print		
5, 6		\$50.00	65					
2				true				
3, 7, 8	true				blank			
9	false	print	print		Pay rate exceeds \$25			
10	false	print	print		Hours worked exceeds 60			
4								
5, 6		EOF						
2				false				

EXAMPLE 8.6 Product orders report

The Acme Spare Parts Company wants to produce a product orders report from its product orders file. Each record on the file contains the product number of the item ordered, the product description, the number of units ordered, the retail price per unit, the freight charges per unit, and the packaging costs per unit.

Your algorithm is to read the product orders file, calculate the total amount due for each product ordered and print these details on the product orders report.

The amount due for each product is calculated as the product of the number of units ordered and the retail price of the unit. A discount of 10% is allowed on the

amount due for all orders over \$100.00. The freight charges and packaging costs per unit must be added to this resulting value to determine the total amount due.

The output report is to contain headings and column headings as specified in the following chart:

ACME SPARE PARTS

ORDERS REPORT

PAGE xx

PRODUCT NO	PRODUCT DESCRIPTION	UNITS ORDERED	TOTAL AMOUNT DUE
xxxx	xxxxxxxxxx	xxx	xxxxx
xxxx	xxxxxxxxxx	xxx	xxxxx

Each detail line is to contain the product number, product description, number of units ordered and the total amount due for the order. There is to be an allowance of 45 detail lines per page.

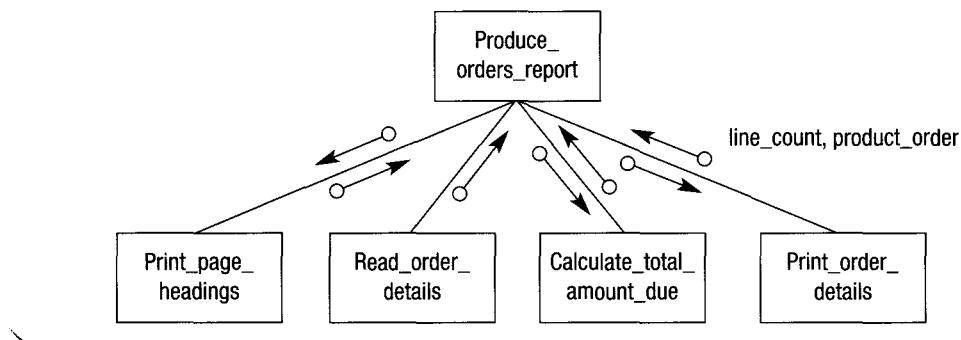
A Define the problem

Input	Processing	Output
Product orders record • prod_number • prod_description • no_of_units • retail_price • freight_charge • packaging_cost	Print headings as required Read order record Calculate total amount due Print order details	Main headings column headings page number detail lines • prod_number • prod_description • no_of_units • total_amount_due

B Group the activities into modules

The four steps in the processing component of the defining diagram will become the four modules in the algorithm. Note that Print_page_headings is a reusable module that is called whenever the report needs to skip to a new page.

C Construct a hierarchy chart



D Establish the logic of the mainline of the algorithm, using pseudocode

When a product record is read, the product's details will be collected into a data structure called product_order, which will be passed between modules as a parameter. The total amount due, when calculated, will also be put into this data structure. The variable line_count will be passed to the modules Print_page_headings and Print_order_details as a parameter.

```
Produce_orders_report
1   Set page_count to zero
2   Print_page_headings (line_count)
3   Read_order_details (product_order)
4   DOWHILE more records
5     IF line_count > 45 THEN
        Print_page_headings (line_count)
      ENDIF
6     Calculate_total_amount_due (product_order)
7     Print_order_details (product_order, line_count)
8     Read_order_details (product_order)
  ENDDO
END
```

E Develop pseudocode for each successive module in the hierarchy chart

The pseudocode for Print_page_headings is standard for a page heading routine that will increment the page counter, print a series of headings and reset the line counter.

```
Print_page_headings (line_count)
9   Add 1 to page_count
10  Print main heading 'ACME SPARE PARTS'
11  Print heading 'ORDERS REPORT'
12  Print column headings 1
13  Print column headings 2
14  Print blank line
15  Set line_count to zero
  END

Read_order_details (product_order)
16  Read product order record
END
```

```

Calculate_total_amount_due (product_order)
17   amount_due = no_of_units * retail_price
18   IF amount_due > $100.00 THEN
        discount = amount_due * 0.1
    ELSE
        discount = zero
    ENDIF
19   amount_due = amount_due - discount
20   freight_due = freight_charge * no_of_units
21   packaging_due = packaging_charge * no_of_units
22   total_amount_due = amount_due + freight_due + packaging_due
END

Print_order_details (product_order, line_count)
23   Print prod_number, prod_description, no_of_units, total_amount_due
24   add 1 to line_count
END

```

F Desk check the solution algorithm

1 Input data

Three test cases will be used to test the algorithm. To test for correct page skipping, we would temporarily reduce the line limit from 45 to a conveniently small number – for example, two.

Record	prod_no	prod_description	no_of_units	retail_price	freight_charge	packaging_charge
1	100	rubber hose	10	\$1.00	\$0.20	\$0.50
2	200	steel pipe	20	\$2.00	\$0.10	\$0.20
3	300	steel bolt	100	\$3.00	\$0.10	\$0.20
EOF						

2 Expected results

ACME SPARE PARTS ORDERS REPORT				PAGE 1
PRODUCT NO	PRODUCT DESCRIPTION	UNITS ORDERED	TOTAL AMOUNT DUE	
100	Rubber hose	10	\$17.00	
200	Steel pipe	20	\$46.00	
300	Steel bolt	100	\$300.00	

3 Desk check table

Statement number	DOWHILE condition	page_count	line_count	prod_no	no_of_units	retail_price	freight_charge	pckg_charge	total_amount_due
1		0							
2, 9-15		1	0						
3, 16				100	10	1.00	0.20	0.50	
4	true								
5									
6, 17-22									17.00
7, 23-24				1	print	print			print
8, 16				200	20	2.00	0.10	0.20	
4	true								
5									
6, 17-22									46.00
7, 23-24				2	print	print			print
8, 16				300	100	3.00	0.10	0.20	
4	true								
5									
6, 17-22									300.00
7, 23-24				3	print	print			print
8				EOF					
4	false								

Chapter summary

This chapter introduced a modular approach to program design. A module was defined as a section of an algorithm that is dedicated to the performance of a single function. Top-down design was defined as the process of dividing a problem into major tasks and then into further subtasks within those major tasks until all the tasks have been identified. Programming examples were provided showing the benefits of using modularisation.

Hierarchy charts were introduced as a method of illustrating the structure of a program that contains modules. Hierarchy charts show the names of all the modules and their hierarchical relationship to each other.

Intermodule communication was defined as the flow of information or data between modules. Local and global variables were introduced, along with the scope of a variable and the side effects of using only global data.

The passing of parameters was introduced as a form of intermodule communication, and the differences between formal and actual parameters, and value and reference parameters, was explained.

The steps in modularisation that a programmer must follow were listed. These were: define the problem; group the activities into subtasks or functions; construct a hierarchy chart; establish the logic of the mainline, using pseudocode; develop the pseudocode for each successive module in the hierarchy chart; and desk check the solution algorithm.

Programming examples using these six steps in modularisation were then developed in pseudocode.

Programming problems

Construct a solution algorithm for the following programming problems. To obtain your final solution, you should:

- define the problem
 - group the activities into modules (also consider the data that each module requires)
 - construct a hierarchy chart
 - establish the logic of the mainline using pseudocode
 - develop the pseudocode for each successive module in the hierarchy chart
 - desk check the solution algorithm.
- 1 Design an algorithm that will prompt for and accept an employee's annual salary, and calculate the annual income tax due on that salary. Income tax is calculated according to the following table and is to be displayed on the screen.

Portion of salary	Income tax rate (%)
\$0 to \$4999.99	0
\$5000 to \$9999.99	6
\$10 000 to \$19 999.99	15
\$20 000 to \$29 999.99	20
\$30 000 to \$39 999.99	25
\$40 000 and above	30

Your program is to continue to process salaries until a salary of zero is entered.

- 2** Design an algorithm that will prompt for and accept four numbers, sort them into ascending sequence and display them to the screen. Your algorithm is to include a module called Order_two_numbers that is to be called from the mainline to sort two numbers at a time.
- 3** Design an algorithm that will prompt for and accept a four-digit representation of the year (for example, 2003). Your program is to determine if the year provided is a leap year and print a message to this effect on the screen. Also print a message on the screen if the value provided is not exactly four numeric digits. Continue processing until a sentinel of 0000 is entered.
- 4** The members of the board of a small university are considering voting for a pay increase for their 25 faculty members. They are considering a pay increase of 8%. However, before doing so, they want to know how much this pay increase will cost. Design an algorithm that will prompt for and accept the current salary for each of the faculty members, then calculate and display their individual pay increases. At the end of the algorithm, print the total faculty payroll before and after the pay increase. and the total pay increase involved.
- 5** Design an algorithm that will produce an employee payroll register from an employee file. Each input employee record contains the employee number, employee's gross pay, income tax, union dues, and other deductions. Your program is to read the employee file and print a detail line for each employee record showing employee number, gross pay, income tax, union dues, other deductions and net pay. Net pay is calculated as gross pay – income tax – union dues – other deductions. At the end of the report, print the total net pay for all employees.
- 6** Design an algorithm that will produce an inventory report from an inventory file. Each input inventory record contains the item number, open inventory amount, amount purchased and amount sold. Your program is to read the inventory file and print a detail line for each inventory record showing item number, open inventory amount, amount purchased, amount sold and final inventory amount. The final inventory amount is calculated as opening inventory amount + purchases – sales. At the end of the report, print the total open inventory amount, the total amount purchased, the total amount sold and the total final inventory amount.
- 7** Design an algorithm that will produce a savings account balance report from a customer savings account file. Each input savings account record contains the account number, balance forward, deposits (sum of all deposits), withdrawals (sum of all withdrawals) and interest earned. Your program is to read the savings account file and print a detail line for each savings account record showing account number, balance forward, deposits, withdrawals, interest earned and final account balance. The final account balance is calculated as balance forward + deposits – withdrawals + interest. A heading is to appear at the top of each page and allowance is to be made for 45 detail lines per page. At the end of the report, print the total balances forward, total deposits, total withdrawals, total interest earned and total final account balances.

- 8 Design an algorithm that will read a file of sales volume records and print a report showing the sales commission owing to each salesperson. Each input record contains salesperson number, name and that person's volume of sales for the month. The commission rate varies according to sales volume, as follows:

On sales volume (\$)	Commission rate (%)
\$0.00-\$200.00	5
\$200.01-\$1000.00	8
\$1000.01-\$2000.00	10
\$2000.01 and above	12

The calculated commission is an accumulated amount according to the sales volume figure. For example, the commission owing for a sales volume of \$1200.00 would be calculated as follows:

$$\text{Commission} = (200 * 5\%) + ((1000 - 200) * 8\%) + ((1200 - 1000) * 10\%)$$

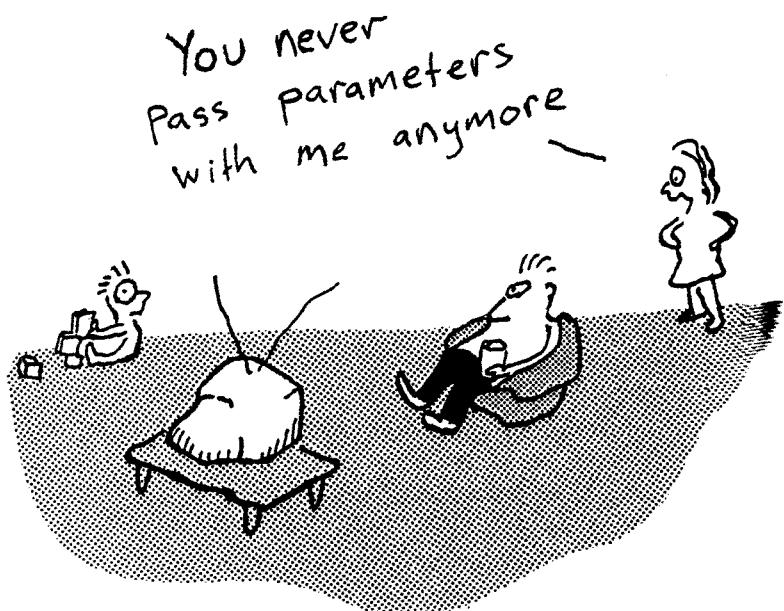
Your program is to print the salesperson's number, name, volume of sales and calculated commission, with the appropriate column headings.

- 9 Design an algorithm that will prompt for and receive your current cheque book balance, followed by a number of financial transactions. Each transaction consists of a transaction code and a transaction amount. The transaction code can be a deposit ('D') or a cheque ('C'). Your program is to add each deposit transaction amount to the balance and subtract each cheque transaction amount. After each transaction is processed, a new running balance is to be displayed on the screen, with a warning message if the balance becomes negative. When there are no more transactions, a 'Q' is to be entered for transaction code to signify the end of the data. Your algorithm is then to display the initial and final balances, along with a count of the number of cheques and deposits processed.
- 10 At Olympic diving competition level, 10 diving judges award a single mark (with one decimal place) for each dive attempted by a diving competitor. This mark can range from 0 to 10. Design an algorithm that will receive a score from the 10 judges and calculate the average score. The screen should display the following output:

Judge	1	2	3	4	5	6	7	8	9	10
Mark	6.7	8.1	5.8	7.0	6.6	6.0	7.6	6.1	7.2	7.0
Score for the dive	6.81									

Chapter 9

Further modularisation, cohesion and coupling



Objectives

- To further develop modularisation using a more complex problem
- To introduce cohesion as a measure of the internal strength of a module
- To introduce coupling as a measure of the extent of information interchange between modules

Outline

- 9.1 Steps in modularisation
- 9.2 Module cohesion
- 9.3 Module coupling
- Chapter summary
- Programming problems

9.1 Steps in modularisation

In Chapter 8, the six steps to be followed when using top-down modular design to develop a solution algorithm for a problem were established as follows:

- 1 Define the problem by dividing it into its three components: input, output and processing. The processing component should consist of a list of activities to be performed.
- 2 Group the activities into subtasks or functions to determine the modules that will make up the program. Remember that a module is dedicated to the performance of a single function. Not all the activities may be identified at this stage. Only the modules of the first level of the hierarchy chart may be identified, with other more subordinate modules developed later.
- 3 Construct a hierarchy chart to illustrate the modules, and their relationship to each other. Once the structure (or organisation) of the program has been developed, the order of processing of the modules can be considered. Intermodule communication and the passing of parameters can also be considered at this step.
- 4 Establish the logic of the mainline of the algorithm in pseudocode. This mainline should contain some initial processing before the loop, some processing within the loop, and some final processing after exiting the loop. It should contain calls to the major processing modules of the program, and should be easy to read and understand.
- 5 Develop the pseudocode for each successive module in the hierarchy chart. The modularisation process is complete when the pseudocode for each module on the lowest level of the hierarchy chart has been developed.
- 6 Desk check the solution algorithm. This is achieved by first desk checking the mainline, and then each subordinate module in turn.

This chapter develops a solution algorithm for a more complex problem – that is, a problem which, when divided into submodules, has more than one level in the hierarchy chart.

EXAMPLE 9.1 Calculate vehicle registration costs

A program is required to calculate and print the registration cost of a new vehicle that a customer has ordered. The program is to be interactive. That is, all the input details will be provided at a terminal on the salesperson's desk. The program is to get the input details, calculate the federal tax, calculate the registration costs, calculate the total amount payable and then output the required information to the screen.

The input details required are:

- owner's name
- vehicle make

- vehicle model
- weight (in kg)
- body type (sedan or wagon)
- private or business code ('P' or 'B')
- wholesale price of vehicle.

The federal tax is calculated at the rate of \$2.00 for each \$100.00, or part thereof, of the wholesale price of the vehicle.

The vehicle registration cost is calculated as the sum of the following charges:

Registration fee	\$27.00	
Tax levy	Private	5% of wholesale price
	Business	7% of wholesale price
Weight tax	Private	1% of weight (converted to \$)
	Business	3% of weight (converted to \$)
Insurance premium	Private	1% of wholesale price
	Business	2% of wholesale price

The total amount payable = federal tax + total registration charges. The information to be printed to the screen is as follows:

Vehicle make:

Vehicle model:

Body type:

Registration fee:

Tax levy:

Weight tax:

Insurance premium:

Total registration charges:

Federal tax:

Total amount payable:

The program is to process registration costs until an owner's name of 'XXX' is entered. None of the other entry details will be required after the value 'XXX' has been entered.

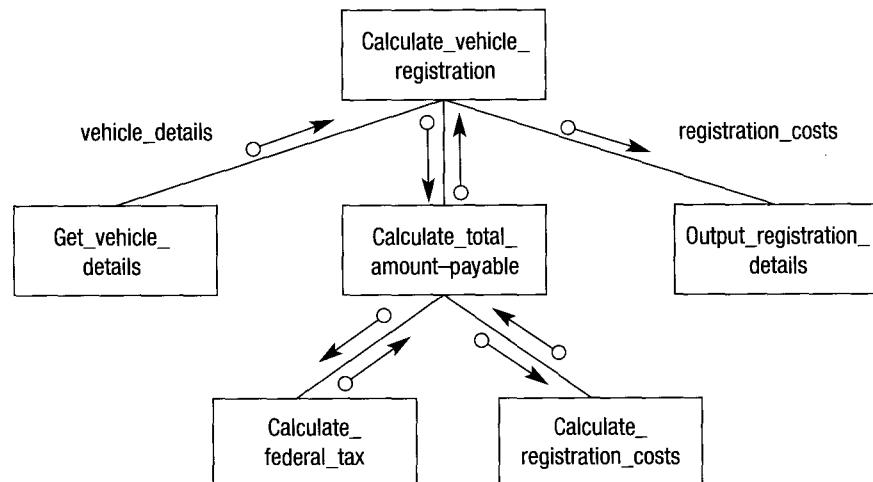
A Define the problem

Input	Processing	Output
owners_name	Get input details	vehicle_make
vehicle_make	Calculate federal_tax	vehicle_model
vehicle_model	Calculate registration_costs	body_type
weight	Calculate total_amount_payable	registration_fee
body_type	Output details to screen	tax_levy
usage_code		weight_tax
wholesale_price		insurance_premium
		total_registration_charges
		federal_tax
		total_amount_payable

B Group the activities into modules

- 1 Get input details: there are a number of input fields to read from the screen, so a module called Get_vehicle_details is created to perform this function. Note that the read from the screen of the owner's name must be separate, as it is the entry of 'XXX' in this field that will cause the repetition to cease.
- 2 The total amount payable is calculated in the module Calculate_total_amount_payable. This module will call two subordinate modules to calculate the federal tax and registration costs respectively.
- 3 Display details to screen: there are a number of output fields to display on to the screen, so a module called Output_registration_details is created to perform this function.

C Construct a hierarchy chart



The hierarchy chart illustrates the structure that the algorithm will take. The mainline will call three subordinate modules: Get_vehicle_details, Calculate_total_amount_payable and Output_registration_details. The module Calculate_total_amount_payable will call two modules: Calculate_federal_tax and Calculate_registration_costs.

D Establish the logic of the mainline of the algorithm using pseudocode

The input vehicle details will be collected into a data structure called vehicle_details. All the registration costs will be collected into a data structure called registration_costs. The total amount payable will also be put into the data structure, registration_costs. These data structures will then be passed between modules as parameters.

```
Calculate_vehicle_registration
1   Read owners_name
2   DOWHILE owners_name NOT = 'XXX'
3       Get_vehicle_details (vehicle_details)
4       Calculate_total_amount_payable (vehicle_details, registration_costs)
5       Output_registration_details (vehicle_details, registration_costs)
6   Read owners_name
    ENDDO
END
```

The mainline consists of a Read before the loop, calls to its three subordinate modules within the loop, and a Read just before the end of the loop.

E Develop the pseudocode for each successive module in the hierarchy chart

- 1 Get_vehicle_details is a module that prompts for and reads the required fields and collects them in a data structure called vehicle_details. The owner's name is read separately in the mainline.

```
Get_vehicle_details (vehicle_details)
7   Prompt and Get vehicle_make
8   Prompt and Get vehicle_model
9   Prompt and Get_weight
10  Prompt and Get_body_type
11  Prompt and Get_usage_code
12  Prompt and Get_wholesale_price
END
```

2 Calculate_total_amount_payable calls two other modules.

```
Calculate_total_amount_payable (vehicle_details, registration_costs)
13   Calculate_federal_tax (vehicle_details, federal_tax)
14   Calculate_registration_costs (vehicle_details, registration_costs)
15   total_amount_payable = federal_tax + total_registration_charges
END
```

3 Calculate_federal_tax calculates the federal tax, which is payable at the rate of \$2.00 for each \$100.00 or part thereof of the wholesale price of the car. (A variable called tax_units is used to count the number of whole \$100 units.)

```
Calculate_federal_tax (vehicle_details, federal_tax)
16   Set tax_units = zero
17   DOWHILE wholesale_price > $100
18     wholesale_price = wholesale_price - 100
19     increment tax_units by 1
20   ENDDO
21   federal_tax = (tax_units + 1) * $2.00
22 END
```

4 Calculate_registration_costs contains all the processing required to calculate the costs of registration, and collects these registration costs in a data structure called registration_costs.

```
Calculate_registration_costs (vehicle_details, registration_costs)
21   registration_fee = $27.00
22   IF usage_code = 'P' THEN
23     tax_levy = wholesale_price * 0.05
     weight_tax = weight * 0.01
     insurance_premium = wholesale_price * 0.01
   ELSE
     tax_levy = wholesale_price * 0.07
     weight_tax = weight * 0.03
     insurance_premium = wholesale_price * 0.02
   ENDIF
23   total_registration_charges = registration_fee + tax_levy + weight_tax +
     insurance_premium
END
```

5 Output_registration_details outputs the required fields to the screen.

```
Output_registration_details (vehicle_details, registration_costs)
24   Output vehicle_make
25   Output vehicle_model
26   Output body_type
27   Output registration_fee
28   Output tax_levy
29   Output weight_tax
30   Output insurance_premium
31   Output total_registration_charges
32   Output federal_tax
33   Output total_amount_payable
END
```

F Desk check the solution algorithm

1 Input data

As there are two branches in the logic of the program, two test cases should be sufficient to test the algorithm. Only the relevant input fields will be provided.

Record	weight	usage_code	wholesale_price
Record1	1000	P	30 000
Record2	2000	B	20 000
XXX			

2 Expected results

Vehicle make:	Record 1	Record 2
Vehicle model:	Record 1	Record 2
Body type:	Record 1	Record 2
Registration fee:	\$27.00	\$27.00
Tax levy:	\$1500.00	\$1400.00
Weight tax:	\$10.00	\$60.00
Insurance premium:	\$300.00	\$400.00
Total registration charges:	\$1837.00	\$1887.00
Federal tax:	\$600.00	\$400.00
Total amount payable:	\$2437.00	\$2287.00

3 Desk check table

The desk checking will be of the main processing steps of the mainline. When a call to a module is made, all the processing steps of the module are recorded on one line of the desk check table. This, in effect, checks both the mainline logic and the modules at the same time.

Statement number	owners_name	DOWHILE condition	weight	usage_code	whole-sale_price	federal_tax	total_registration	total_amount_payable
1	Record1							
2		true						
7-12			1000	P	30 000			
4, 13-23						600	1837	2437
5, 24-33						output	output	output
6	Record2							
2		true						
3, 7-12			2000	B	20 000			
4, 13-23						400	1887	2287
5, 24-33						output	output	output
6	XXX							
2		false						

9.2 Module cohesion

A module has been defined as a section of an algorithm that is dedicated to the performance of a single function. It contains a single entry and a single exit, and the name chosen for the module should describe its function.

Programmers often need guidance in determining what makes a good module. Common queries include: 'How big should a module be?', 'Is this module too small?' and 'Should I put all the read statements in one module?'

There is a method you can use to remove some of the guesswork when establishing modules. You can look at the cohesion of the module. Cohesion is a measure of the internal strength of a module – that is, how closely the elements or statements of a module are associated with each other. The more closely the elements of a module are associated, the higher the cohesion of the module. Modules with high cohesion are considered good modules, because of their internal strength.

Edward Yourdon and Larry Constantine¹ established seven levels of cohesion and placed them in a scale from the weakest to the strongest.

¹ Edward Yourdon and Larry Constantine, *Structured Design: Fundamentals of a Discipline of Computer Program and System Design* (Prentice-Hall, 1979).

Cohesion level	Cohesion attribute	Resultant module strength
Coincidental	Low cohesion	Weakest
Logical		
Temporal		
Procedural		
Communicational		
Sequential		
Functional	High cohesion	Strongest

Each level of cohesion in the table will be discussed in this chapter, and pseudocode examples that illustrate each level will be provided.

Coincidental cohesion

The weakest form of cohesion a module can have is coincidental cohesion. It occurs when elements are collected into a module simply because they happen to fall together. There is no meaningful relationship between the elements at all, and so it is difficult to concisely define the function of the module.

Fortunately, these types of modules are rare in today's programming practice. They typically used to occur as a result of one of the following conditions:

- An existing program may have been arbitrarily segmented into smaller modules because of hardware constrictions on the operation of the program.
- Existing modules may have been arbitrarily subdivided to conform to a badly considered programming standard (for example, each module should have no more than 50 program statements).
- A number of existing modules may have been combined into one module either to reduce the number of modules in a program or to increase the number of statements in a module to a particular minimum number.

Here is a pseudocode example of a module that has coincidental cohesion:

```

File_processing
  Open employee updates file
  Read employee record
  Print_page_headings
  Open employee master file
  Set page_count to one
  Set error_flag to false
END

```

Notice that the instructions within the module have no meaningful relationship to each other.

Logical cohesion

Logical cohesion occurs when the elements of a module are grouped together according to a certain class of activity. That is, the elements fall into some general category because they all do the same kind of thing.

An example might be a module that performs all the read statements for three different files: a sort of 'Read_all_files' module. In such a case, the calling module would need to indicate which of the three files it required the called module to read, by sending a parameter.

A module such as this is slightly stronger than a coincidentally cohesive module, because the elements are somewhat related. However, logically cohesive modules are usually made up of a number of smaller, independent sections, which should exist independently rather than be combined together because of a related activity. Often when a module such as this is called, only a small subset of the elements within the module will be executed.

A pseudocode example for a 'Read_all_files' module might look like this:

```
Read_all_files (file_code)
CASE of file_code
1 : Read customer transaction record
    IF not EOF
        increment customer_transaction_count
    ENDIF
2 : Read customer master record
    IF not EOF
        increment customer_master_count
    ENDIF
3 : Read product master record
    IF not EOF
        increment product_master_count
    ENDIF
ENDCASE
END
```

Notice that the three Read instructions in this module perform three separate functions.

Temporal cohesion

Temporal cohesion occurs when the elements of a module are grouped together because they are related by time. Typical examples are initialisation and finalisation modules, where elements are placed together because they perform certain housekeeping functions at the beginning or end of a program.

A temporally cohesive module can be considered a logically cohesive module, where time is the related activity. However, it is slightly stronger than a logically cohesive module because most of the elements in a time-related module are executed each time the module is called. Usually, however, the elements are not all related to the same function.

A pseudocode example of a temporally cohesive module might look like this:

```
Initialisation
    Open transaction file
    Issue prompt 'Enter today's date - DDMMYY'
    Read todays_date
    Set transaction_count to zero
    Read transaction record
    IF not EOF
        increment transaction_count
    ENDIF
    Open report file
    Print_page_headings
    Set report_total to zero
END
```

Notice that the elements of the module perform a number of functions.

Procedural cohesion

Procedural cohesion occurs when the elements of a module are related because they operate according to a particular procedure. That is, the elements are executed in a particular sequence so that the objectives of the program are achieved. As a result, the modules contain elements related more to program procedure than to program function.

A typical example of a procedurally cohesive module is the mainline of a program. The elements of a mainline are grouped together because of a particular procedural order.

The weakness of procedurally cohesive modules is that they cut across functional boundaries. That is, the procedure may contain only part of a function at one level, but at the same time may contain multiple functions at a lower level, as in the pseudocode example below:

```
Read_student_records_and_total_student_ages
    Set number_of_records to zero
    Set total_age to zero
    Read student record
    DOWHILE more records exist
        Add age to total_age
        Add 1 to number_of_records
        Read student record
    ENDDO
    Print number_of_records, total_age
END
```

Note that the use of the word 'and' in the module name indicates that the module performs more than one function.

Communicational cohesion

Communicational cohesion occurs when the elements of a module are grouped together because they all operate on the same (central) piece of data. Communicationally cohesive modules are commonly found in business applications because of the close relationship of a business program to the data it is processing. For example, a module may contain all the validations of the fields of a record, or all the processing required to assemble a report line for printing.

Communicational cohesion is acceptable because it is data-related. It is stronger than procedural cohesion because of its relationship with the data, rather than the control-flow sequence.

The weakness of a communicationally cohesive module lies in the fact that usually a combination of processing for a particular piece of data is performed, as in this pseudocode example:

```
Validate_product_record
    IF transaction_type NOT = '0' THEN
        error_flag = true
        error_message = 'invalid transaction type'
        Print_error_report
    ENDIF
    IF customer_number is NOT numeric THEN
        error_flag = true
        error_message = 'invalid customer number'
        Print_error_report
    ENDIF
    IF product_no = blanks
        OR product_no has leading blanks THEN
            error_flag = true
            error_message = 'invalid product no'
            Print_error_report
    ENDIF
END
```

Sequential cohesion

Sequential cohesion occurs when a module contains elements that depend on the processing of previous elements. That is, it might contain elements in which the output data from one element serves as input data to the next. Thus, a sequentially cohesive module is like an assembly line: a series of sequential steps that perform successive transformations of data.

Sequential cohesion is stronger than communicational cohesion because it is more problem-oriented. Its weakness lies only in the fact that the module may perform multiple functions or fragments of functions.

Here is a pseudocode example of a sequentially cohesive module:

```

Process_purchases
    Set total_purchases to zero
    Get number_of_purchases
    DO loop_index = 1 to number_of_purchases
        get purchase
        add purchase to total_purchases
    ENDDO
    sales_tax = total_purchases * sales_tax_percent
    amount_due = total_purchases + sales_tax
END

```

Note that this module first calculates total_purchases and then uses the variable total_purchases in the subsequent calculation of amount_due.

Functional cohesion

Functional cohesion occurs when all the elements of a module contribute to the performance of a single specific task. The module can be easily named by a single verb followed by a two-word object.

Mathematically oriented modules are a good example of functional cohesion, as the elements making up the module form an integral part of the calculation.

A pseudocode example of a functionally cohesive module is the module Calculate_sales_tax:

```

Calculate_sales_tax
    IF product is sales tax exempt THEN
        sales_tax = 0
    ELSE
        IF product_price < $50.00 THEN
            sales_tax = product_price * 0.25
        ELSE
            IF product_price < $100.00 THEN
                sales_tax = product_price * 0.35
            ELSE
                sales_tax = product_price * 0.5
            ENDIF
        ENDIF
    ENDIF
END

```

Summary of cohesion levels

When designing the structure of an algorithm, you should try to form modules that have a single problem-related function. If functional cohesion is achieved, the modules will be more independent, easier to read and understand, and more maintainable than modules with lesser cohesion.

In some cases, it is not easy to construct a program where every module has functional cohesion. Some modules may contain lower levels of cohesion, or

even a combination of types of cohesion. This may not be a problem. However, it is important that you can recognise the various cohesion levels and justify a module with a lower cohesion in a particular set of circumstances.

Your prime consideration is to produce modules and programs that are easy to understand and modify. The higher the cohesion of the modules, the more likely it is that you have achieved this aim.

9.3 Module coupling

When designing a solution algorithm, look not only at the cohesion of modules but also at the flow of information between modules. You should aim to achieve module independence – that is, modules that have fewer and simpler connections with other modules. These connections are called interfaces or couples.

Coupling is a measure of the extent of information interchange between modules. Tight coupling implies large dependence on the structure of one module by another. Because there are a higher number of connections, there are many paths along which errors can extend into other parts of the program.

Loose coupling is the opposite of tight coupling. Modules with loose coupling are more independent and easier to maintain.

Glenford Myers² devised a coupling scale similar to Yourdon and Constantine's cohesion scale.

Coupling level	Coupling attribute	Resultant module design quality
Common	Tight coupling	Poorest
External		
Control		
Stamp		
Data	Loose coupling	Best

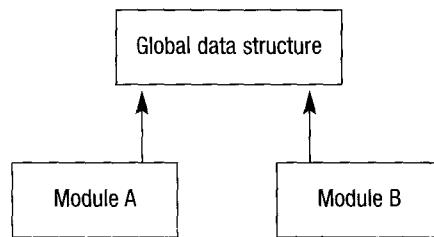
The five levels of coupling are listed in a scale from the poorest module design quality to the best. Each of the levels of coupling will be discussed, and pseudocode examples which illustrate each level will be provided. Note that these levels of coupling are not definitive. They are merely the coupling levels that Myers believes can exist in modular programs.

Common coupling

Common coupling occurs when modules reference the same global data structure. (A data structure is a collection of related data items, such as a record or an array.) When modules experience common coupling, a global data structure is shared by the modules.

² Glenford Myers, *Composite Structured Design* (Van Nostrand Reinhold, 1978).

This means that the data can be accessed and modified by any module in the program, which can make the program difficult to read.



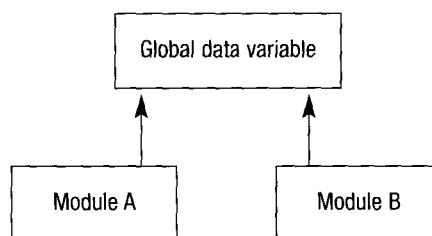
The following pseudocode example shows two modules that experience common coupling because they access the same global data structure (the customer record):

```
A    Read_customer_record
      Read customer record
      IF EOF THEN
        set EOF_flag to true
      ENDIF
    END

B    Validate_customer_record
      IF customer_number is NOT numeric THEN
        error_message = 'invalid customer number'
        Print_error_report
      ENDIF
      :
      :
    END
```

External coupling

External coupling occurs when two or more modules access the same global data variable. It is similar to common coupling except that the global data is an elementary data item, rather than a data structure. Because the global data has a simpler structure, external coupling is considered to be looser than common coupling.



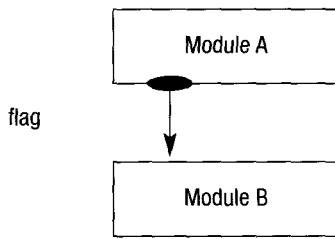
The following pseudocode example shows two modules that exhibit external coupling because they share the same global data item (sales_tax).

```
A    Calculate_sales_tax
    IF product_is_sales_exempt THEN
        sales_tax = 0
    ELSE
        IF product_price < $50.00 THEN
            sales_tax = product_price * 0.25
            :
            :
        ENDIF
    ENDIF
END

B    Calculate_amount_due
    :
    :
    amount_due = total_amount + sales_tax
END
```

Control coupling

Control coupling occurs when a module passes another module a control variable that is intended to control the other module's logic. These control variables are referred to as program flags, or switches, and are passed between modules in the form of parameters.



The weakness of control-coupled modules is that the passing of the control field between modules implies that one module is aware of the internal logic of the other.

The following pseudocode example shows two modules that are logically cohesive because of the passing of the control parameter (input_code):

```
A    Process_input_code
    Read input_code
    Choose_appropriate_action (input_code)
    :
    :
END
```

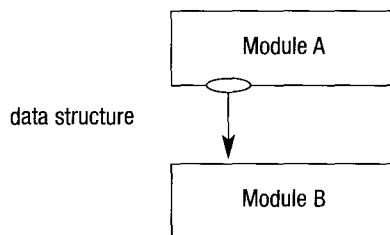
```

B      Choose_appropriate_action (input_code)
      CASE OF input_code
      1 : Read employee record
      2 : Print_page_headings
      3 : Open employee master file
      4 : Set page_count to zero
      5 : error_message = 'Employee number not numeric'
      ENDCASE
END

```

Stamp coupling

Stamp coupling occurs when one module passes a non-global data structure to another module in the form of a parameter.



Stamp-coupled modules demonstrate loose coupling and offer good module design quality. The only relationship between the two modules is the passing of the data structure between them; there is no need for either module to know the internal logic of the other.

The following pseudocode example shows two modules that are stamp coupled because of the passing of the data structure current_record.

```

A      Process_transaction_record
      :
      :
      IF transaction record is for a male THEN
          Process_male_student (current_record)
      ELSE
          Process_female_student (current_record)
      ENDIF
      :
      :
END

```

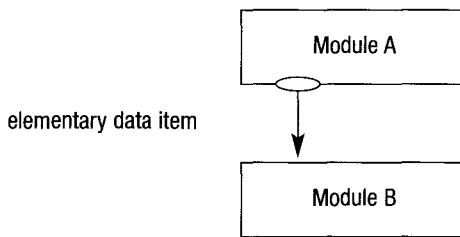
```

B    Process_male_student (current_record)
      increment male_student_count
      IF student_age > 21 THEN
          increment mature_male_count
      ENDIF
      :
      :
END

```

Data coupling

Data coupling occurs when a module passes a non-global data variable to another module. It is similar to stamp coupling except that the non-global data variable is an elementary data item, not a data structure.



Modules that are data coupled demonstrate the loosest coupling and offer the best module design qualities. The only relationship between the two modules is the passing of one or more elementary data items between them.

The following pseudocode example shows two modules that are data coupled because they pass the elementary data items total_price and sales_tax.

```

A    Process_customer_record
      :
      :
      Calculate_sales_tax (total_price, sales_tax)
      :
END

B    Calculate_sales_tax (total_price, sales_tax)
      IF total_price < $10.00 THEN
          sales_tax = total_price * 0.25
      ELSE
          IF total_price < $100.00 THEN
              sales_tax = total_price * 0.3
          ELSE
              sales_tax = total_price * 0.4
          ENDIF
      ENDIF
END

```

A summary of coupling levels

When designing solution algorithms, you should aim towards module independence and a minimum of information interchange between modules.

If the programming language allows it, try to uncouple each module from its surroundings by:

- 1 passing data to a subordinate module in the form of parameters, rather than using global data
- 2 writing each subordinate module as a self-contained unit that can: accept data passed to it; operate on it without reference to other parts of the program; and pass information back to the calling module, if required.

Chapter summary

This chapter revised the six steps to be followed when using top-down modular design to develop a solution to a typical programming problem. A solution algorithm using three levels of modularisation in its hierarchy chart was then developed.

Cohesion and coupling were introduced and must be considered when designing modular programs. A program that has been well designed has modules which are independent, easy to read and easily maintained. Such modules are likely to exhibit high cohesion and loose coupling.

Cohesion is a measure of the internal strength of a module. The higher the cohesion, the better the module. Seven levels of cohesion were given and each level was discussed, with a pseudocode example provided.

Coupling is a measure of the extent of information interchange between modules. The fewer the connections between the modules, the more loosely they are coupled, offering good module design quality. Five levels of coupling were given and each level was discussed, with a pseudocode example provided.

Programming problems

Construct a solution algorithm for the following programming problems. To obtain your final solution, you should:

- define the problem
- group the activities into modules (also consider the data that each module requires)
- construct a hierarchy chart
- establish the logic of the mainline using pseudocode
- develop the pseudocode for each successive module in the hierarchy chart
- desk check the solution algorithm.

- 1** Design an algorithm that will produce a reorder list of products from a product inventory file. Each input product record contains the item number, the quantity on hand for the item, the quantity on order, the minimum inventory level for that item, and an obsolete code ('X' if the item is obsolete, blank if it is not).

Your program is to read the product file and determine which items are to be reordered. An item is to be reordered if it is not obsolete and if the quantity of the item currently on hand, plus the amount on order, is less than its minimum inventory level. Print a detail line for each item to be reordered, listing the item number, quantity on hand, quantity on order and minimum inventory level. Print headings and column headings at the top of each page, allowing for 45 detail lines per page, and at the end of the report, the total number of items to be reordered.

- 2** Design an algorithm that will produce a list of selected student names from a student file. Each input student record contains the student's number, the student's name, the number of semester hours the student is currently taking, and the age of the student.

Your program is to read the student file and prepare a list of names of all full-time students (students taking 12 or more semester hours) who are 30 years of age or older. If a student appearing on the list is taking more than 20 semester hours, place three asterisks after the student's name. Print a detail line for each student selected, listing student number, student name, age and number of semester hours. Print headings and column headings at the top of each page, allowing for 45 detail lines per page, and at the end of the report, the total number of selected students.

- 3** Design an algorithm that will produce a list of selected customers from a customer file. Each input record contains the customer's name, current monthly sales and year-to-date sales. Each time the program is run, a parameter record containing a dollar amount is read in as the first record in the file.

Your program is to read the parameter record, followed by the customer file, and prepare a list of customers whose purchases total at least \$10 000 in the current month. Customers should also be included in the list if their year-to-date sales are at least as great as the amount read in as a parameter.

Print a detail line for each customer, listing customer's name, current monthly sales, year-to-date sales and the parameter amount. Print headings and column headings at the top of each page, allowing for 45 detail lines per page, and at the end of the report, the total number of selected customers.

- 4** Design an algorithm that will produce a tax report from an employee income file. Each input record contains the employee number and the employee's taxable income. There is one record for each employee.

Your program is to read the employee income file and calculate the tax owing on that employee's taxable income, according to the following table:

Taxable income	Tax payable
\$0-\$14 999.99	20% of taxable income
\$15 000-\$29 999.99	\$3000 + 30% of amount greater than \$15 000
\$30 000-\$49 999.99	\$7500 + 40% of amount greater than \$30 000
\$50 000-\$74 999.99	\$15 500 + 50% of amount greater than \$50 000
\$75 000 and above	\$28 000 + 75% of amount greater than \$75 000

Print a detail line for each employee, listing the employee number, taxable income and tax payable. Print headings and column headings at the top of each page, allowing for 45 detail lines per page, and at the end of the report, the total taxable income and total tax payable.

- 5 Design an algorithm that will create a data validation edit report from an inventory file. Each field on the inventory record is to be validated as follows:

Field	Format
Stock number	Numeric
Item number	Numeric (1-5000)
Description	Alphanumeric
Quantity on hand	Numeric (500-999)
Quantity on order	Numeric (500-999)
Price per unit	Numeric (10-1000)
Inventory reorder level	Numeric (50-500)

If a field is found to be in error, print a line in the data validation edit report showing the stock number, the item number and an appropriate message, as indicated in the diagram below. There may be multiple messages for the one record. Print headings and column headings at the top of each page, allowing for 45 detail lines per page.

Data validation edit report

Stock number	Item number	Message
00742	4003	Quantity on hand out of range
00853	5201	Quantity on order out of range
00932	1007	Price per unit not numeric
00932	1007	Reorder level not valid

- 6 Design an algorithm that will produce a list of successful applicants who have applied at the local Riff Raff department store for credit. Each input record contains the applicant's name, employment status, years in current job (if any), years at current residence, monthly wages, amount of non-mortgage debt, and number of children.

Your program is to read the applicant's file and determine whether or not each applicant will be granted credit. The store grants credit to a person who has worked in the same job for more than one year, as well as someone who is employed and has lived in the same location for at least two years. However, if a person owes more than two months' wages in non-mortgage debt or has more than six children, credit is denied.

Print a detail line for each applicant, listing the applicant's name and whether or not that applicant has been granted credit. Print headings and column headings at the top of each page, allowing for 45 detail lines per page, and at the end of the report, the total number of successful applicants.

- 7 Design an algorithm that will calculate the percentage discount allowed on a customer's purchase, from a customer file. Each input record contains the customer's number, the customer's name, the class code and, for wholesale customers, the number of units purchased and the distance from the warehouse. The class code contains either 'R' for retail customers or 'W' for wholesale customers. The end-of-file of the input file is denoted by a customer number of 99999.

Your program is to read the customer file and determine the percentage discount, if any, on a customer's purchase, according to the following guidelines. Retail customers get no discount. Wholesale customers who purchase fewer than 10 units also receive no discount. A 10% discount is given to wholesale customers who purchase at least 10 but fewer than 30 units, and are within 50 kilometres of the distributor's warehouse. If a wholesale customer purchases between 10 and 30 units but is more than 50 kilometres away, only a 5% discount is allowed. Wholesale customers who purchase 30 or more units receive a 15% discount if they are within 50 kilometres of the warehouse. If they are more than 50 kilometres away, they only get a 10% discount.

Print a detail line for each customer, listing the customer number, customer name, class code, number of units purchased and the percentage discount (if any) to be applied on a customer's purchases.

Print headings and column headings at the top of each page, allowing for 45 detail lines per page.

- 8 The Tidy Phones telephone company's charges file contains records for each call made by its Metroville subscribers during a month. Each record on the file contains the subscriber's name, subscriber's phone number, phone number called, distance from Metroville of the number called (in kilometres) and the duration of the call (in seconds).

Design a program that will read the Tidy Phones charges file and produce a telephone charges report, as follows.

TELEPHONE CHARGES

SUBSCRIBER NAME	SUBSCRIBER NUMBER	PHONE NUMBER CALLED	COST OF CALL
xxxx	xxxxxxxxxx	xxx-xxxx	999.99
xxxx	xxxxxxxxxx	xxx-xxxx	999.99
		TOTAL REVENUE	9999.99

The cost of each call is calculated as follows:

Distance from Metroville	Cost (\$)/minute
less than 25 km	0.35
25 ≤ km < 75	0.65
75 ≤ km < 300	1.00
300 ≤ km ≤ 1000	2.00
greater than 1000 km	3.00

Main headings and column headings are to appear as printed on the report, with an allowance of 45 detail lines per page. The total revenue line is to print three lines after the last detail line.

- 9 Design an algorithm that will produce a payroll register from an employee file. Each input record contains the employee number, the hours worked that week and the rate of pay.

Your program is to read the employee file, and for each employee number, look up a table of employee numbers and names to retrieve the employee's name. The table contains about 100 entries, and is in sequence of employee number with the number 9999 used as a sentinel to mark the end of the table. If the employee number cannot be found in the table, a message is to print on the report and no more processing is to be performed for that record. The gross pay for each employee is also to be calculated as hours worked times rate of pay.

Print a detail line for each employee, listing the employee's number, employee's name, hours worked, rate of pay and gross pay. Print headings and column headings at the top of each page, allowing for 45 detail lines per page, and at the end of the report, the total number of employees and total gross pay.

- 10 The Mitre-11 hardware outlets require an inventory control program that is to accept order details for an item, and generate a shipping list and a back order list.

Design an interactive program that will conduct a dialogue on the screen for the input values, and print two reports as required. The screen dialogue is to appear as follows:

ENTER Item No. 99999
 Quantity on hand 999
 Order quantity 999
 Order number 999999

If an item number does not have precisely five digits, an error message is to appear on the screen. If the quantity on hand is sufficient to meet the order (order quantity \leq quantity on hand), one line is to be printed on the shipping list. If the quantity on hand is insufficient to meet the order (order quantity $>$ quantity on hand), the order is to be filled partially by whatever stock is available. For this situation, one line should appear on the shipping list with the appropriate number of units shipped (quantity on hand) and a message 'Order partially filled'. An entry for the balance of the order (order quantity - quantity on hand) is to be printed on the back order list.

If the quantity on hand is zero, the message 'Out of stock' is to appear on the shipping list, and an entry for the full order quantity is to be printed on the back order list.

Your program is to continue to process inventory orders until a value of zero is entered for the item number.

Report layouts for the shipping list and back order list are as follows:

MITRE-11 HARDWARE

PAGE xx

INVENTORY CONTROL SHIPPING LIST

ORDER NO.	ITEM NO.	UNITS SHIPPED	MESSAGE
999999	99999	999	—
999999	99999	999	—

MITRE-11 HARDWARE

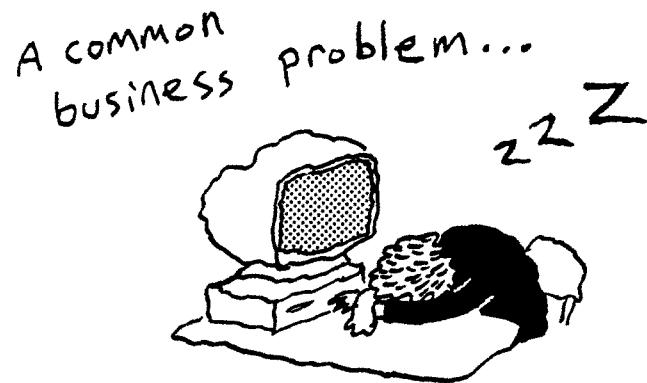
PAGE xx

INVENTORY CONTROL BACK ORDER LIST

ORDER NO.	ITEM NO.	BACK ORDER QTY
999999	99999	999
999999	99999	999

Chapter 10

General algorithms for common business problems



Objectives

- To provide general pseudocode algorithms to four common business applications. Topics covered are:
 - report generation with page break
 - single-level control break
 - multiple-level control break
 - sequential file update

Outline

- 10.1 Program structure
 - 10.2 Report generation with page break
 - 10.3 Single-level control break
 - 10.4 Multiple-level control break
 - 10.5 Sequential file update
- Chapter summary
Programming problems

10.1 Program structure

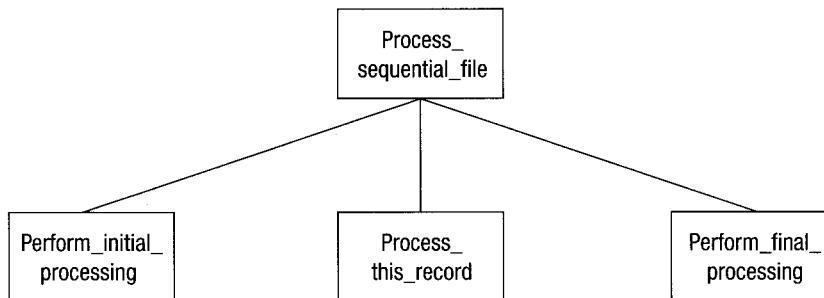
The aim of this chapter is to present a number of general pseudocode solutions for typical programming problems. All of the features covered in the previous nine chapters have been incorporated into these solutions, with the result that each solution offers a sound modular structure with highly cohesive modules.

Throughout this book, reference has been made to a general solution algorithm for the processing of sequential files. This algorithm is a skeleton solution, and in pseudocode looks like this:

```
Process_sequential_file
    Initial processing
    Read first record
    DOWHILE more records exist
        Process this record
        Read next record
    ENDDO
    Final processing
END
```

This basic solution algorithm forms the framework for most commercial business programs. It does not include processing for page headings, control breaks, total lines or special calculations. However, you can easily incorporate these requirements by expanding this general solution.

This general solution algorithm can also be modularised:



The mainline module would now look like this:

```
Process_sequential_file
    Perform_initial_processing
    Read first record
    DOWHILE more records exist
        Process_this_record
        Read next record
    ENDDO
    Perform_final_processing
END
```

The module Process_this_record can be extended, as required, for the processing of a particular programming problem. We will use this basic program structure to develop solution algorithms to four common business programming applications.

10.2

Report generation with page break

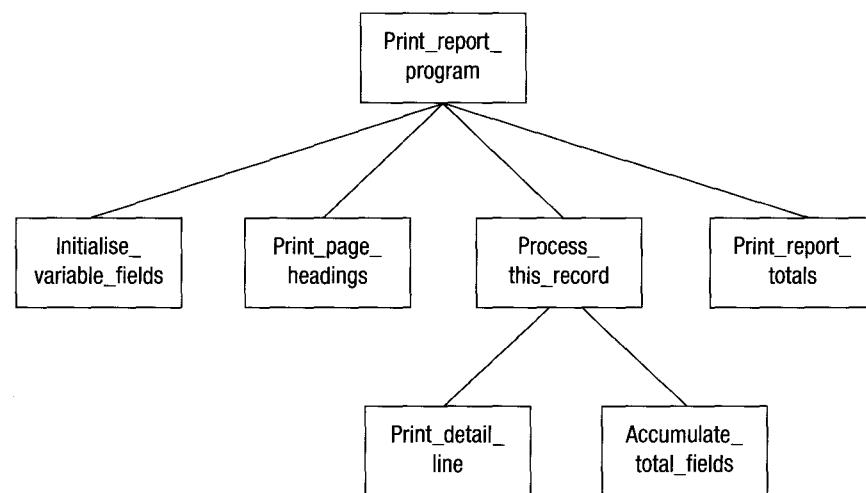
Most reports require page heading lines, column heading lines, detail lines and total lines. Reports are also required to skip to a new page after a pre-determined number of detail lines have been printed.

A typical report might look like this:

GLAD RAGS CLOTHING COMPANY		
12/5/2003	CURRENT ACCOUNT BALANCES	PAGE: 1
CUSTOMER NUMBER	CUSTOMER NAME	ACCOUNT BALANCE
12345	Sporty's Boutique	The Mall, Redfern \$300.50
12346	Slinky's Nightwear	245 Picnic Road, Pymble \$400.50
		Total customers on file 200
		Total customers with balance owing 150
		Total balance owing \$4300.00

Our general solution algorithm for processing a sequential file can be extended by the addition of new modules that cater for these report requirements, as follows:

A Hierarchy chart



Once the hierarchy chart has been established, the solution algorithm can be developed in pseudocode.

B Solution algorithm

Mainline

```
Print_report_program
    Initialise_variable_fields
    Print_page_headings
    Read first record
    DOWHILE more records exist
        IF linecount > max_detail_lines THEN
            Print_page_headings
        ENDIF
        Process_this_record
        Read next record
    ENDDO
    Print_report_totals
END
```

Subordinate modules

- 1 Initialise_variable_fields
set accumulators to zero
set pagecount to zero
set linecount to zero
set max_detail_lines to required value
END
- 2 Print_page_headings
increment pagecount
Print main heading lines
Print column heading lines
Print blank line (if required)
set linecount to zero
END
- 3 Process_this_record
Perform necessary calculations (if any)
Print_detail_line
Accumulate_total_fields
END
- 4 Print_detail_line
Print detail line
increment linecount
END

- 5 Accumulate_total_fields
increment accumulators as required
END
- 6 Print_report_totals
Print total line(s)
END

This general pseudocode solution can now be used as a framework for any report program that requires page breaks.

10.3 Single-level control break

Printed reports that also produce control break total lines are very common in business applications. A control break total line is a summary line for a group of records that contain the same record key. This record key is a designated field on each record, and is referred to as the control field. The control field is used to identify a record or a group of records within a file. A control break occurs each time there is a change in value of the control field. Thus, control break total lines are printed each time a control break is detected.

Here is a single-level control break report.

MULTI-DISK COMPUTER COMPANY						PAGE: 1
12/05/2003		SALES REPORT BY SALESPERSON				
SALESPERSON NUMBER	SALESPERSON NAME	PRODUCT NUMBER	QTY SOLD	PRICE	EXTENSION AMOUNT	
1001	Mary Smith	1032	2	\$10.00	\$20.00	
		1033	2	\$20.00	\$40.00	
		1044	2	\$30.00	\$60.00	
				Sales total for Mary Smith	\$120.00	
1002	Jane Brown	1032	2	\$10.00	\$20.00	
		1045	1	\$35.00	\$35.00	
				Sales total for Jane Brown	\$55.00	
				Report sales total	\$175.00	

Note that a control break total line is printed each time the salesperson number changes.

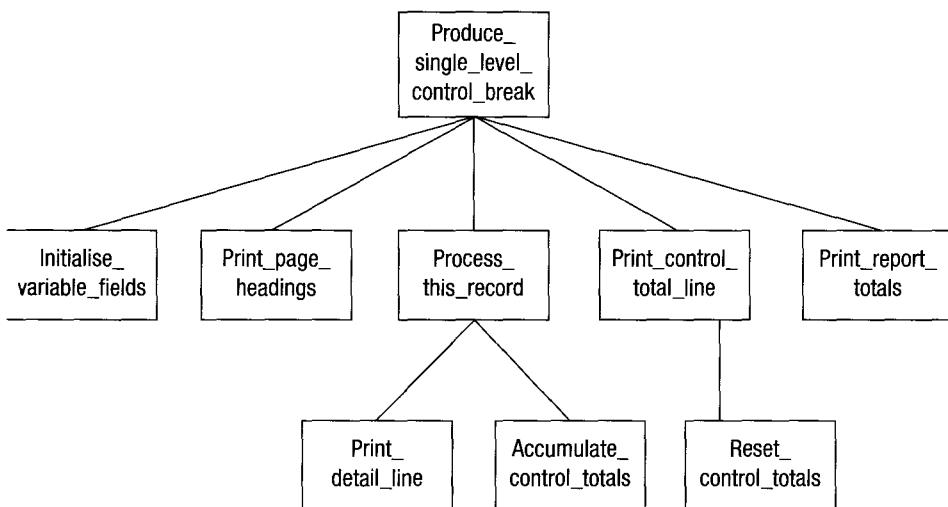
There are two things you must consider when designing a control break program:

- 1 The file to be processed must have been sorted into control field sequence. (In the example above, the file was sorted into ascending sequence of salesperson number.) If the file has not been sorted, erroneous results will occur.

- 2 Each time a record is read from the input file, the control field on the current record must be compared with the control field on the previous record. If the control fields are different, a control break total line must be printed for the previous set of records, *before* the current record is processed.

The general solution algorithm, which was developed for a report generation program, can be extended by the addition of two new modules to incorporate a single-level control break. These modules are named Print_control_total_line and Reset_control_totals.

A Hierarchy chart



All control break report programs will require the following variables:

- 1 A variable named this_control_field which will hold the control field of the record just read.
- 2 A variable named prev_control_field which will hold the control field of the previous record. (To cater for the first record, the statements after the first Read statement will set the new control field to both the variables this_control_field and prev_control_field.)
- 3 One or more variables to accumulate the control break totals.
- 4 One or more variables to accumulate the report totals.

B Solution algorithm

Mainline

```
Produce_single_level_control_break
Initialise_variable_fields
Print_page_headings
Read first record
this_control_field = control field
prev_control_field = control field
DOWHILE more records exist
    IF this_control_field NOT = prev_control_field THEN
        Print_control_total_line
        prev_control_field = this_control_field
    ENDIF
    IF linecount > max_detail_lines THEN
        Print_page_headings
    ENDIF
    Process_this_record
    Read next record
    this_control_field = control field
ENDDO
Print_control_total_line
Print_report_totals
END
```

There are four points in this mainline algorithm that are essential for a control break program to function correctly:

- 1 Each time a new record is read from the file, the new control field is assigned to the variable this_control_field.
- 2 When the first record is read, the new control field is assigned to both the variables this_control_field and prev_control_field. This will prevent the control totals printing before the first record has been processed.
- 3 The variable prev_control_field is updated as soon as a change in the control field is detected.
- 4 After the end of the file has been detected, the module Print_control_total_line will print the control break totals for the last record or set of records.

Subordinate modules

- 1 Initialise_variable_fields
 set control total accumulators to zero
 set report total accumulators to zero
 set pagecount to zero
 set linecount to zero
 set max_detail_lines to required value

END

- 2 Print_page_headings
 - increment pagecount
 - Print main heading lines
 - Print column heading lines
 - Print blank line (if required)
 - set linecount to zeroEND
- 3 Process_this_record
 - Perform necessary calculations (if any)
 - Print_detail_line
 - Accumulate_control_totalsEND
- 4 Print_control_total_line
 - Print control total line
 - Print blank line (if required)
 - increment linecount
 - Reset_control_totalsEND
- 5 Print_report_totals
 - Print report total lineEND
- 6 Print_detail_line
 - Print detail line
 - increment linecountEND
- 7 Accumulate_control_totals
 - increment control total accumulatorsEND
- 8 Reset_control_totals
 - add control total accumulators to report total accumulators
 - set control total accumulators to zeroEND

Notice that when a control total line is printed, the module Reset_control_totals is called. This module will add the control totals to the report totals and reset the control totals to zero for the next set of records. This general solution algorithm can now be used as a framework for any single-level control break program.

10.4

Multiple-level control break

Often reports are required to produce multiple-level control break totals. For instance, the sales report produced in Section 10.3 may require sales totals for each salesperson in the company, as well as sales totals for each department within the company.

The monthly sales report might then look like this:

MULTI-DISK COMPUTER COMPANY						
12/05/03		SALES REPORT BY SALESPERSON				PAGE 1
DEPT	SALESPERSON NUMBER	SALESPERSON NAME	PRODUCT NUMBER	QTY SOLD	PRICE	EXTENSION AMOUNT
01	1001	Mary Smith	1032	2	\$10.00	\$20.00
			1033	2	\$20.00	\$40.00
			1044	2	\$30.00	\$60.00
			Sales total for Mary Smith			\$120.00
	1002	Jane Brown	1032	2	\$10.00	\$20.00
			1045	1	\$35.00	\$35.00
			Sales total for Jane Brown			\$55.00
			Sales total for Dept 01			\$175.00
02	1050	Jenny Ponds	1033	2	20.00	\$40.00
			1044	2	30.00	\$60.00
			Sales total for Jenny Ponds			\$100.00
			Sales total for Dept 02			\$100.00
			Report sales total			\$275.00

Note that a control break total line is printed each time the salesperson number changes and each time the department number changes. Thus, there are two control fields in this file: salesperson number and department number.

The concepts that applied to a single-level control break program also apply to a multiple-level control break program:

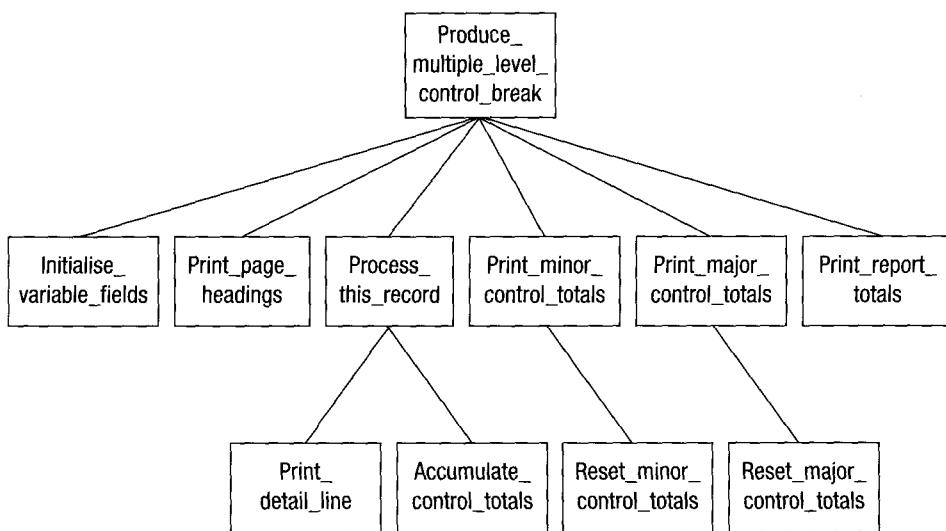
- 1 The input file must be sorted into control field sequence. When there is more than one control field, the file must be sorted into a sequence of minor control field within major control field. (To produce the sales report, the sales file must have been sorted into salesperson number within department number.)

- 2 Each time a record is read from the file, the control field on the current record must be compared with the control field of the previous record. If the minor control field has changed, the control totals for the previous minor control field must be printed. If the major control field has changed, the control totals for the previous minor control field *and* major control field must be printed.

The general solution algorithm that was developed for a single-level control break program can be extended by the addition of two new modules to incorporate a two-level control break. If three control breaks were required, another two modules would be added to the solution algorithm, and so on.

The names of the modules that produce the control totals have been changed slightly, so that they indicate which level of control break has occurred. These new module names are: Print_minor_control_totals, Print_major_control_totals, Reset_minor_control_totals and Reset_major_control_totals.

A Hierarchy chart



B Solution algorithm

Mainline

```
Produce_multiple_level_control_break
    Initialise_variable_fields
    Print_page_headings
    Read first record
        this_minor_control_field = minor control field
        prev_minor_control_field = minor control field
        this_major_control_field = major control field
        prev_major_control_field = major control field
    DOWHILE more records exist
        IF this_major_control_field NOT = prev_major_control_field THEN
            Print_minor_control_totals
            prev_minor_control_field = this_minor_control_field
            Print_major_control_totals
            prev_major_control_field = this_major_control_field
        ELSE
            IF this_minor_control_field NOT = prev_minor_control_field THEN
                Print_minor_control_totals
                prev_minor_control_field = this_minor_control_field
            ENDIF
        ENDIF
        IF linecount > max_detail_lines THEN
            Print_page_headings
        ENDIF
        Process_this_record
        Read next record
        this_minor_control_field = minor control field
        this_major_control_field = major control field
    ENDDO
    Print_minor_control_totals
    Print_major_control_totals
    Print_report_totals
END
```

The points to be noted in this mainline are:

- 1 Each time a new record is read from the input file, the new control fields are assigned to the variables this_minor_control_field and this_major_control_field.
- 2 When the first record is read, the new control fields are assigned to both the current and previous control field variables. This will prevent control totals printing before the first record has been processed.
- 3 After the end of the input file has been detected, the two modules Print_minor_control_totals and Print_major_control_totals will print control totals for the last minor control field record, or set of records, and the last major control field set of records.

Subordinate modules

- 1 Initialise_variable_fields
 - set minor control total accumulators to zero
 - set major control total accumulators to zero
 - set report total accumulators to zero
 - set pagecount to zero
 - set linecount to zero
 - set max_detail_lines to required valueEND
- 2 Print_page_headings
 - increment pagecount
 - Print main heading lines
 - Print column heading lines
 - Print blank line (if required)
 - set linecount to zeroEND
- 3 Process_this_record
 - Perform necessary calculations (if any)
 - Print_detail_line
 - Accumulate_control_totalsEND
- 4 Print_minor_control_totals
 - Print minor control total line
 - Print blank line (if required)
 - increment linecount
 - Reset_minor_control_totalsEND
- 5 Print_major_control_totals
 - Print major control total line
 - Print blank line (if required)
 - increment linecount
 - Reset_major_control_totalsEND
- 6 Print_report_totals
 - Print report total lineEND
- 7 Print_detail_line
 - Print detail line
 - increment linecountEND

- 8 Accumulate_control_totals
 increment minor control total accumulators
 END
- 9 Reset_minor_control_totals
 add minor control total accumulators to major control total accumulators
 set minor control total accumulators to zero
 END
- 10 Reset_major_control_totals
 add major control total accumulators to report total accumulators
 set major control total accumulators to zero
 END

Because the solution algorithm has simple design and good modular structure, the processing of intermediate control field breaks as well as major and minor control field breaks can be handled easily. The solution algorithm would simply require the addition of two new modules: Print_intermed_control_totals and Reset_intermed_control_totals. The IF statement in the mainline would then be expanded to include this extra condition, as follows:

```

IF this_major_control_field NOT = prev_major_control_field THEN
    Print_minor_control_totals
    prev_minor_control_field = this_minor_control_field
    Print_intermed_control_totals
    prev_intermed_control_field = this_intermed_control_field
    Print_major_control_totals
    prev_major_control_field = this_major_control_field
ELSE
    IF this_intermed_control_field NOT = prev_intermed_control_field THEN
        Print_minor_control_totals
        prev_minor_control_field = this_minor_control_field
        Print_intermed_control_totals
        prev_intermed_control_field = this_intermed_control_field
    ELSE
        IF this_minor_control_field NOT = prev_minor_control_field THEN
            Print_minor_control_totals
            prev_minor_control_field = this_minor_control_field
        ENDIF
    ENDIF
ENDIF

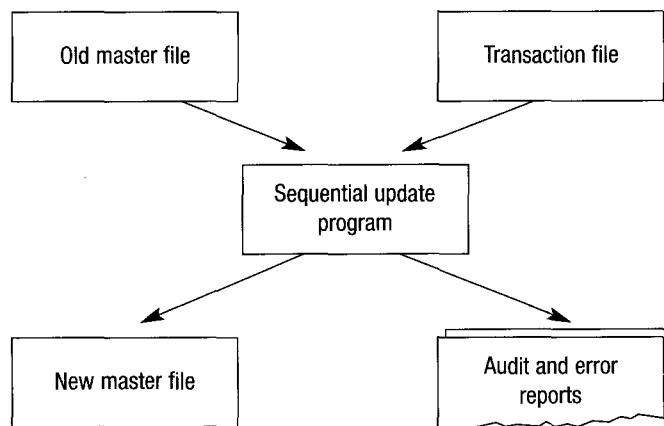
```

This pseudocode algorithm can now be used to process any multiple-level control break program.

10.5 Sequential file update

Most current file transaction update systems are real-time systems; however, for batch processing applications, sequential file updating is very common. It involves updating a master file by applying update transactions on a transaction file. Both files are sequential. A new master file that incorporates the update transactions is produced. Usually, audit reports and error reports are also printed.

A system flowchart of a sequential update program would look like this:



System concepts

1 Master file

A master file is a file that contains permanent and semi-permanent information about the data entities it contains. The records on the master file are in sequence, according to a key field (or fields) on each record. For example, a customer master file may contain the customer's number, name, address, phone number, credit rating and current balance, and may be in sequence of customer number. In this case, customer number would be the key.

2 Transaction file

A transaction file contains all the data and activities that are included on the master file. If the transaction file has been designed specifically to update a master file, there are usually three types of update transactions on this file. These are transactions to:

- add a new record
- update or change an existing record
- delete an existing record.

For example, a customer transaction file might contain transactions that are intended to add a new customer record, change some data on an existing customer record, or delete a customer record on the customer master file. The transaction file is also in sequence according to the same key field as the master record.

3 Audit report

An audit report is a detailed list of all the transactions that were applied to the master file. It provides an accounting trail of the update activities that take place, and is used for control purposes.

4 Error report

An error report is a detailed list of errors that occurred during the processing of the update. Typical errors might be the attempted update of a record that is not on the master file, or the addition of a record that already exists. This error report will require some action to confirm and correct the identified errors.

Sequential update logic

The logic of a sequential update program is more difficult than the other problems encountered, because there are two sequential input files. Processing involves reading a record from each of the input files and comparing the keys of the two records. As a result of this comparison, processing falls generally into three categories:

- 1** If the key on the transaction record is less than the key on the old master record, the transaction is probably an add transaction. The details on the transaction record should be put into master record format, and the record should be written to the new master file. Another record should then be read from the transaction file.
- 2** If the key on the transaction record is equal to the key on the old master record, the transaction is probably an update or delete transaction. If the transaction is an update, the master record should be amended to reflect the required changes. If the transaction is a delete, the master record should not be written to the new master file. Another transaction record should then be read from the transaction file.
- 3** If the key on the transaction record is greater than the key on the old master record, there is no matching transaction for that master record. In this case the old master record should be written unchanged to the new master file and another record read from the old master file.

Sequential update programs also need to include logic that will handle multiple transaction records for the same master record, and the possibility of transaction records that are in error. The types of transaction record errors that can occur are:

- an attempt to add a new master record when a record with that key already exists on the master file
- an attempt to update a master record when there is no record with that key on the master file
- an attempt to delete a master record when there is no record with that key on the master file
- an attempt to delete a master record when the current balance is not equal to zero.

Balance line algorithm

The logic of the sequential update program has fascinated programmers for many years. Authors have offered many solutions to the problem, but none of these has been a truly general solution. Most solutions have been designed around a specific programming language.

A good general solution algorithm written in pseudocode was presented by Barry Dwyer in a paper entitled 'One More Time — How to Update a Master File'.¹ This algorithm has been referred to as the balance line algorithm. It handles multiple transaction records for the one master record, as well as the possibility of transaction record errors.

A modularised version of the balance line algorithm is presented in this chapter. It introduces the concept of a current record. The current record is the record that is currently being processed, ready for updating and writing to the new master file. The current record is established when the record keys on the two files are compared. The current record will be the record that has the smaller record key. Its format will be that of a new master record.

Thus, if the key on the transaction record is less than the key on the old master record, the current record will be made up of the fields on the transaction record. If the key on the transaction record is equal to or greater than the key on the old master record, the old master record will become the current record.

The current record will remain the current record until there are no more transactions to be applied to that record. It will then be written to the new master file, and a new current record will be established.

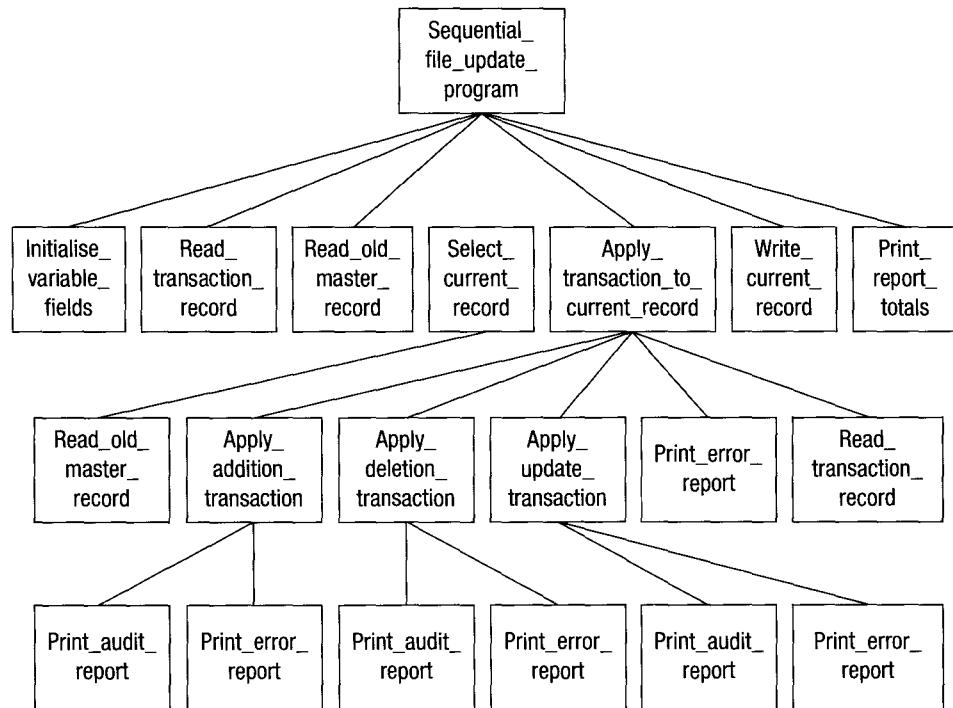
Another variable, `current_record_status`, is used as a program flag to indicate whether or not the current record is available for processing. If the `current_record_status` is active, the current record has been established and is available for updating or writing out to the new master file. If the `current_record_status` is inactive, the current record is not available to be updated or written to the new master file – for example, the current record may have been marked for deletion.

The processing of the two files will continue until `end_of_job` has been reached. `End_of_job` will occur when both the input files have no more data to be processed. Since it is not known which file will reach end of file first, the record key of each file will be set to a high value when EOF is reached. When the record key of one of the files is high, the other file will continue to be processed, as required, until the record key on that file is assigned the same high value. `End_of_job` occurs when the record keys on both the files is the same high value.

Let us now establish a general solution algorithm for a sequential update program. The logic provided will also include the printing of the audit and error reports.

¹ Barry Dwyer, 'One More Time – How to Update a Master File', *Comm. ACM*, Vol. 124, No. 1, January 1981.

A Hierarchy chart



B Solution algorithm

Mainline

```
Sequential_file_update_program
  Initialise_variable_fields
  Read_transaction_record
  Read_old_master_record
  set current_record_status to 'inactive'
  DOWHILE NOT end_of_job
    Select_current_record
    DOWHILE transaction_record_key = current_record_key
      Apply_transaction_to_current_record
    ENDDO
    IF current_record_status = 'active' THEN
      Write_current_record
      set current_record_status to 'inactive'
    ENDIF
  ENDDO
  Print_report_totals
END
```

Subordinate modules

- 1 Initialise_variable_fields
 - set total_transaction_records to zero
 - set total_old_master_records to zero
 - set total_new_master_records to zero
 - set total_error_records to zero
 - set end_of_job to falseEND
- 2 Read_transaction_record
 - Read transaction record
 - IF NOT EOF THEN
 - increment total_transaction_records
 - ELSE
 - set transaction_record_key to high value
 - IF old_master_record_key = high value THEN
 - set end_of_job to true
 - ENDIF
 - ENDIFEND
- 3 Read_old_master_record
 - Read old master record
 - IF NOT EOF THEN
 - increment total_old_master_records
 - ELSE
 - set old_master_record_key to high value
 - IF transaction_record_key = high value THEN
 - set end_of_job = true
 - ENDIF
 - ENDIFEND
- 4 Select_current_record
 - IF transaction_record_key < old_master_record_key THEN
 - set up current record with transaction record fields
 - ELSE
 - set up current record with old master record fields
 - set current_record_status to 'active'
 - Read_old_master_record
 - ENDIFEND

```
5    Apply_transaction_to_current_record
      CASE OF transaction_type
        addition : Apply_addition_transaction
        deletion : Apply_deletion_transaction
        update : Apply_update_transaction
        other :   error_message = 'invalid transaction type'
          Print_error_report
      ENDCASE
      Read_transaction_record
    END

6    Write_current_record
      Write current record to new master file
      increment total_new_master_records
    END

7    Print_report_totals
      Print total_transaction_records
      Print total_old_master_records
      Print total_new_master_records
      Print total_error_records
    END

8    Apply_addition_transaction
      IF current_record_status = 'inactive' THEN
        set current_record_status to 'active'
        Print_audit_report
      ELSE
        error_message = 'Invalid addition; record already exists'
        Print_error_report
      ENDIF
    END

9    Apply_deletion_transaction
      IF current_record_status = 'active' THEN
        set current_record_status to 'inactive'
        Print_audit_report
      ELSE
        error_message = 'Invalid deletion, record not on master file'
        Print_error_report
      ENDIF
    END
```

```

10  Apply_update_transaction
    IF current_record_status = 'active' THEN
        apply required change(s) to current record
        Print_audit_report
    ELSE
        error_message = 'Invalid update, record not on master file'
        Print_error_report
    ENDIF
END

11  Print_audit_report
    Print transaction details on audit report
    CASE OF transaction_type
        addition : print 'record added'
        deletion : print 'record deleted'
        update : print 'record updated'
    ENDCASE
END

12  Print_error_report
    Print transaction details on error report
    Print error_message
    increment total_error_records
END

```

Note that end_of_job is reached when the transaction_record_key = high value AND the old_master_record_key = high value. This pseudocode algorithm can now be used to process any sequential file update program.

Chapter summary

The aim of this chapter was to develop general pseudocode algorithms to four common business applications. The applications covered were:

- report generation with page break
- single-level control break
- multiple-level control break
- sequential file update.

In each section, the application was discussed, a hierarchy chart was developed, and a general solution algorithm was presented in pseudocode. These solution algorithms can be used when writing programs that incorporate any of the above applications.

Programming problems

Using the sample solution algorithms provided in this chapter, design a solution algorithm for the following programming problems. Your solution should contain:

- a defining diagram
- a hierarchy chart
- a pseudocode algorithm
- a desk check of the solution.

- 1 Design an algorithm to produce a list of customers from the Glad Rags Clothing Company's customer master file. Each record on the customer master file contains the customer's number, name, address (street, city, state and postcode) and account balance.

Your program is to read the customer master file and print a report of all customers whose account balance is greater than zero. Each detail line is to contain the customer's number, name, address and account balance. Print headings and column headings at the top of each page, allowing for 35 detail lines per page, and at the end of the report, the total customers on file, the total customers with balance owing, and the total balance owing, as follows:

GLAD RAGS CLOTHING COMPANY			
xx/xx/xx	CURRENT ACCOUNT BALANCES		PAGE: XX
CUSTOMER NUMBER	CUSTOMER NAME	ADDRESS	ACCOUNT BALANCE
xxxxx	xxxxxxxxxx	xxxxxxxxxxxxxxxxxxxx	9999.99
xxxxx	xxxxxxxxxx	xxxxxxxxxxxxxxxxxxxx	9999.99
:	:	:	:
		Total customers on file	999
		Total customers with balance owing	999
		Total balance owing	99999.99

- 2 Design an algorithm to produce a sales commission report from a company's sales file. Each record on the sales file contains the salesperson's number, name and sales amount.

Your program is to read the sales file, calculate the sales commission according to the following table, and print a sales commission report.

Sales range	Commission rate
\$0-\$499.99	No commission
\$500.00-\$749.99	2%
\$750.00 and above	3%

Each detail line is to contain the salesperson's number, sales amount, commission rate and total commission. Print headings and column headings at the top of each page, allowing for 35 detail lines per page, and at the end of the report, the total commission, as follows:

SALES COMMISSIONS					PAGE: XX
SALESPERSON NUMBER	SALESPERSON NAME	SALES AMOUNT	COMMISSION RATE	COMMISSION	
xxxxx	xxxxxxxxxxxx	9999.99	x%	999.99	
xxxxx	xxxxxxxxxxxx	9999.99	x%	999.99	
TOTAL COMMISSION					9999.99

- 3 Design an algorithm that will create a validation report from a customer sales file. Each field on the sales record is to be validated as follows:

Field	Format
Customer number	Numeric
Customer name	Alphanumeric
Street	Alphanumeric
Town	Alphanumeric
Postcode	Numeric
Phone	Alphanumeric
Fax	Alphanumeric
Balance due	Numeric
Credit limit	Numeric (0-\$1000)

If a field is found to be in error, print a line on the validation report showing the customer number, name, address (street, town, postcode) and an appropriate message, as indicated in the diagram below. There may be multiple messages for the one record. Print headings and column headings at the top of each page, allowing for 45 detail lines per page.

VALIDATION REPORT			
CUSTOMER SALES FILE			PAGE: XX
CUSTOMER NUMBER	CUSTOMER NAME	ADDRESS	MESSAGE
xxxxx	xxxxxxxxxxxx	xxxxxxxxxxxx	Postcode not numeric
xxxxx	xxxxxxxxxxxx	xxxxxxxxxxxx	Credit limit invalid

- 4 The Multi-Disk computer company requires a single-level control break program to produce a sales report by salesperson from their sales file. Design an algorithm that will read the sales file and create the sales report as shown below.

Each record on the sales file contains the salesperson's number, name, the product number of the product sold, the quantity sold and the price of the product. There may be more than one record for each salesperson, depending on the products sold that month. The sales file has been sorted into ascending sequence of salesperson number.

Your program is to read the sales file sequentially, calculate the extension amount (price * quantity sold) for each product sold and print a detail line for each record processed. Control total lines showing the sales total for each salesperson are to be printed on change of salesperson number. Print headings and column headings at the top of each page, allowing for 40 detail lines per page.

MULTI-DISK COMPUTER COMPANY						PAGE:xx
SALES REPORT BY SALESPERSON						xx/xx/xx
SALESPERSON NUMBER	SALESPERSON NAME	PRODUCT NUMBER	QTY SOLD	PRICE	EXTENSION AMOUNT	
xxxx	xxxx xxxxxx	xxxx	99	999.99	9 999.99	
		xxxx	99	999.99	9 999.99	
		xxxx	99	999.99	9 999.99	
Sales total for xxxx xxxxxx						99 999.99
Report sales total						999 999.99

- 5 The same sales file as described in Problem 4 exists, with the addition of a further field, the department number. The sales file has been sorted into ascending sequence of salesperson number within department number. Print the same sales report, with the additional requirement of printing a sales total line on change of department number, as well as on change of salesperson number.

Print the report details as per the following sales report: headings and column headings at the top of each page, allowing for 40 detail lines per page.

MULTI-DISK COMPUTER COMPANY							PAGE:xx
SALES REPORT BY SALESPERSON							xx/xx/xx
DEPT	SALESPERSON NUMBER	SALESPERSON NAME	PRODUCT NUMBER	QTY SOLD	PRICE	EXTENSION AMOUNT	
xx	xxxx	xxxx xxxxxx	xxxx	xx	999.99	9 999.99	
			xxxx	xx	999.99	9 999.99	
			xxxx	xx	999.99	9 999.99	
			Sales total for xxxx xxxxxx		99 999.99		
			Sales total for dept xx		99 999.99		
			Report sales total		999 999.99		

- 6 ABC University requires a single-level control break program to produce a lecturer information report by lecturer from the university's course file. Design an algorithm that will read the course file and create the lecturer information report as shown below.

Each record on the course file contains details of a lecturer's teaching load – that is, the lecturer's number, name, the course number of the course being taught, the credit hours for that course and the class size. There may be more than one record for each lecturer, depending on the number of courses he or she teaches. The course file has been sorted into ascending sequence of lecturer number.

Your program is to read the course file sequentially, calculate the lecturer's contact hours ((class size/50) * credit hours), and produce the lecturer information report. On change of lecturer number, print control total lines showing the total contact hours for each lecturer. Print headings and column headings at the top of each page, allowing for 40 detail lines per page.

ABC UNIVERSITY					
LECTURER INFORMATION REPORT					
LECTURER NUMBER	LECTURER NAME	COURSE NUMBER	CREDIT HOURS	CLASS SIZE	CONTACT HOURS
xxxx	xxxxxxxxxx	xxxxx	x	xxx	xxx
		xxxxx	x	xxx	xxx
		xxxxx	x	xxx	xxx
		Contact hours for lecturer xxxx		x xxx	
		Contact hours for university		xx xxx	

- 7 The same course file as described in Problem 6 exists, with the addition of a further field, the university department number. The course file has been sorted into

ascending sequence of lecturer number within department number. Print the same lecturer information report, with the additional requirement of printing a total contact hours line on change of department number, as well as on change of lecturer number.

Print the report details as per the following lecturer information report. Print headings and column headings at the top of each page, allowing for 40 detail lines per page.

ABC UNIVERSITY
LECTURER INFORMATION REPORT

DEPT NUMBER	LECTURER NUMBER	LECTURER NAME	COURSE NUMBER	CREDIT HOURS	CLASS SIZE	CONTACT HOURS
XXX	XXXX	XXXXXXXXXX	XXXXX	X	XXX	XXX
			XXXXX	X	XXX	XXX
			XXXXX	X	XXX	XXX
Contact hours for lecturer xxxx						X XXX
Contact hours for department xxx						X XXX
Contact hours for university						XX XXX

- 8** The same course file as described in Problem 7 exists, with the addition of a further field, the college number. The course file has been sorted into ascending sequence of lecturer number within department number within college number. The same lecturer information report is to be printed, with the additional requirement of printing a total contact hours line on change of college number, as well as on change of department number and on change of lecturer number.

Print the report details as per the following lecturer information report. Print headings and column headings at the top of each page, allowing for 40 detail lines per page.

ABC UNIVERSITY
LECTURER INFORMATION REPORT

COLLEGE NUMBER	DEPT NUMBER	LECTURER NUMBER	LECTURER NAME	COURSE NUMBER	CREDIT HOURS	CLASS SIZE	CONTACT HOURS
XXXXX	XXX	XXXX	XXXXXXXXXX	XXXXX	X	XXX	XXX
			XXXXXXXXXX	XXXXX	X	XXX	XXX
			XXXXXXXXXX	XXXXX	X	XXX	XXX
Contact hours for lecturer xxxx							XXX
Contact hours for department xxx							X XXX
Contact hours for college xxxx							XX XXX
Contact hours for university							XX XXX

- 9** The XYZ Bank requires a program to sequentially update its savings account master file. A sequential file of update transactions is to be used as the input transaction file, along with the customer master file.

The customer master file contains the customer's account number, and the balance forward amount. The customer transaction update file contains three types of records, as follows:

- i** Deposit records containing a record code of 'D', the account number and the amount of a deposit.
- ii** Withdrawal records containing a record code of 'W', the account number and the amount of a withdrawal.
- iii** Interest records containing a record code of 'I', the account number and the amount of interest earned.

There is a deposit record for each deposit a customer made, a withdrawal record for each withdrawal, and an interest record if interest was credited during the period. The updating process consists of adding each deposit or interest to the balance forward amount on the master record, and subtracting each withdrawal.

Both files have been sorted into account number sequence. There can be multiple update transactions for any one savings account master record and a new savings account master file is to be created.

If a transaction record is in error, the transaction details are to be printed on the transaction error report, with one of the following messages:

- 'invalid deposit, account number not on file'
- 'invalid withdrawal, account number not on file'
- 'invalid interest record, account number not on file'

- 10** The Yummy Chocolates confectionery company requires a program to sequentially update its customer master file. A sequential file of update transactions is to be used as the input transaction file, along with the customer master file.

The customer master file contains the customer number, customer name, customer address (street, city, state and postcode) and account balance. The customer transaction file contains the same fields, as well as a transaction code of 'A' (add), 'D' (delete) and 'U' (update).

Both files have been sorted into customer number sequence. There can be multiple update transactions for any one customer master record and a new customer master file is to be created.

Transaction records are to be processed as follows:

- i** If the transaction record is an 'Add', the transaction is to be written to the new customer master file.
- ii** If the transaction record is a 'Delete', the old master record with the same customer number is not to be written to the new customer master file.
- iii** If the transaction record is an update, the old master record with the same customer number is to be updated as follows:
 - if customer name is present, update customer name

if street is present, update street
if town is present, update town
if state is present, update state
if postcode is present, update postcode
if balance paid is present, subtract balance paid from account balance on old customer master record.

As each transaction is processed, print the transaction details on the customer master audit report, with the message, 'record added', 'record deleted' or 'record updated' as applicable.

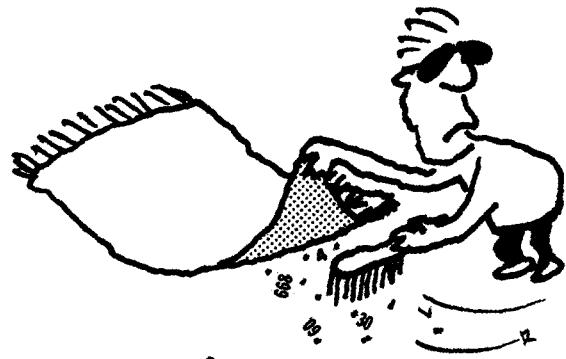
If a transaction record is in error, the transaction details are to be printed on the customer update errors report, with one of the following messages:

'invalid addition, customer already exists'
'invalid deletion, customer not on file'
'invalid update, customer not on file'.

Chapter

11

Detailed object-oriented design



Information hiding

Objectives

- To introduce object-oriented design
- To define objects, classes, attributes, operations and information hiding
- To list the steps required to create an object-oriented design solution
to a problem

Outline

- TT 1 Introduction to object-oriented design
 - TT 2 Steps in creating an object-oriented solution
 - TT 3 Programming example using object-oriented design
 - TT 4 Interface and GUI objects
- Chapter summary
Programming problems

11.1

Introduction to object-oriented design

In this book, program design has concentrated on what a program does. In general, data has been considered only when it is required for input, output or calculations in algorithms. This is said to be a *procedural* approach to program design because it involves identifying and organising the *processes* in the problem solution.

Procedural design has been found to have some limitations in the development of large systems or networked and multi-user systems. Large procedural systems, even if they are structured and modular, can still become extremely complicated and difficult to understand. Developers are not always aware of the work being performed on the system by other members of the development team. As a result, the same blocks of code are sometimes duplicated in different parts of the system, when ideally they should only occur once, and remain available to be reused. It can become difficult to trace and use efficiently the data being used by different processes in large procedural systems. Object-oriented technology not only addresses these difficulties in large system development, it also provides flexibility and economy to the programming of multi-user and networked systems.

Object-oriented design asks that we interact with the problem in much the same way that we interact with our world – we treat it as a set of separate objects that perform actions and relate to each other. An object-oriented program is a set of interacting *objects*, each one responsible for its own activities and data.

Encapsulation and information hiding

Objects are said to *encapsulate* (enclose together in a single indivisible unit, like in a capsule) their data and the processes that act on that data. Their internal processes and data can operate while existing independently from the rest of the system. This means that objects can be used in several places in one system or across several systems, possibly at the same time. The code inside an object can be easily maintained without interfering with the rest of the system. Conversely, changes to other parts of the system will not directly affect the object.

In object-oriented design, each object can be regarded as a ‘black box’ whose internal workings are hidden from all other objects. This principle of information hiding simplifies the use of objects, because the rest of the system does not need to know how they are structured or how they perform their operations. Information hiding is achieved through encapsulation.

The goal of information hiding is to make the object as robust and independent as possible. It ensures that attribute values cannot be accidentally changed by other parts of the system, and carefully controls the interactions that the object has with other objects.

Objects

An object can be considered as a container for a set of data and the operations that need to be performed on it. An object has the following properties:

- It has an *identity* that is unique for the lifetime of the object.
- It has data in the form of a set of characteristics or *attributes*, each of which has a value at any given point in time.
- A set of *operations* or *methods* can be performed on the data.
- It is an instance (example) of a *class*.

Consider a real-world object, such as a car. Each car object has an *identity* in the form of its licence plate and has *attributes* to describe it: such as make, model, number of doors, body length, engine size, colour and speed. Cars also have *operations*, actions that they can perform: such as accelerate, stop, brake and turn.

Classes and objects

An object is created from a template or pattern, called a *class*, which defines the basic relationships, attributes and operations available to the objects of that class. Each class in a system should bear a unique name. When an object is created, it acquires a unique identity and the attributes and operations from its class. The process of creating objects from classes is called *instantiation*, and an object is described as an *instance*, or example, of its class. Many objects can be instantiated from a single class, each object containing the same attributes, but not necessarily storing the same values in those attributes.

Attributes

An object's data is stored in attributes. Attributes are the properties or characteristics that describe that particular object. Objects of the same class will have an identical set of attributes, each with the same name and data type, but the attributes of one object may contain different data values from those of another object. Furthermore, those data values may change at any point in an object's lifetime.

Operations

Objects may receive messages from other objects, asking them to perform services or operations. They may also need to perform services just for themselves. Thus, objects usually have a set of operations, sometimes called methods, procedures or functions, which perform these services. The term 'method' is used to describe the implementation of an operation.

Values stored in the attributes of an object may only be changed by the operations encapsulated inside that object. In other words, the data inside any object will only be affected by that object's own operations.

In pseudocode, the name of the operation should be meaningful and describe the function of the operation, such as calculatePay() or validateMark(). Convention dictates that empty parentheses are included after the operation names to clearly distinguish them from attributes. Parameters passed to or from the operations may be enclosed in the parentheses.

Let's look at the class named Car, represented by the following diagram, which lists the name of the class followed by the attributes of the class and the set of operations that the class can perform.

Car
make
model
doors
bodyLength
engineSize
colour
speed
accelerate()
stop()
brake()
turn(direction)

A Car class

The Ford and Toyota objects in the figure below are instances of the Car class. Many car objects can be instantiated from the single Car class. Each car object will be able to employ all the Car class operations, but the values in their attributes may be different, and can change.

RVJ635 : Car	SVU478 : Car
make = 'Ford' model = 'Falcon' doors = 4 bodyLength = 300 engineSize = 6 colour = 'blue' speed = 0	make = 'Toyota' model = 'Corolla' doors = 5 bodyLength = 200 engineSize = 4 colour = 'red' speed = 60
accelerate() stop() brake() turn(direction)	accelerate() stop() brake() turn(direction)

Car objects

As you can see from the examples given, object names, attributes and operations should be assigned meaningful names.

Constructors

The process of instantiating an object from the class template is performed when a special operation, or set of instructions, known as a *constructor* is called or invoked. Constructors assign initial values to a new object's attributes. Constructors usually have the same name as their class. The pseudocode for instantiating a new object is:

Create object-name as new Class-name()

For example,

Create car as new Car()

The keyword *new* refers to the creation of a new object. By convention, class names start with an upper case, or capital letter, such as *Car*, and object names are written in lower case, such as *car*.

Constructors may:

- have parameters that initialise the attributes with specific values – for example, in pseudocode: Create car as new Car(Ford, Falcon, 4, 300, 6, blue, 0), or
- have no parameters, in which case a new object is assigned all the default values for its attributes – for example, in pseudocode, Create car as new Car().

Default constructors should be included in a class as a matter of good practice.

Accessors and mutators

The values in the attributes of an object should be available to all operations in that object, but hidden from external objects. For safety, only special public operations, known as accessors and mutators, should allow external objects to access the values in attributes.

Accessor operations pass attribute values to external objects. By convention, accessor names start with the word *get*, such as *getPayslip()*. An accessor is simply a descriptive name for an operation that accesses a value.

Mutator operations enable external objects to change the values stored in attributes. By convention, mutator names start with the word *set*, such as *setPayRate()*. A mutator is simply a descriptive name for an operation that changes the value of an attribute.

Messages

Objects must be able to communicate and interact with other objects, but how can this be achieved if their data and operations are encapsulated, and therefore hidden from the external world? Communication is achieved when one object passes a *message* to another. In most cases, a message is a call made by one object to an operation in another object. The receiving object takes responsibility for performing the services defined by that operation. In other

words, a message from one object to another usually initiates the processing of an operation in the receiving object. For example, a message to a bank account object, called account, requesting the display of an account balance, may take the form of the call:

```
account.displayBalance()
```

This call is written in pseudocode as the object name followed by the name of the operation, separated by a period or 'dot operator'.

Note that the account balance attribute itself would be hidden from an external object. The only way an external object could access the account balance would be to send a message to the account object asking for the balance.

In order to produce the required services, the operation may need to receive information from the calling object. In some cases, the operation may return a value to the caller. In pseudocode notation, these inputs and outputs are received and sent as parameters, enclosed in parentheses after the name of the operation, with the sent values first in the parameter list and the returned values last. For example, a message to a cheque account object requesting deposit services would take the form of the call:

```
chequeAccount.deposit(amount, newBalance)
```

Visibility

Operations that perform services for other objects must be visible to those other objects, whereas other operations are hidden. How does an external object distinguish between visible and invisible operations?

By declaring an object's operations as *public* or *private*, we are describing their visibility to the system. Other forms of visibility are possible, too. The *public* operations of an object are those producing services requested by other objects. Other objects can see the specification (basically its name, parameters and return value) of a public operation, defining *what* it can do and what information it requires to perform the service. However, its implementation, or *how* it is going to perform the service, remains hidden.

Other, *private*, operations are needed to perform the internal actions in an object and cannot be accessed directly from outside the object.

Similarly, the *attributes* of an object may be private, or invisible, to the rest of the system, or they may be public. Public attributes are not desirable, as their use can lead to unintended side effects.

To illustrate the concept of visibility, consider a bank Account object. Some of the operations of the Account object are necessarily public, such as displayBalance(), deposit() and withdraw(). These operations help the account object to interact with external systems such as an ATM.

The public operation, displayBalance(), may need the services of a calculateInterest() operation. Similarly, the public withdraw() operation may need the services of a verifySufficientFunds(amount) operation to check that the account balance is greater than the withdrawal amount. The calculateInterest() and verifySufficientFunds(amount) are not visible to the

user at the ATM. In fact, they are not visible to any objects outside the account object. They are internal operations of each account because external objects do not require them, and therefore they should be declared as private. The balance attribute, like all attributes, should also be private. Consider the problems that might arise if the deposit() or withdraw() operations in one account could change the balance attribute in another account!

The attributes and the public and private operations for the account class can be represented in the following diagram. Public operations are preceded by a + (plus) and private operations are, preceded by a - (minus).

Account
balance
+displayBalance()
+deposit()
+withdraw()
-calculateInterest()
-verifySufficientFunds(amount)

An Account class with public and private operations

11.2 Steps in creating an object-oriented solution

Many different object-oriented analysis and design techniques have been developed. The Unified Modelling Language (UML) is a comprehensive graphical language to support several major techniques covering most of the object-oriented development life cycle. This text concentrates specifically on algorithm design of operations in the detailed design stage of the life cycle, rather than on the entire design stage. Object-oriented design is usually approached from several perspectives:

- Real-world objects are modelled.
- Objects are seen as having responsibilities and providing services.
- Sequences of user interactions are modelled through different possible scenarios.

Object-oriented approaches are suitable for the development of large, interactive systems. For small systems, the degree of abstraction required can overcomplicate problems addressed in introductory programming. This text presents a simplified approach for creating an object-oriented solution to the type of problem encountered in introductory computer programming courses. The static behaviour of classes is considered and the interactive or dynamic behaviour, other than the design of the operations themselves, is left for more advanced texts. UML is referred to when simplified approaches are not used.

The steps in creating an object-oriented solution for a simple problem are:

- 1 Identify the classes, together with their attributes, responsibilities and operations.
- 2 Determine the relationship between the objects of those classes.
- 3 Design the algorithms for the operations, using structured design.
- 4 Develop a test or driver algorithm.

To start your object-oriented design, read the problem definition carefully, looking for nouns and noun phrases. These may translate into the *objects* or their *attributes* in the program. Think about the common characteristics of these potential objects and whether or not they could be grouped into classes. To identify the *operations*, look closely at the verbs that describe the actions performed by the nouns. Think about the responsibilities of the possible objects, and how these might translate into operations, as well as about how each class of objects may relate to other classes of objects. Now think about the solution from a user's perspective. What are the different possible paths a user may take? Revise your list of the objects and their operations and then list them in a class table to create a framework from which to refine a model.

A CLASS TABLE

Class	Attributes	Responsibilities	Operations
object name	attribute 1 attribute 2 ...		operation1() ...

Using the four steps for creating an object-oriented solution to a simple problem, let's look at Example 8.5, which was developed in Chapter 8, using top-down design.

EXAMPLE 11.1 Calculate employee's pay

A program is required by a company to read an employee file containing the employee's number, pay rate and the number of hours worked in a week. The program is then to validate the pay rate and the hours worked fields and, if valid, compute the employee's weekly pay and print it along with the input data on to a payslip.

Validation: According to the company's rules, the maximum hours an employee can work per week is 60 hours, and the maximum hourly rate is \$25.00 per hour. If the hours worked field or the hourly rate field is out of range, the input data and an appropriate message is to be printed and the employee's weekly pay is not to be calculated.

Weekly pay calculation: Weekly pay is calculated as hours worked times pay rate. If more than 35 hours are worked, payment for the overtime hours worked is calculated at time-and-a-half.

Step 1: Identify the classes, together with their attributes, responsibilities and operations

First, just as if you were using a top-down approach to define a problem, underline the nouns and adjectives relevant to the problem. This will begin to reveal the possible classes, the objects that will be derived from these classes, and their associated attributes. With the nouns and adjectives underlined, the problem statement would look like this:

A program is required by a company to read an employee file containing the employee's number, pay rate and the number of hours worked in a week. The program is then to validate the pay rate and the hours worked fields and, if valid, compute the employee's weekly pay and print it along with the input data on to a payslip.

Validation: According to the company's rules, the maximum hours an employee can work per week is 60 hours, and the maximum hourly rate is \$25.00 per hour. If the hours worked field or the hourly rate field is out of range, the input data and an appropriate message is to be printed and the employee's weekly pay is not to be calculated.

Weekly pay calculation: Weekly pay is calculated as hours worked times pay rate. If more than 35 hours are worked, payment for the overtime hours worked is calculated at time-and-a-half.

Start a list of the underlined nouns. Not all of the nouns and adjectives in the problem statement are vital to the solution. For example, you do not need to underline the words *company* or *program*. Nouns that are duplicated, or have the same meaning as a noun that is already in the list, can be omitted.

You can see that the word employee in the first sentence is associated with the descriptive nouns employee number, pay rate and number of hours worked. This indicates that employee could be an object with attributes of employee number, pay rate and hours worked. But do these attributes really define an employee in the real world? In fact, they more closely define the contents of a payslip; so *one* class might be named Payslip.

The input data of pay rate and hours worked will become Payslip attributes. In the real world, this data would probably have been entered from timesheets. Weekly pay is calculated from the input pay rate and hours worked fields. Therefore, it is a 'derived' value and, unless it is required to play an important role, may not need to be an attribute. Weekly pay, as an output, is an important part of the solution, so there is a good case for making it an attribute.

Overtime hours worked and message are not always necessary for the existence of a Payslip object, so they can be omitted from the class definition at this stage.

Having identified the possible classes in the problem statement, consider the program from the user's point of view. Users will only need to start the program and view the printed payslips. In the event of invalid timesheet data, an error message will appear instead of a payslip. The problem statement does not address file-reading problems and we will assume that they will be dealt with elsewhere in the system.

A program about employees, time sheets and payslips models another object, namely a Payroll. The payroll object can be the user interface – the part of the program that a user will see. The user interface will be the program title, possibly a request for the correct data file, printed payslips and any error messages. The payroll object will also interface with other entities outside the program, like the file containing timesheet data. The timesheet data file is a suitable attribute for the Payroll class.

Using meaningful names, copy the classes, objects and attributes that you have identified into a class table:

Class	Attributes	Responsibilities	Operations
Payslip	empNumber payRate hoursWorked weeklyPay		
Payroll	timesheet data file		

Having identified the classes, consider the *responsibilities* of each class. Each Payslip object is responsible for validating and calculating the weekly pay from the timesheet data of one employee. Only valid payslips are printed, and therefore the behaviour of external objects may depend on the validity of each payslip's timesheet data. Each Payslip object is also responsible for communicating the validity of its timesheet data to external objects. Although derived from attribute values, *validity* provides information about the state of the object and persists throughout the object's lifetime. An attribute, validInput, can be added to the Payslip class.

Although each Payslip object deals with just one payslip, multiple Payslip objects may be processed. However, only one Payroll object will be needed. It will be responsible for reading multiple text records from the timesheet data file and for creating many payslips. In other words, the program will have a single interface.

Class	Attributes	Responsibilities	Operations
Payslip	empNumber payRate hoursWorked weeklyPay validInput	Validate data Report validity of data Calculate weekly pay Print a payslip for each valid timesheet	
Payroll	timesheet data file	Read timesheet data	

Now establish the set of operations for each class by underlining the verbs and verb phrases relevant to the problem in the problem statement.

A program is required by a company to read an employee file containing the employee's number, pay rate and the number of hours worked in a week. The program is then to validate the pay rate and the hours worked fields and, if valid, compute the employee's weekly pay and print it along with the input data on to a payslip.

Validation: According to the company's rules, the maximum hours an employee can work per week is 60 hours, and the maximum hourly rate is \$25.00 per hour. If the hours worked field or the hourly rate field is out of range, the input data and an appropriate message is to be printed and the employee's weekly pay is not to be calculated.

Weekly pay calculation: Weekly pay is calculated as hours worked times pay rate. If more than 35 hours are worked, payment for the overtime hours worked is calculated at time-and-a-half.

The Payslip class

Initially, we can assume that the program interface will deal with all input and output and that the Payslip objects will receive information via messages, probably to constructors and mutators. Accessors will pass information about Payslip attribute values to external objects.

Mutators

The words validate the pay rate and the hours worked fields indicate that an operation must validate the data. In object-oriented programs, the data in an object's attributes is closely related to the operations that act on it. Furthermore, each attribute's data should be validated as soon as it is input to prevent the attribute being updated with an incorrect value. Instead of having one validate() operation for the entire Payslip object, *mutators* can validate the data for each timesheet data attribute. Conventional names will be given to these mutators, such as setEmpNumber, setPayRate and setHoursWorked.

The words compute the employee's weekly pay indicate an operation to calculate the employee's weekly pay amount. Although weeklyPay is derived from other attributes, any operation to change its value is regarded as a mutator, so the private mutator setPay() will calculate the weekly pay. If we had decided that weeklyPay should not be an attribute of Payslip, accessible via an accessor, then setPay() may have been a public operation, responsible for conveying the weekly pay value to external objects.

Accessors

The words print the pay and print the input data indicate that the program needs to access the hidden attribute values in the Payslip class. First, to prevent invalid payslips being printed, an accessor, getValidInput(), may be needed to report on the validity of the payslip data.

With regard to the printing of the data, two possible approaches come to mind. In the first approach, accessors pass attribute values to interface objects to be formatted and printed. This approach is appropriate if values from more than one object are being printed – for example, if data from employee name and address objects and employee leave objects are added to the payslip. In the second approach, each Payslip object can take the responsibility for printing itself. With only one class to be printed, we decide to take the second approach, so printPayslip(), a public operation that is not an accessor, will be required. We should still include accessors to ensure flexibility for the Payslip class.

The Payroll class

The Payroll class takes responsibility for interfacing with the user, the input data file and the Payslip objects. The words read an employee file indicate that an operation is required to read the timesheet data file, so readTimesheet() becomes a public operation of the Payroll class. The Payroll object uses the Payslip constructor to create each Payslip object, and it also uses the Payslip operation, printPayslip(), to print the valid payslips. The Payslip constructor will use mutators to validate the values passed to it. Payroll needs another public operation, which we will call run(), to coordinate and control the Payslip objects.

As we define each object's operations, we can also begin to identify the information that the operations may need in order to perform their responsibilities. The information will be shown in parentheses in the same way that parameters are shown for modules. The operation names, preceded by + (plus), if they are public operations, and - (minus), if they are private operations, can now be added to the class table. As the design develops, the visibility of operations may change. We will assume that all attributes are private.

Class	Attributes	Responsibilities	Operations
Payslip	-empNumber	Validate data	+setEmpNumber()
	-payRate	Report validity of data	+setPayRate() +setHoursWorked() +getValidInput(validInput) (accessor)
	-hoursWorked	Calculate weekly pay	-setPay() (mutator)
	-weeklyPay	Print a payslip for each valid timesheet	+printPayslip()
	-validInput		
Payroll	-timesheet data file	Read timesheet data file	+readTimesheet()
			+run()

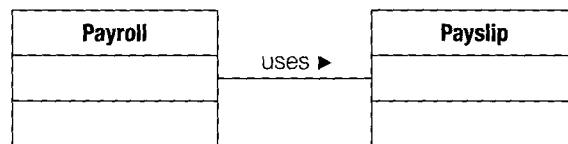
At this point, some important differences between object-oriented and procedural approaches become clear. In the procedural approach, you define what the program has to do; data is independent of the procedures that use it. Data is passed into and between procedures when and where it is required, so it must be carefully scheduled and managed, along with the procedures that use it. This requires the designer and programmer to be familiar with all parts of the system that may access or process data used in the section they are working on.

In the object-oriented approach, designers and programmers building other objects only need to know of an object's existence, services it may be able to provide and the inputs it requires to produce those services. Once an object is instantiated, only its public operations are available for the rest of the program to access as necessary. Its attributes and private operations remain hidden.

In this example, the Payslip object provides the service of calculating and printing the weekly pay. In order to provide those services, it needs timesheet data supplied to it, in the form of employee number, hours worked and pay rate. The Payroll object performs the services of passing the timesheet data file to the Payslip object. It is this 'black box' nature of objects that allows many programmers to work on one large system.

Step 2: *Determine the relationship between the objects of those classes*

The class table developed from the problem statement has two classes. An object from the Payroll class *uses* objects from the Payslip class. The diagram below shows this in a simplified UML notation:



Other relationships between classes are developed and modelled in Chapter 12.

Step 3: *Design the algorithms for the operations, using structured design*

Each operation in the class table requires a step-by-step description of the instructions to produce the required behaviour. Each algorithm should start with the name of the operation and finish with an END statement. It should use correct indentation and follow the rules of structured design.

The data that a Payslip object needs is brought within its scope as soon as it is created. Attribute values are available to all operations within a class, and so empNumber, weeklyPay, validInput, hoursWorked and payRate are visible to each operation.

The first responsibility of the Payslip object is to validate the data passed to its constructor. Mutators can be used to validate the timesheet data, setting the derived attribute, validInput to false if any attribute has an invalid value.

```
setEmpNumber(inEmpNumber)
    empNumber = inEmpNumber
END

setPayRate(inPayRate)
    Set errorMessage to blank
    IF inPayRate > MAX_PAY THEN
        errorMessage = 'Pay rate exceeds $25.00'
        Print empNumber, inPayRate, errorMessage
        validInput = false
    ELSE
        payRate = inPayRate
    ENDIF
END

setHoursWorked(inHoursWorked)
    Set errorMessage to blank
    IF inHoursWorked > MAX_HOURS THEN
        errorMessage = 'Hours worked exceeds 60'
        Print empNumber, inHoursWorked, errorMessage
        validInput = false
    ELSE
        hoursWorked = inHoursWorked
    ENDIF
END

setPay()
    IF hoursWorked <= NORM_HOURS THEN
        weeklyPay = payRate * hoursWorked
    ELSE
        overtimeHours = hoursWorked - NORM_HOURS
        overtimePay = overtimeHours * payRate * 1.5
        weeklyPay = (payRate * NORM_HOURS) + overtimePay
    ENDIF
END
```

The Payslip constructor can use the mutators to initialise its attributes' values. However, if the data is invalid, the value may not be initialised, or it may be initialised with invalid data. Therefore, a default constructor should initialise all attributes.

```
Payslip()
  Set empNumber to 'Unknown'
  Set payRate to 0.0
  Set hoursWorked to 0
  Set validInput to true
  Set weeklyPay to 0.0
END
```

The constructor that accepts input can call the default constructor before setting the attribute values:

```
Payslip (inEmpNumber, inPayRate, inHoursWorked)
Payslip()
setEmpNumber(inEmpNumber)
setPayRate(inPayRate)
setHoursWorked(inHoursWorked)
IF validInput THEN
  setPay()
ENDIF
END
```

External objects need to know very little about payslips apart from their validity and the weekly pay that they calculate. Accessors can convey this information, but are generally too trivial to express in Pseudocode. An accessor to inform other objects about the validity of a payslip would be:

```
getValidInput(validInput)
END
```

The only operation in the Payslip class that is not a constructor, mutator or accessor is the operation to print the payslip:

```
printPayslip()
IF validInput THEN
  Print empNumber, payRate, hoursWorked, weeklyPay
ENDIF
END
```

Step 4: Develop a test or driver algorithm

The final step is to develop another class that has an operation with an algorithm like the mainline in a procedural program. The main driver algorithm ties the objects together and coordinates their activities. This test, or driver class, will provide an interface between Payslip objects and the world, including users. The timesheet data file is part of the interface between the world and Payslip objects. There is only one data file, but there may be multiple Payslip objects. It is therefore sensible to let the driver deal with the mechanics of file processing, leaving the Payslip objects to concentrate on

their specific responsibilities. The timesheet data file will be an attribute of the driver class, and the driver's constructor will open the file.

```
Payroll()
  timesheetFile = 'Timesheet.dat'
END
```

If there are records in the timesheet file, the driver object will create Payslip objects by calling, or invoking, the Payslip constructor with the pseudocode:

```
Create payslip as new Payslip()
```

When more than one object is instantiated from a class, it is important that the program can identify exactly which copy of an operation is being used. A special notation in which the object name is followed by the operation name, separated by a period called the 'dot operator', manages this. For example, the printPayslip() operation owned by a Payslip object is referred to as payslip.printPayslip(). Note that this notation can only be used to access visible or public operations. Because private operations are not publicly accessible they are not used with the dot operator.

We return to structured techniques to design the driver class. The defining diagram is:

Input	Processing	Output
Timesheet data <ul style="list-style-type: none">• empNumber• hoursWorked• payRate	Read timesheet data Create payslips from timesheet data Print valid payslips	Printed payslips Error messages for invalid payslips

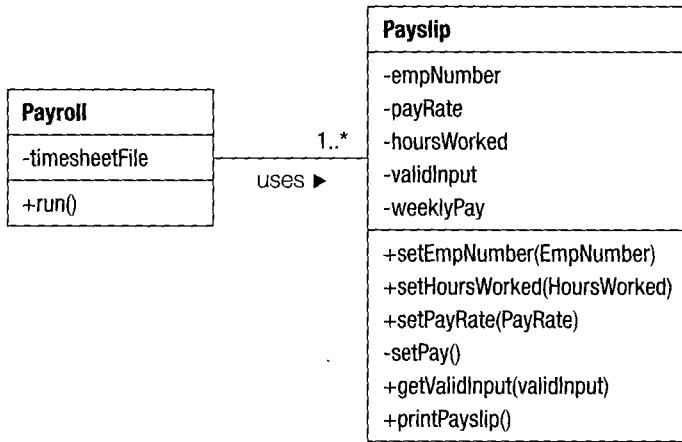
This translates into an algorithm to drive the Payslip class:

```
run()
  Read empNumber, payRate, hoursWorked
  DOWHILE more records
    Create payslip as new Payslip(empNumber, payRate, hoursWorked)
    payslip.printPayslip()
    Read empNumber, payRate, hoursWorked
  ENDDO
END
```

In the class table we defined a readTimesheet() operation and a printPayslip() operation. Each of these operations has only required one line in the algorithm. In the actual program, these lines may need to be expanded into operations.

We now have two classes, a Payslip class and a Payroll class. With the exception of calculation and driver operations, most of the responsibilities of

the two classes have been taken care of by constructors, accessors and mutators. Information has been hidden and protected, encapsulated with these operations within objects. The diagram below shows the classes in simplified UML notation. The accessors and mutators have been included for instructional purposes and are not required in the diagram.



The steps in object-oriented design are not always followed sequentially. As this example shows, sometimes a step can be completely omitted. However, as you have seen in procedural design, the same steps may have to be repeated and refined several times before the design is complete. The driver algorithm may need to be developed at the same time as the other operation algorithms, rather than waiting until all the operation algorithms have been designed.

11.3 Programming example using object-oriented design

EXAMPLE 11.2 Print student results

Design a class to manage student results in a subject. A unique student number identifies each student. During the course of the subject, each student completes three assignments (representing 40% of the final mark but each scored out of 100) and an examination (also scored out of 100). The final mark is calculated by multiplying the sum of the assignments by 0.133 and the examination by 0.6 and adding the two products together. The class will allow a user to update an assignment mark or an examination mark, and to print the final mark along with the student number for each student.

Step 1: Identify the classes, together with their attributes, responsibilities and operations

To commence the design, underline the nouns and noun phrases to identify the objects and their attributes.

Design a class to manage student results in a subject. A unique student number identifies each student. During the course of the subject, each student completes three assignments (representing 40% of the final mark but each scored out of 100) and an examination (also scored out of 100). The final mark is calculated by multiplying the sum of the assignments by 0.133 and the examination by 0.6 and adding the two products together. The class will allow a user to update an assignment mark or an examination mark, and to print the final mark along with the student number for each student.

In the example, it is apparent that a student object will be needed, with attributes of unique student number, three assignments and an examination. There is no need to make the final mark an attribute, because it can be derived from the other attributes.

Now underline the verb and verb phrases to identify the responsibilities and the operations that the object needs to perform.

Design a class to manage student results in a subject. A unique student number identifies each student. During the course of the subject, each student completes three assignments (representing 40% of the final mark but each scored out of 100) and an examination (also scored out of 100). The final mark is calculated by multiplying the sum of the assignments by 0.133 and the examination by 0.6 and adding the two products together. The class will allow a user to update an assignment mark or an examination mark, and to print the final mark along with the student number for each student.

The underlined verbs and verb phrases indicate that there are three public or visible operations – namely, update an assignment mark, update an examination mark and print the final mark, and one private operation, calculate final mark. Two mutators will be used, one to update the assignment mark, called `setAsst()`, and another to update the exam mark, called `setExam()`. The input mark must also be validated before it can be used in the calculations; so another private operation to validate the mark is required. The class table can now be drawn.

Class	Attributes	Responsibilities	Operations
Student	studentNumber asstOne asstTwo asstThree examMark	update assignment mark update exam mark print final mark calculate final mark	+setAsst(asstNum, result) +setExam(result) -validateMark() -calculateFinalMark() +printFinalMark()

Step 2: Determine the relationships between the objects of those classes

There is only one object in the solution, so you can move to the next step.

Step 3: Design the algorithms for the operations, using structured design

Public operations

An algorithm is required for each operation in the object table. The mutator, setAsst(), requires that two parameters be passed to it: the assignment number and the result for the assignment. The result, passed as a parameter, will be validated by the private operation, validateMark(), before updating the mark for that assignment.

```
setAsst(asstNum, result)
    validateMark(result, validInput)
    IF validInput THEN
        CASE OF asstNum
            1: asstOne = result
            2: asstTwo = result
            3: asstThree = result
        OTHERWISE
            Report invalid assignment number error
        ENDCASE
    ENDIF
END
```

Similarly, the mutator setExam() will require the examination result to be passed to it. This result will also be validated by validateMark() before being used.

```
setExam(result)
    validateMark(result, validInput)
    IF validInput THEN
        examMark = result
    ENDIF
END
```

The results will be printed by the operation, printFinalMark(), which calls the private operation calculateFinalMark() to calculate the final mark before printing.

```
printFinalMark()
    calculateFinalMark(finalMark)
    Print studentNumber, finalMark
END
```

Private operations

The assignment and examination results can be tested for range in validateMark() to ensure that no invalid marks update the attribute values. This operation will return a parameter, validInput, which indicates if the input mark is valid.

```
validateMark(result, validInput)
    Set validInput to true
    IF (result < 0 OR result >100) THEN
        Set validInput to false
        Report invalid result error
    ENDIF
END
```

The private operation, calculateFinalMark(), will calculate the final mark and return this value in a parameter.

```
calculateFinalMark(finalMark)
    finalMark = (asstOne + asstTwo + asstThree) * 0.133
    finalMark = finalMark + (examMark * 0.6)
END
```

Step 4: *Develop a test or driver algorithm*

The problem definition stated that a class be designed, rather than a program. This is a common task in object-oriented programming. Rather than develop a mainline algorithm that might drive an entire program, we will write a simple Test class, called testStudent, to allow the trialling of the Student class. This Test class will simply test that all the operations in the student class work correctly, so that it can be used later in the development of the system.

A constructor for the Student class, named Student, also needs to be developed to create the Student object and initialise the studentNumber attribute. The remaining attributes (asstOne, asstTwo, asstThree and examMark) will be set to default values in this constructor.

```
Student(inStudentNumber)
    set studentNumber to inStudentNumber
    set asstOne to 0
    set asstTwo to 0
    set asstThree to 0
    set examMark to 0
END
```

Note that the value of studentNumber is passed as a parameter to the constructor, which will create a Student object with the pseudocode:

```
Create student as new Student(studentNumber)
```

The Student class can now be trialled, using the Test class, testStudent:

```
testStudent()
  Set studentNumber to 111555
  Create student1 as new Student(studentNumber)
  Set asstNum to 2
  Set result to 80
  student1.setAsst(asstNum, result)
  Set studentNumber to 222000
  Create student2 as new Student(studentNumber)
  Set asstNum to 1
  Set result to 95
  student2.setAsst(asstNum, result)
  student1.printFinalMark()
  student2.printFinalMark()
END
```

11.4 Interface and GUI objects

An interface is a device in a program that connects the user's responses to the computer's actions. Many popular programming languages provide a graphical user interface (GUI), which enables the programmer to select the elements of the program's user interface from a pre-existing range of options. These languages are called 'visual' languages, and include Visual Basic, Visual C and Visual J. Java also shares these features.

Interface design is a subset of program design, as it concentrates on one aspect of the program's performance and implementation. Good interfaces make the program easy and comfortable to use and combines elements of psychology, aesthetic design and good programming techniques.

The interfaces are developed from predesigned classes available in the programming language. The user interface options may include windows, buttons, menus, boxes to hold text, drop-down lists and many more. Once they are created, the programmer tailors the interface elements to suit the needs of the program. While some of the programs that provide GUIs are not strictly object oriented in their internal functioning, the interfaces usually are, and object-oriented approaches should be used in their design.

Each user interface element provided in the visual languages is an object with attributes and operations. The size, shape, colour, heading labels and modality of a window or form object on a screen are attributes the values of which are defined by the programmer when the code for the program is written. The way the window behaves in response to events that may come from the user, the program or the system that the program is running on, is defined by the window's operations that the programmer has chosen to use.

EXAMPLE 11.3 Library locator interface

Consider a program that supplies users with the location of a book in a library, based on its call number. The library stores materials from 000 to 250 on the ground level, from 251 to 700 on the first level, and from 701 onwards on the second level. The user will need to provide information to the program about the call number of the book he or she is seeking.

In the algorithm, this menu may appear as a case statement or as a nested decision structure. In a visual language, the inputs for these decision structures can be expressed in at least two possible ways: the user can be asked to enter a choice using text, or to click on one particular object on the screen that represents the option, such as a menu button, an option box or radio buttons.

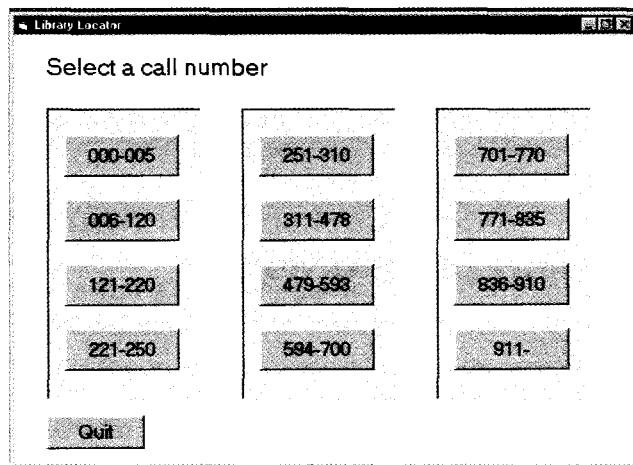
If typed user input is chosen, the program will have to test for invalid inputs and report them with error messages, because the user who types a call number into an input area could potentially make an error. This option requires that the designer prepare algorithms for error trapping and the programmer write the corresponding code. However, a specific location on the correct floor can be provided with fewer objects populating the screen.

In the second approach, there is no invalid input possible. The mainline algorithm will still be a complex decision structure that calls separate modules for each choice, but that decision structure will be presented as a window containing objects that represent the menu choices. The user will select an option that matches his or her requirements and the correct location can be displayed.

To design the user interface, first plan the interface layout. This can be done on paper or on screen, according to the tools that you have available. Next, using this first draft plan, create an interface object table that differs slightly from the earlier object tables. Because the purpose here is *interface design*, rather than *program design*, at this point you can ignore operations and concentrate on the appearance of the screen objects. The interface object table will allow you to specify which objects will appear and what the initial values of some of their attributes will be. The interface object table for the sample problem could begin like this:

Object	Attributes	Initial Value
Window	Caption	'Library Locater'
	BackColor	grey
Box 1	BackColor	green
Button 1	Caption	'000-005'
Button 2	Caption	'005-120'
...
Box 2	BackColor	yellow
...
Box 3	BackColor	blue
...
Button <i>n</i>	Caption	'Quit'

This table becomes a reference for the programmer when development commences. Using the interface object table to set the captions and back colours, one possible interface could be:



Procedural algorithms that use repetition structures often need to test for sentinel values. In the library locator example, the selection structure will occur inside a loop that is terminated when the user selects the 'Quit' option. Although the repetition algorithm will look like many procedural algorithms with a sentinel value of 'Quit', the code that is produced will potentially include no repetition structures. With thoughtful interface design, the user does not have to type in a sentinel value to leave the program. The exit condition can be represented as one of the options on the screen.

Having gone to the trouble to design the algorithm for a programmer to implement, don't overlook the interface design. The choices of interface design can have a significant impact on the complexity of your algorithm and on the way your algorithm is implemented in the programming language.

Chapter summary

Object-oriented design is a fundamentally different process to procedural design. Instead of decomposing the problem into functions, the problem is broken up into the objects in the system, and the attributes and operations for each object are identified.

Objects encapsulate their data and operations, and can be regarded as 'black boxes' for the purposes of large system design. Objects are instantiated from classes that are templates defining the attributes and operations for objects of the type, but individual objects of the same type may store different data values in their attributes. The operations and attributes that are accessible by external objects are described as public, and those that are internal to the object are private. The steps in designing an object-oriented solution for a programming problem are:

- 1** Identify the classes, together with their attributes, responsibilities and operations.
- 2** Determine the relationship between the objects of those classes.
- 3** Design the algorithms for the operations, using structured design.
- 4** Develop a test or driver algorithm.

Interface design for visual programming languages uses object-oriented design principles. Interface objects have operations and attributes. The choice of interface design can reduce the complexity of both an algorithm and the resulting program.

Programming problems

- 1** A parts inventory file contains inventory records each with the fields:

- part number (6 characters, 2 alpha and 4 numeric, e.g. AA1234)
- part description
- inventory balance.

Design a program that will read the file, validate the part number for each record and print the details of all valid inventory records that have an inventory balance equal to zero.

- a** Design the class table.
- b** Write an algorithm for each operation in the table.
- c** Write an algorithm for a driver class.

- 2** Expand the algorithm for the TestStudent class used in Example 11.2 so that it tests all of the public operations and attributes in the Student class.

- 3** Design a class that calculates and prints the balance owed by each customer of a phone company during the billing period. Phone calls are charged at 25c per minute

and charges are added to the balance as the phone calls are completed. The constructor will set the balance owing to zero. Private operations will be needed to calculate the call charges as they are added and to update the balance following the calculation. Use a public operation to add a new call to the balance.

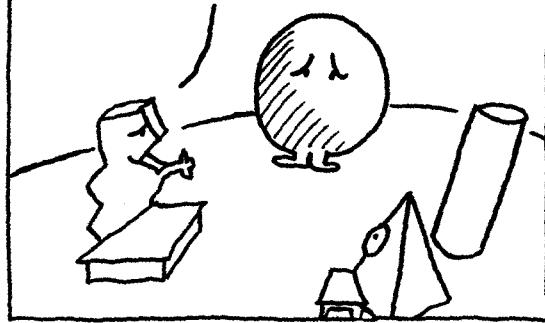
- a Design the class table.
 - b Write an algorithm for each operation in the table.
 - c Write an algorithm for a Test class that tests all operations and attributes.
- 4 Write an object-oriented design for a program that will prompt for and receive the diameter of a circle, and calculate and display the circumference and the area of that circle.
 - a Design the class table.
 - b Write an algorithm for each operation.
 - c Write an algorithm for a circle Test class.
- 5 Design a class to manage a share portfolio. Shareholdings are identified by the company name, the number of shares purchased, the date of the purchase, the cost per share and the current price per share. You must be able to calculate the value of current shares, the profit or loss made on the stock purchase and be able to sell the shares.
 - a Design the class table.
 - b Write an algorithm for each operation.
 - c Write an algorithm for a Test class.
- 6 Write an algorithm for a default constructor for the Share class in Problem 5.
- 7 A library needs a program to keep track of the current loans. Each book has a title, an ISBN, an author, publisher, publication date, call number and a unique accession number. Library patrons have a unique user code, name, street address, postcode and an overdue balance that can be changed when a fine is imposed or paid. The balance can be printed, as well as the user code and the patron's name and phone number. When a loan is made, the patron's user code and the loan item's accession number are recorded, as well as the date borrowed and the due date. When a loan is overdue, a fine of \$1 per day is charged to the borrower's overdue balance.

Design one or more classes that could be used by this program. Create the class table and write the algorithms for all operations. Treat each class you design as though it has no relationship to the other classes.
- 8 Design the interface for a program that will prompt a user for his or her astrological sign and display the current prediction for the user. Prepare the interface object table and the interface layout.
- 9 In the 'Programming Problems' section of Chapter 8, read Problem 9 and design an interface for the program. Prepare the interface object table and the interface layout.
- 10 In the 'Programming Problems' section of Chapter 9, read Problem 10 and design the screen interface for the program. Prepare the interface object table and the interface layout. Explain how the interface design may impact on the algorithm for this problem.

Chapter 12

Simple object-oriented design for multiple classes

I'd like you to welcome
a new OBJECT
to this CLASS



Objectives

- To introduce the concept of program design for multiple classes
- To describe relationships between classes
- To introduce polymorphism and operation overriding in object-oriented design
- To develop an object-oriented design solution to a multi-class problem

Outline

12.1 Object-oriented design with multiple classes

12.2 Programming example with multiple classes

Chapter summary

Programming problems

12.1

Object-oriented design with multiple classes

A major advantage of object-oriented programming languages is their usefulness in constructing large programs; however, it would be very unusual for a program to make use of only one class. In designing programs that use multiple classes, we need to consider not only the individual class design but also the relationships between the classes and therefore between the objects that are instantiated from those classes. It is also useful to understand how the design of classes can evolve.

Notations

Object-oriented design approaches and object-oriented programming have taken some time to mature as methodologies. A notation called UML, standing for Unified Modelling Language, has emerged from the work of Rumbaugh, Booch and Jacobson. The notation in this chapter is based on a simplified UML standard.

UML allows a designer to represent the relationships between classes as well as between objects. This chapter will introduce some of the UML graphical notation used to design classes and their relationships.

Relationships between classes

When more than one class is used in a program, there can be three main types of relationships between objects from two or more of the classes.

- 1 The simplest relationship is between two classes that are independent of each other but one class might *use* the services the other provides. This relationship is called an *association*.
- 2 A class may be made up of other classes, or contain other classes that are part of itself. A collective relationship may be either *aggregation* or *composition*.
- 3 A class may *inherit* all the attributes and operations of a parent class, but is given a unique name as well as its own additional attributes and operations. This form of relationship between a parent and a child class is *generalisation*.

Association

An association between two classes is required when the classes need to interact or communicate for the program to achieve its purpose. For example, a Car class and a Garage class are independent, although a car may sometimes *use* garage services, such as parking. As a result of this association, objects that are instantiated from these two classes will be able to communicate by passing or receiving messages. In some circumstances, a Garage object will be created without a Car object, and vice versa.

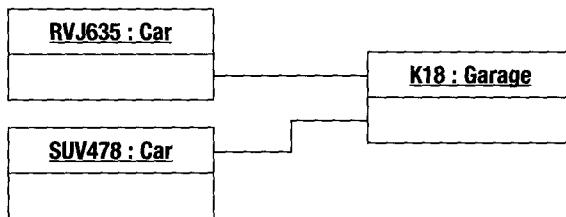
Just as instances of classes are called objects, instances of associations are called *links*. In a class diagram, a small arrow indicates the direction to read the description of the association, although association allows communica-

tion in both directions. Classes that have an association are able to communicate by passing messages. The numbers at each end of the association show how many objects can have this association, when it is instantiated.



A class diagram showing association between Car and Garage

Read this example as ‘A Car object uses a Garage object’, or in the other direction, ‘A Garage object is used by 1 or 2 Car objects’.



An object diagram showing links between Car objects and a Garage object

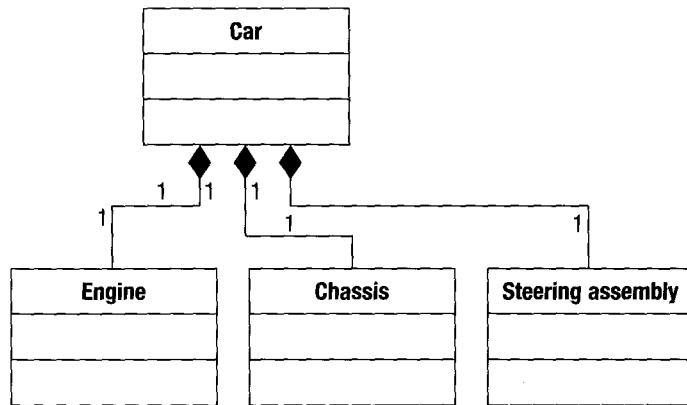
Aggregation and composition

Aggregation and composition are special forms of association where ~~objects~~ of one class, the container class, are *made up of* or *consist of* other ~~objects~~ from other classes, the component classes. These are *whole-part associations*: one class is the *whole* that is made up of the *parts*. The class at the *whole* end can have attributes and operations of its own, as well.

In its mildest form, called *aggregation*, the part or *component* classes ~~that~~ make up the whole are able to exist without necessarily being part of ~~the~~ aggregation. For example, they could be created before they are added ~~to~~ the aggregation.

In its strongest form, called *composition*, the component classes can ~~only~~ exist during the lifetime of the container object and cannot exist outside ~~the~~ composition. When the whole object is created, all of the needed ~~component~~ objects are created. When the whole object is destroyed, all of its ~~component~~ objects are also destroyed.

A car is a *composition* of many parts, including an engine. So, every ~~object~~ of the Car class needs to have an object of the Engine class to be able to ~~work~~ effectively, and an object of the Engine class cannot fulfil all of its ~~responsibilities~~ outside its vehicle. An Engine object can only be part of one Car ~~at a~~ time. These relationships can be shown graphically.



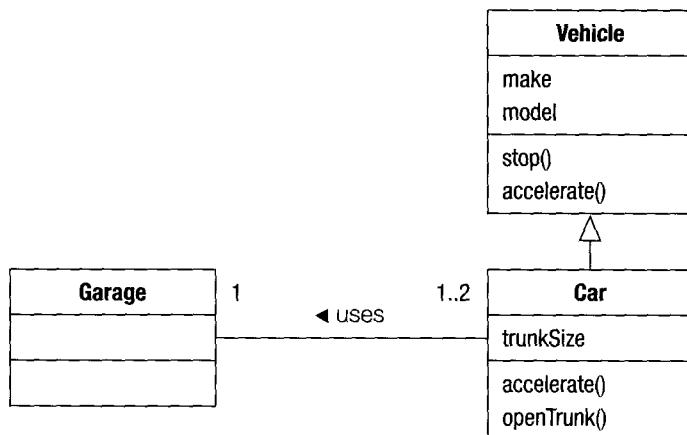
Car composition class diagram

The filled-diamond headed line and the number 1 between Car and Engine show that an engine can only be part of a car and any engine can belong to only one car. Aggregation uses an unfilled diamond.

Generalisation

Sometimes, one class shares features with another class, but has enough differences to deserve its own identity. The first class then becomes a *kind of* or *type of* the second class. For example, cars are *a type of* vehicle, just as bicycles and buses are types of vehicle. *Generalisation* is a class hierarchy that lets us group the shared attributes and operations into a top-level class, and then to define one or more lower-level classes with extra or different attributes and operations. The top-level class, also called the *parent class* or *superclass*, has shared attributes and operations, and the *child classes* or *subclasses*, *inherit* these, adding their own attributes and operations to make them distinct.

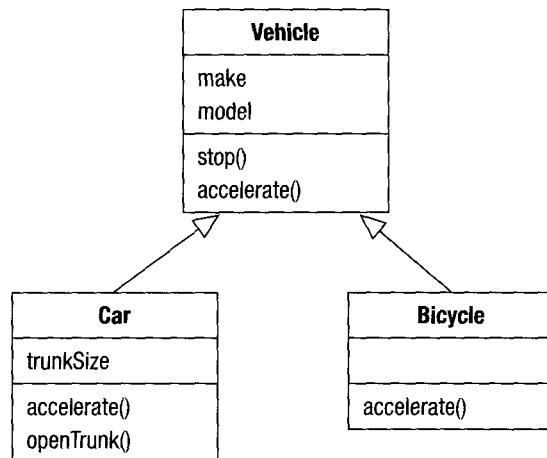
A Vehicle class with attributes shared by all vehicles, such as make and model, and operations such as stop(), turn() and accelerate(), would partly describe both cars and bicycles, but not completely. We can define a Car child-class that will *inherit* all the attributes and operations from the Vehicle superclass. But a car has additional attributes that are particular to cars, such as trunkSize, and operations, such as openTrunk().



Using UML notation, the open-headed arrow between Vehicle and Car shows that Car is *a type of* Vehicle. Generalisation notation is different from association notation in UML, because it represents a type of class hierarchy instead of an association.

Polymorphism

Polymorphism, meaning many-shaped, describes the use of operations of the same name for a variety of purposes. Using the car example, the bicycle is another type of vehicle that inherits all the same attributes and operations as a car. But accelerate() for a car is quite different from accelerate() for a bicycle. Both need the operation, but it is achieved in quite different ways for objects of each type. The car and the bicycle classes will each need to provide their own definition of accelerate() to be complete. This is an example of polymorphism.



Operation overriding occurs when a parent class provides an operation, but the inheriting child class defines its own version of that operation. Both Car and Bicycle inherit an **accelerate()** operation from their parent, the superclass, Vehicle. The **accelerate()** operation in the Car object is different from the **accelerate()** operation in the Bicycle object, although it has the same name and purpose. Both operations are probably different from the **accelerate** operation in Vehicle. Therefore, whenever **accelerate()** is called for by an object, such as Car or Bicycle, the version of the operation used by that particular object will be utilised. In other words, the operation in a subclass will *override* the operation in the superclass. In pseudocode, the operation can be written with the dot notation, such as `car.accelerate()` or `bicycle.accelerate()`.

In another type of polymorphism, *overloading*, several operations in a single class can have the same name. To differentiate between the operations, each one must have a different number of parameters. For example, a bicycle class may have two versions of the **stop()** operation, such as **stop(handbrake)** and **stop(handbrake, footbrake)**. When a call is made to the operation, **stop()**, the number of arguments in the call is evaluated and the correct **stop()** operation invoked.

12.2 Programming example with multiple classes

In Chapter 11, Example 11.1, the problem definition required only one major class. The problem can now be extended so that it more closely resembles a real application. The additional information for the problem definition follows the Weekly pay calculation paragraph of the problem statement, and illustrates program design that uses inheritance and the composition association.

EXAMPLE 12.1 Calculate employee's pay

A program is required by a company to read an employee file containing the employee's number, pay rate and the number of hours worked in a week. The program is then to validate the pay rate and the hours worked fields and, if valid, compute the employee's weekly pay and print it along with the input data on to a payslip.

Validation: According to the company's rules, the maximum hours an employee can work per week is 60 hours, and the maximum hourly rate is \$25.00 per hour. If the hours worked field or the hourly rate field is out of range, the input data and an appropriate message is to be printed and the employee's weekly pay is not to be calculated.

Weekly pay calculation: Weekly pay is calculated as hours worked times pay rate. If more than 35 hours are worked, payment for the overtime hours worked is calculated at time-and-a-half.

Most employees are paid according to these procedures, but two additional categories of employees are paid differently.

- Programmers can work on only one project. They do not earn overtime, but they are entitled to a \$50 bonus for each week the project is running ahead of schedule. (An existing Project class has an operation, called `getStatus(daysAhead,empNumber)`, that returns the number of days ahead of or behind schedule.)
- Sales representatives earn commissions of \$200 if their weekly sales exceed \$19 999.99, and \$100 if their sales are in the range of \$10 000 to \$19 999.99. (An existing WeeklySales class that is associated with each salesperson has an operation, called `getSales(empSales,empNumber)`, that returns the dollar and cents value of the total sales for that representative for that week.)

The field in the input file representing information about the employee type contains the character 'E' for all employees, except programmers and sales representatives, which have the input fields of 'P' and 'R', respectively.

As the payroll is processed, the weekly pay for each employee should be added to the wages total for that week and printed after the last employee has been processed. All the data that has been printed on the payslips should also be written to a file.

Notice that the problem definition refers to two existing classes, Project and WeeklySales. Since these objects already exist, they can be used in this program as well as in other programs that may need them. This is the principle of object reuse.

Step 1: Identify the classes, together with their attributes, responsibilities and operations

With a complex problem, this step involves a repeated process of identification, development and refinement, until you are satisfied with the design of the class table. To simplify the process, the problem definition in this example will be divided into three parts: the original problem; the next three paragraphs; and the final two paragraphs. The objects, attributes, responsibilities and operations will be identified and refined in three stages.

Stage 1

1.1 Payslip Class

A class table has already been developed for the original problem. The Payslip class was developed in Chapter 11 for Example 11.1, as follows:

Class	Attributes	Responsibilities	Operations
Payslip	-empNumber	Validate data	+setEmpNumber(empNumber)
	-payRate	Report validity of data	+setPayRate(payRate)
	-hoursWorked	Calculate weekly pay	+setHoursWorked(hoursWorked)
	-weeklyPay	Print a payslip for each valid timesheet	+getValidInput(validInput)
	-validInput		-setPay()
Payroll	timesheet data file	Read timesheet data file	+printPayslip()
			+readTimesheet()
			+run()

Stage 2

To identify the classes and objects for the new parts of the problem, we need to underline the relevant nouns and adjectives in the next section of the problem statement, as follows:

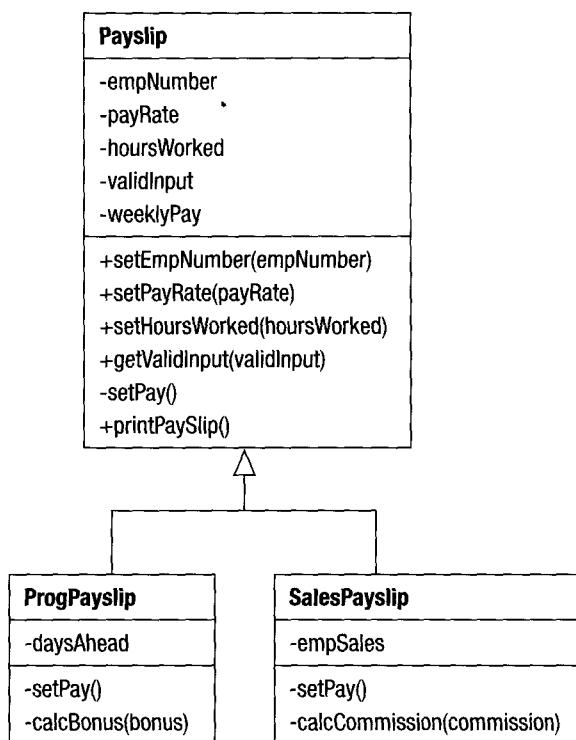
Most employees are paid according to these procedures, but two additional categories of employees are paid differently.

- Programmers can work on only one project. They do not earn overtime, but they are entitled to a \$50 bonus for each week the project is running ahead of schedule. (An existing Project class has an operation, called getStatus(daysAhead, empNumber), that returns the number of days ahead of or behind schedule.)

- Sales representatives earn commissions of \$200 if their weekly sales exceed \$19 999.99, and \$100 if their sales are in the range of \$10 000 to \$19 999.99. (An existing WeeklySales class that is associated with each salesperson has an operation, called getSales (empSales, empNumber), that returns the dollar and cents value of the total sales for that representative for that week.)

1.2 Programmer and Sales representative classes, attributes and operations

Looking at the above nouns and noun phrases, Programmers and Sales representatives are types of employees, with different payroll calculations, so in this program they are likely to be objects, and probably types of Payslip object. When objects are grouped like this according to common properties, but differ in some attributes or operations, they can be organised into an inheritance hierarchy. The parent class will be Payslip and its subclasses will be named ProgPayslip and SalesPayslip, respectively.



The diagram above shows the ProgPayslip and SalesPayslip classes as children of Payslip, inheriting the operations and attributes of the parent.

We are told that a class called Project already exists. The Project class can be considered a 'black box', charged with the responsibility of informing an external object of the number of days that the project is ahead of or behind schedule.

Similarly, there is an existing class called WeeklySales. This class is also considered a black box, charged with the responsibility of returning the dollar and cents value of the total sales or weekly sales for each sales representative.

Programmers aren't paid overtime, but earn a bonus, based on the weeks that their projects are ahead of schedule. The number of daysAhead of schedule can be communicated by a Project object and is therefore a suitable attribute for ProgPayslip, because bonus can be calculated from it. As part of the weekly pay, bonus is a temporary variable, not required by external objects, and its calculation is trivial, so we will not make it a derived attribute. Derived attributes can avoid the overhead of recomputation but should be used with caution, because whenever the base attributes from which they are derived are updated, they too must be updated.

Sales representatives earn commission based on their weekly sales. The weekly sales earned by a sales representative can be communicated by a WeeklySales object, so empSales can become an attribute of SalesPayslip. A commission can be calculated from empSales and added to the pay for that representative. Like bonus, commission is not required as an attribute.

Responsibilities

ProgPayslip and SalesPayslip objects have the same responsibilities as Payslip objects. They differ only in the way they calculate the weekly pay for the programmer or sales representative, respectively.

Operations

Taking into consideration the responsibilities of each class, underline the verbs and verb phrases relevant to the problem, as shown here.

Most employees are paid according to these procedures, but two additional categories of employees are paid differently.

- Programmers can work on only one project. They do not earn overtime, but they are entitled to a \$50 bonus for each week the project is running ahead of schedule. (An existing Project class has an operation, called getStatus(daysAhead, empNumber), that returns the number of days ahead of or behind schedule.)
- Sales representatives earn commissions of \$200 if their weekly sales exceed \$19 999.99, and \$100 if their sales are in the range of \$10 000 to \$19 999.99. (An existing WeeklySales class that is associated with each salesperson has an operation, called getSales (empSales,empNumber), that returns the dollar and cents value of the total sales for that representative for that week.)

For a programmer, the existing Project class and its operation, getStatus(daysAhead,empNumber) will handle the operations suggested by the verb phrase: running ahead of schedule. The daysAhead value can be passed to a ProgPayslip object as a parameter value in a constructor or a mutator. The ProgPayslip object can then calculate the bonus earned, with a private operation called calcBonus(), handling the operation suggested by the phrase entitled to a bonus.

Similarly, for a sales representative, the existing WeeklySales class and its operation getSales(empSales,empNumber) will handle the operations suggested by the verb phrase: returns the dollar and cents value of the total sales for that week. The value for empSales can be passed to the SalesPayslip constructor or to a mutator. Each sales representative will earn commission, so a private operation, called calcCommission(), is required for the SalesPayslip class. The verb exceed, and its related information, is useful in the algorithm to calculate the commission, but does not contribute to the class and object development stage.

We can now extend the class table, adding the new classes, their attributes, responsibilities and operations. The double colon between the class names shows that they are derived from the Payslip class and inherit the attributes and operations of that class. The additions to the class table are shaded.

Class	Attributes	Responsibilities	Operations
Payslip	-empNumber	Validate data	+setEmpNumber(empNumber) +setPayRate(payRate) +setHoursWorked(hoursWorked) +getValidInput(validInput)
	-payRate	Report validity of data	-setPay()
	-hoursWorked	Calculate weekly pay	+printPayslip()
	-weeklyPay	Print a payslip for each valid timesheet	
Payroll	-validInput timesheet data file	Read timesheet data file	+readTimesheet() +run()
			-calcBonus(bonus)
Payslip:: ProgPayslip	-daysAhead	Calculate bonus	-calcCommission(commission)
Payslip:: SalesPayslip	-empSales	Calculate commission	
Project		Supply number of days that a project is ahead or behind schedule	+getStatus(daysAhead,empnumber)
WeeklySales		Supply weekly sales figures for a sales representative	+getSales(empSales,empnumber)

Stage 3

Now let's look at the last two paragraphs in the problem definition. The noun phrases have been underlined to help identify the objects and their attributes, and the verb phrases have been double underlined to identify the operations for each object.

The field in the input file representing information about the employee type contains the character 'E' for all employees except programmers and sales representatives, which have the input fields of 'P' and 'R', respectively.

As the payroll is processed, the weekly pay for each employee should be added to the wages total for that week and printed after the last employee has been processed. All the data that has been printed on the payslips should also be written to a file.

In this program, the original timesheet data is converted from raw text format into Payslip objects. The employee type field in the data file acts as a temporary variable directing the driver to construct Payslip, ProgPayslip or SalesPayslip objects. Once the different Payslip objects have been constructed, employee type is no longer needed. From then on, polymorphism and encapsulation will automatically ensure that the correct operation is performed, whereas in a procedural system, procedures would be needed to differentiate the different types of employees.

1.3 Refining the existing classes

In Chapter 11, the Payroll class acted as a driver program, serving as an interface between the world and Payslip objects. The Payroll class was responsible for reading the input file and instantiating the Payslip objects that were themselves responsible for the processing and printing of the weekly pay. Now the problem statement requires that the data be written to another file, and a wages total for that week be added to the weekly payroll.

Reading input data from a text file, then processing, printing and writing back to a file, one payslip at a time, is slow. A more efficient approach is to organise all valid payslips to be *parts of* another class, a container class, and to leave the interface tasks, such as reading the file and writing to the file, to the driver object. The container class can then undertake the higher-level responsibilities of calculating the total weekly wages. The payslips will be stored in memory in a data structure (such as an array), which is an attribute of the container class. The driver class can then close the input file, and when processing is complete, save the entire data structure representing all the Payslip objects to the output file, in one operation.

With this approach each class is responsible for quite different tasks:

- 1 the driver, or interface, class, for interaction with the outside world
- 2 the container class, for managing the payslips and calculating the total weekly wages
- 3 the Payslip class, for its existing responsibilities as well as those needed for the additional subclasses.

The new approach requires an additional class to act as the container class for the Payslip objects. The new class can be called WeeklyPayroll.

1.4 The WeeklyPayroll class

Attributes

In object-oriented programming languages, a data structure can be contained in a single object. The WeeklyPayroll class will contain an attribute that is an array data structure, called *payslips*. The derived attribute, *payslipQty*, will be used as an index of the array, and will record the number of payslips in the array. The wages total for that week is clearly connected to the container class. When each payslip is added to the payslips array, its weekly pay can be added to a derived attribute, *totalPay*. Recomputation of the total wages for the week will therefore not be necessary – for example, during printing.

Responsibilities

The WeeklyPayroll class is responsible for organising the payslips for each employee and for calculating the total wages.

Operations

To help with the organisation of the payslips, an operation called *addPayslip()* can add a payslip to the array. As the verb added indicates, the total weekly wages will be calculated by adding the *weeklyPay* from each Payslip object to the value of the *totalPay* attribute of the WeeklyPayroll object. To achieve this, the WeeklyPayroll class may need a operation that we will call *calcTotalPay()*. This mutator will remain private to ensure that the derived attribute is not updated independently of its base attribute. A *printWeeklyPayroll()* operation can use the *printPayslip()* operation in each payslip to print that payslip, and then print the value of *totalPay*, handling the problem statement phrase 'printed after the last employee has been processed'.

As with the Payslip class, accessors should be included in the WeeklyPayroll class to ensure that external objects can use WeeklyPayroll objects according to their particular needs. An accessor, *getPayslip(empNumber, payslip)*, can use an employee number to locate an employee's payslip in the structure. Another accessor, *getTotalPay()*, will supply the value of the total wages for that week.

The class table for the class to contain payslips is as follows:

Class	Attributes	Responsibilities	Operations
WeeklyPayroll	-payslips	Create a data structure of payslips	+addPayslip(payslip)
	-payslipQty	Supply a payslip for an employee number	+getPayslip(empNumber, payslip)
	-totalPay	Calculate the total weekly wages	-calcTotalPay()
		Supply the total weekly wages	+getTotalPay(totalPay)
		Print all payslips and the total weekly wages	+printWeeklyPayroll()

1.5 The Payroll class

After adding the container class, changes are needed for the driver class, Payroll, which we created in the previous chapter. The new Payroll object can obtain data from the existing Project and WeeklySales objects and provide it to the relevant payslips when they are instantiated. In object-oriented programming, files are considered objects in their own right, with operations for file reading and file writing. The driver can take responsibility for writing to the output file, which, like the input file, can be an attribute.

Structured techniques may be needed to break down the Payroll class run() operation into more than one module to perform these additional tasks.

The Class table below shows the entire class structure for the program and highlights the advantages of object-oriented design for large programs. In spite of the substantially more complicated problem definition, the only work required is in the shaded areas. The other parts of the program can be developed later or are already in place and hidden from us except when we need them. Notice that no changes have been made to the existing Payslip class even though two subclasses have extended it. The getPay() accessor has been highlighted because it might be used in the extended example, but good practice would have determined its inclusion in the original class.

Class	Attributes	Responsibilities	Operations
Payslip	-empNumber	Validate data	+setEmpNumber(empNumber)
			+setPayRate(payRate)
			+setHoursWorked(hoursWorked)
	-payRate	Report validity of data	+getValidInput(validInput)
	-hoursWorked	Calculate weekly pay	- setPay()
	-validInput	Print a payslip for each valid timesheet	+printPayslip()
	-weeklyPay	Supply the weekly pay amount	+getPay(weeklyPay)
Payroll	timesheet data file	Read timesheet data file	+readTimesheet()
	weeklyrun data file	Write weekly run data file	+writeWeeklyRun()
			+run()
Payslip:: ProgPayslip	-daysAhead	Calculate bonus	-calcBonus(bonus)
Payslip:: SalesPayslip	-empSales	Calculate commission	-calcCommission(commission)
Project		Supply number of days that the project is ahead or behind schedule	+getStatus(daysAhead,empNumber)

WeeklySales		Supply weekly sales figures for a sales representative	+getSales(empSales,empNumber)
WeeklyPayroll	-payslips	Create a data structure of payslips	+addPayslip(payslip)
	-payslipQty	Supply a payslip for an employee number	+getPayslip(empNumber, payslip)
	-totalPay	Calculate the total weekly wages	-calcTotalPay()
		Supply the total weekly wages	+getTotalPay (totalPay)
		Print all payslips and the total weekly wages	+printWeeklyPayroll()

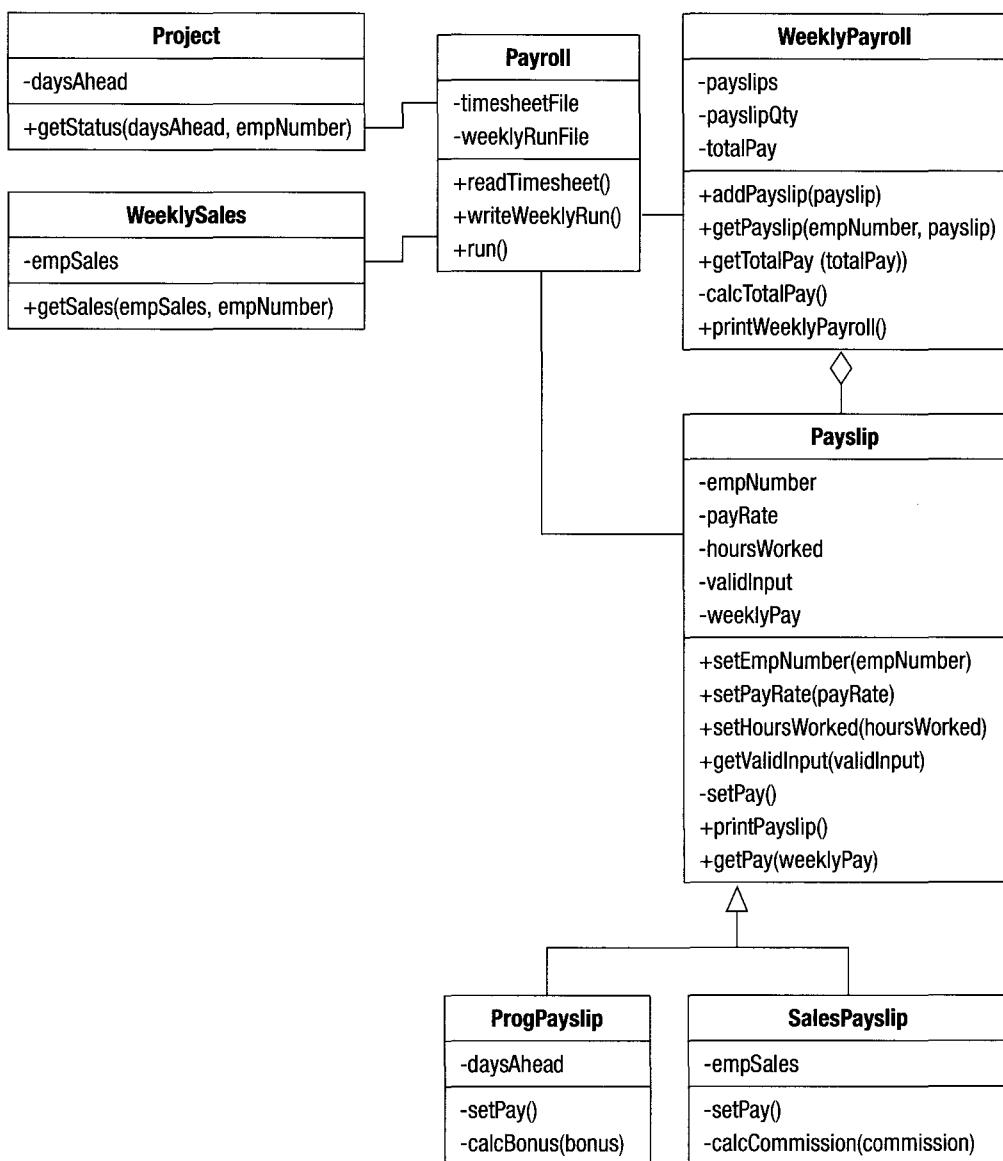
Step 2: Determine the relationships between the classes

In a program of this size, you may have determined some or all of the relationships between objects of different classes while you were identifying attributes and operations. Usually some changes will be made, and in this program, the driver class, Payroll, needs to be changed. The Payroll object uses WeeklyPayroll, Payslip, Project and WeeklySales objects to help it perform its tasks.

In Step 1, the Programmer Payslip and Sales Representative Payslip classes were identified as types of Payslip. In this *inheritance* relationship, ProgPayslip and SalesPayslip are child classes of Payslip. The ProgPayslip and SalesPayslip classes will inherit their own copies of all attributes from the Payslip class and augment them with some that specifically relate to their requirements. They will also extend the Payslip operations with some of their own. Notice in the class table that the child classes are shown next to the parent class, with the relationship indicated by the colon separating them. In the diagram, they are shown connected by open-headed arrows.

The WeeklyPayroll object is *composed of* multiple Payslip objects of each type. The components of composition classes are not always just of one class; they may include many other classes as well.

It is useful to view a simple graphical representation of the relationships between the program's classes.



The open-headed arrow connecting **Payslip** to **ProgPayslip** and **SalesPayslip** shows the inheritance child-parent relationship. Notice that the diamond line connecting **WeeklyPayroll** and **Payslip** is open, rather than shaded. This denotes that the elements in the relationship may also exist independently of the association.

This is a very simple representation of the classes in the program. By using all the features of UML, a more complete picture can be created. Because UML is a sophisticated modelling language, it cannot be covered in detail in this book, but it is certainly worth exploring in more depth to further develop your object-oriented design skills.

Step 3: Design the algorithms for the operations, using structured design

Algorithms for new or changed operations will be described in this section. Some parts of the class table have operations that already exist. Their algorithms will not be described. As with the Payslip class, constructors, mutators and accessors can do much of the work.

3.1 ProgPayslip Class

The ProgPayslip constructor is very simple because it can use the parent class constructor to initialise the inherited attributes, including validInput and weeklyPay.

```
ProgPayslip (inEmpNumber,inPayRate,inHoursWorked,inDaysAhead)
    Inherit Payslip (inEmpNumber, inPayRate, inHoursWorked)
    IF validInput THEN
        Set daysAhead to inDaysAhead
        setPay()
    ENDIF
END
```

Each ProgPayslip object inherits the attributes, constants and operations from the Payslip object to calculate the pay for a normal employee. It also contains project data just for programmers.

However, the algorithm that has already been designed to calculate pay in normal payslips does not include the programmer's bonus. A new operation is required to calculate the pay for programmer payslips.

Using polymorphism, the setPay() operation can be defined differently in the ProgPayslip class, overriding the version inherited from its parent. The ProgPayslip setPay() operation will automatically be used when pay is calculated for a programmer payslip. The setPay() operation in the ProgPayslip class uses a private method to calculate the bonus. BONUSRATE is a constant storing the current bonus rate for a week.

```
setPay()
    calcBonus(bonus)
    weeklyPay = (payRate * NORM_HOURS) + bonus
END

calcBonus(bonus)
    IF daysAhead > 0 THEN
        bonus = daysAhead / 7 * BONUSRATE
    ENDIF
END
```

3.2 SalesPayslip Class

The SalesPayslip constructor follows the same lines as the ProgPayslip constructor.

```
SalesPayslip(inEmpNumber,inPayRate,inHoursWorked,inEmpSales)
Inherit Payslip(inEmpNumber, inPayRate, inHoursWorked)
IF validInput THEN
    Set empSales to inEmpSales
    setPay()
ENDIF
END
```

An operation is required to calculate the commission for the sales representative, based on the empSales attribute. MIN_SALES and MAX_SALES are the constants \$10 000 and \$19 999.99, respectively. The operation, setPay(), inherited from the Payslip class, will need to be changed for the SalesPayslip class to add the calculated commission to the pay calculation. Again, using polymorphism and operation overriding, the setPay() operation for the SalesPayslip class replaces the inherited operation.

```
calcCommission(commission)
IF empSales > MAX_SALES THEN
    commission = 200
ELSE
    IF empSales >= MIN_SALES THEN
        commission = 100
    ENDIF
ENDIF
END
```

Note that weeklyPay has already been initialised when the SalesPayslip constructor called the Payslip constructor, which in turn called its own setPay() operation.

```
setPay()
calcCommission(commission)
weeklyPay = weeklyPay + commission
END
```

3.3 WeeklyPayroll Class

The default constructor for this class initialises the attributes with default values. The mutator, addPayslip(payslip), adds a payslip to the array, payslips. Note that square brackets are used here to distinguish between an array and an operation. The attribute, payslipQty, records the number of payslips in the array. The operation, calcTotalPay(), adds the pay from the current payslip to the total wages for the week.

```

addPayslip(payslip)
    payslips[payslipQty] = payslip
    payslipQty = payslipQty + 1
    calcTotalPay(payslip)
END

calcTotalPay(payslip)
    payslip.getPay(weeklyPay)
    totalPay = totalPay + weeklyPay
END

```

Assuming that a single dimensional array data structure is used, the algorithms for the operations of this class are straightforward and can be found in Chapter 7. These operations are:

getPayslip(empNumber, payslip)	searches the array for payslip details for a particular empNumber
printWeeklyPayroll()	prints all the payslips in the array and then prints the total weekly wages

Step 4: *Develop the program interface and a driver class to implement it*

Now that the new classes have been defined, structured design techniques, as introduced in Chapter 8, can be used to update the internal operations of the Payroll driver.

A Define the problem

Input	Processing	Output
Timesheet data • empType • empNumber • hoursWorked • payRate Project • daysAhead WeeklySales • empSales	Read timesheet data Calculate weeklyPay for each employee Calculate totalPay for all employees Print valid payslips, totalPay Write valid payslips to file	Payslip file Printed payslips totalPay

This defining diagram is similar to:

- 1 the defining diagram for the top-down solution to this problem, found in Chapter 8, Example 8.5; and
- 2 the main algorithm, run(), in the driver object for the previous simpler object-oriented Payroll Processing Example 11.1, in Chapter 11.

However, there are two main differences. In this example:

- 1 The first item of payroll data encountered for each payslip is the employee type. The processing of the rest of the payroll data for that payslip depends on the type of employee: general employee (E), programmer (P) or sales representative (R).
- 2 The payslips are written to a file. For efficiency, the valid Payslip objects become part of an accessible weeklyPayroll object that can be printed faster than text file records, and can be written to the file in a single operation.

The driver algorithm controls and coordinates the different types of objects apparent in the program interface. Having designed the objects from the bottom up, top-down design can be applied to the driver algorithm. The remaining steps of structured design will be carried out by the objects themselves.

The Project and WeeklySales classes are already available in the system, and we know nothing of their internal workings or attributes, other than that the accessors needed for this program are available. Because the project and weeklySales objects are 'black boxes', their internal organisation is hidden from us. It does not matter to our program whether the data being returned is calculated, extracted from a data structure or drawn from a single attribute. In the algorithm, it should be clear that the project and weeklySales objects will be accessed. An 'import' statement with their class names will indicate this.

Import Project, WeeklySales

This now allows project and weeklySales objects to be instantiated in the program. The import statement should be the first line in the algorithm, above and outside any module.

The Payroll constructor can open the two files used by the program.

```
Payroll()  
  Open timesheetFile  
  Open weeklyRunFile  
END
```

Once the files are open, the main driver algorithm performs an input-process-output loop, reading timesheet data, and converting it to payslips that it writes to a file. The Payslip objects take responsibility for validating the input, calculating weekly pay and printing themselves. The weeklyPayroll object takes responsibility for accumulating the total wages for the week and printing it, together with all payslips. Using structured techniques, the run() algorithm will need to call a lower-level module, makePayslip(), which performs the task of creating payslip objects from text records.

```

run()
Create weeklyPayroll as new WeeklyPayroll()
Create project as new Project
Create weeklySales as new WeeklySales
DOWHILE more timesheet records
    makePayslip(project, weeklySales, payslip)
    payslip.getValidInput(validInput)
    IF validInput THEN
        weeklyPayroll.addPayslip(payslip)
    ENDIF
ENDDO
weeklyPayroll.printWeeklyPayroll()
Write weeklyPayroll to weeklyRunFile
END

```

For each employee, the makePayslip() operation reads text data from the timesheet data file. In addition, depending on the employee type, it can obtain the number of days that a programmer's project is ahead of or behind schedule from the Project object, or it can obtain a sales representative's weekly sales from the WeeklySales object. The relevant information is passed to the appropriate constructor, selected from the three types of payslip. From there, the different payslip objects can take responsibility for the validity of their attribute values and for calculating their weekly pay. The makePayslip() operation then returns the newly created payslip to the run() operation.

```

makePayslip(project, weeklySales, returnPayslip)
Read empType,empNumber,payRate,hoursWorked from timesheetFile
CASE OF empType
    P: project.getStatus(daysAhead,empNumber)
        Create payslip as new ProgPayslip(empNumber,payRate,hoursWorked,
            daysAhead)
    R: weeklySales.getSales(empSales,empNumber)
        Create payslip as new SalesPayslip(empNumber,payRate,
            hoursWorked, empSales)
    E: Create payslip as new Payslip(empNumber, payRate, hoursWorked)
ENDCASE
returnPayslip = payslip
END

```

Data validation in object-oriented programs

The above example included data validation, essential for robust programs. Mutator operations performed the data validation in the object-oriented approach, ensuring that attributes can only be set to valid values. In more advanced object-oriented programs, most error processing is handled by separate error-processing classes. In this case, the mutator would test for the error, and pass any errors it traps to the appropriate error-handling class for processing.

Comparing structured and object-oriented design

The Employee Pay programming example used in the previous chapter and extended in this chapter was first encountered as a structured design example in Chapter 8. You can now compare the two approaches. Although the structured approach was clearly less complicated for a simple example, the object-oriented solution was easily extended and easily reused code from existing classes, demonstrating the suitability of object-oriented design for larger systems. Encapsulation and information hiding assists this process, because programmers do not have to become familiar with all the code in an object-oriented program; they can see *what* existing classes can do without being concerned with *how* they do it.

Chapter summary

Most object-oriented programs need more than one class. Classes can be related to each other through association, by aggregation or composition (being in a whole–part relationship) or by inheritance (being a subtype of a class).

When classes are related by inheritance, all subclasses or child classes inherit the attributes and operations of the parent class and supplement them with attributes and operations needed by the subtype.

Through polymorphism, several operations may have the same name, but achieve their purposes through different methods.

Using operation overriding, a child class may substitute the parent class version of an operation with its own specific version. With operation overloading, several operations of the same name may have different numbers of parameters and different algorithms.

There are four steps in designing a solution to a simple multiple class problem:

- 1 Identify the classes, together with their attributes, responsibilities and operations.
- 2 Determine the relationships between the classes.
- 3 Design the algorithms for the operations using structured design.
- 4 Develop the program interface and a driver class to implement it.

This sequence of steps can be repeated several times, gradually filling in missing details until the solution is complete.

Programming problems

- 1 Draw a class diagram to show the relationship between the classes designed in your solution to Programming Problem 7 in Chapter 11. The solution should include at least a book class, a patron class and a book loan class. Write any operation algorithms needed to complete the solution.
- 2 Update the class diagram for the employee pay program to include all the identified operations and attributes. Complete the algorithms for the Project and WeeklySales constructors.
- 3 Prepare the driver algorithm for Programming Problem 7 in Chapter 11, based on your solution classes.
- 4 The library loan system in Programming Problem 7 in Chapter 11 needs to be able to accommodate non-book loans, such as videos, tapes and magazines. Magazines have a title, an ISSN rather than an ISBN, a volume and number, publisher, publication date, call number and a unique accession number. Videotapes have a title, publisher, publication date, call number and a unique accession number. Overdue videotapes are charged at \$2 per day. Modify your solution design accordingly, including the class diagram, class tables and algorithms where necessary.
- 5 Yummy Chocolates requires an object-oriented program for an online catalogue. The catalogue is to display the details of the range of handmade chocolates. Each chocolate product has a product code, a name, a picture and the price per 100 grams. Products can be added to the catalogue, deleted and modified.

Design this program, preparing class tables, a class diagram and algorithms for the operations. Write a driver algorithm.

Programming problems

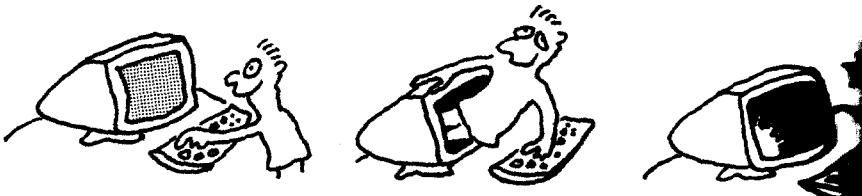
- 1 Draw a class diagram to show the relationship between the classes designed in your solution to Programming Problem 7 in Chapter 11. The solution should include at least a book class, a patron class and a book loan class. Write any operation algorithms needed to complete the solution.
- 2 Update the class diagram for the employee pay program to include all the identified operations and attributes. Complete the algorithms for the Project and WeeklySales constructors.
- 3 Prepare the driver algorithm for Programming Problem 7 in Chapter 11, based on your solution classes.
- 4 The library loan system in Programming Problem 7 in Chapter 11 needs to be able to accommodate non-book loans, such as videos, tapes and magazines. Magazines have a title, an ISSN rather than an ISBN, a volume and number, publisher, publication date, call number and a unique accession number. Videotapes have a title, publisher, publication date, call number and a unique accession number. Overdue videotapes are charged at \$2 per day. Modify your solution design accordingly, including the class diagram, class tables and algorithms where necessary.
- 5 Yummy Chocolates requires an object-oriented program for an online catalogue. The catalogue is to display the details of the range of handmade chocolates. Each chocolate product has a product code, a name, a picture and the price per 100 grams. Products can be added to the catalogue, deleted and modified.

Design this program, preparing class tables, a class diagram and algorithms for the operations. Write a driver algorithm.

Chapter Conclusion

13

Nearly to the END...
Press ESCAPE
then ENTER



Objectives

- Revision of the steps required to achieve good top-down program design

Outline

13.1 Simple program design

Chapter summary

13.1 Simple program design

The aim of this book has been to encourage programmers to follow a series of simple steps in order to develop solution algorithms to given programming problems. These steps are:

- 1 **Define the problem.** To do this, underline the nouns and verbs in the problem description. This helps to divide the problem into its input, output and processing components. These components can be represented in a defining diagram – a table that lists the inputs to the problem, the expected outputs, and the processing steps required to produce these outputs.

At this stage, you should be concerned with *what* needs to be done. When writing down the processing components, simply list the activities to be performed without worrying about *how* to perform them.

- 2 **Group the activities into subtasks or functions.** To do this, look at the defining diagram and group the activities in the processing component into separate tasks. These tasks will become the modules in the program. A module is dedicated to the performance of a single function.

Not all the activities to be performed may have been listed in the defining diagram. If the problem is large, only the top-level subtasks may have been identified at this stage. The basic aim of top-down design is to develop the higher-level modules first, and to develop the lower-level modules only when the higher-level modules have been established. You should concentrate on these higher-level functions before attempting to consider further subordinate functions.

- 3 **Construct a hierarchy chart.** The hierarchy chart shows not only the modules of the program, but also their relationship to each other, in a similar fashion to the organisational chart of a large company.

Just as a company director can change the organisation of the company to suit its operation, so you can change the organisation of the modules in the hierarchy chart. It is good programming practice to study the way the modules have been organised in the overall structure of the program, and to attempt to make this structure as simple and top-down as possible.

Note that you are still only concerned with the tasks that are to be performed. Once the hierarchical structure of the algorithm has been developed, you can begin to consider the logic of the solution.

- 4 **Establish the logic of the mainline of the algorithm.** Use pseudocode and the three basic control structures to establish this logic. Pseudocode is a subset of English that has been formalised and abbreviated to look like a high-level computer language. Keywords and indentation are used to signify particular control structures. The three basic control structures are simple sequence, selection and repetition.

Because you will have already identified the major functions of the problem, you can now use pseudocode and the three control structures to develop the mainline logic. The mainline should show the main processing functions of the problem and the order in which they are to be performed.

We saw that the mainline for most algorithms follows the same basic pattern. This pattern contains some initial processing before the loop, some processing within the loop, and some final processing after exiting the loop.

Chapter 10 developed general algorithms for four common business applications. These algorithms were for the generation of a report with a page break, a single-level control break program, a multiple-level control break program and a sequential file update program. The algorithms that were developed have good program structure and high modular cohesion, and you should use these algorithms as a guide for specific programming problems.

- 5 **Develop the pseudocode for each successive module in the hierarchy chart.** The algorithms for these modules should be developed in a top-down fashion. That is, the pseudocode for each module on the first level should be established, before attempting the pseudocode for the modules on the next or lower level. The modularisation process is complete when the pseudocode for each module on the lowest level of the hierarchy chart has been developed.
- 6 **Desk check the solution algorithm.** By desk checking the algorithm, you attempt to find any logic errors that may have crept into the solution.

Desk checking involves tracing through the logic of the algorithm with some chosen test data exactly as the computer would operate. The programmer keeps track of all major variables in a table as he or she walks through the algorithm. At the end of the desk check, the programmer checks that the output expected from the test data matches the output developed in the desk check.

This detection of errors, early in the design process, can save many frustrating hours during the testing phase. Most programmers who bypass this step do so because they either assume that the algorithm is correct, or because they believe that desk checking is not creative. While it may not be as stimulating as the original design phase, it is really just as satisfying to know that the logic is correct.

Chapter summary

This chapter has revised the steps required to achieve good top-down program design. Program design is considered good if it is easy to read and understand and **easy to alter**.

If you follow these six steps in the development of an algorithm, you will very soon achieve a high level of competence.

Appendix

Flowcharts

1

Outline

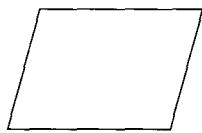
- Introduction to flowcharts and the three basic control structures
- Simple algorithms that use the sequence control structure
- Flowcharts and the selection control structure
- Simple algorithms that use the selection control structure
- The case structure, expressed as a flowchart
- Flowcharts and the repetition control structure
- Simple algorithms that use the repetition control structure
- Further examples using flowcharts
- Flowcharts and modules

This appendix introduces flowcharts as an alternative method of representing algorithms. Flowcharts are popular because they graphically represent the program logic through a series of standard geometric symbols and connecting lines. Flowcharts are relatively easy to learn and are an intuitive method of representing the flow of control in an algorithm. For simplicity, just six standard flowchart symbols will be used to represent algorithms in this text. These are:



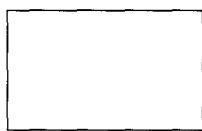
Terminal symbol

The terminal symbol indicates the starting or stopping point in the logic. Every flowchart should begin and end with a terminal symbol.



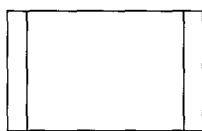
Input/Output symbol

The input/output symbol represents an input or output process in an algorithm, such as reading input or writing output.



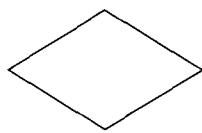
Process symbol

The process symbol represents any single process in an algorithm, such as assigning a value or performing a calculation. The flow of control is sequential.



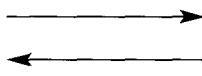
Predefined process symbol

The predefined process symbol represents a module in an algorithm – that is, a predefined process that has its own flowchart.



Decision symbol

The decision symbol represents a decision in the logic involving the comparison of two values. Alternative paths are followed, depending on whether the decision symbol is true or false.



Flowlines

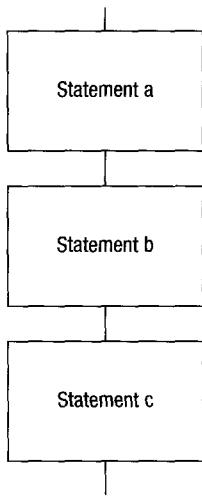
Flowlines connect various symbols in a flowchart, and contain an arrowhead only when the flow of control is not from top to bottom or left to right.

In this appendix the three basic control structures, as set out in the Structure Theorem in pseudocode, will be explained and illustrated using flowcharts.

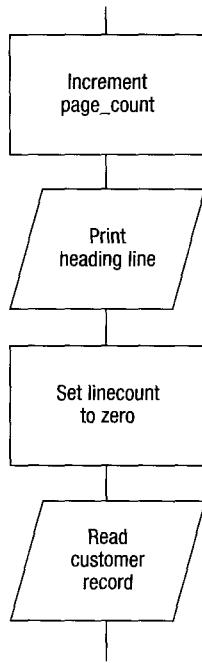
The three basic control structures

1 Sequence

The sequence control structure is defined as the straightforward execution of one processing step after another. A flowchart represents this control structure as a series of process symbols, one beneath the other, with one entrance and one exit.



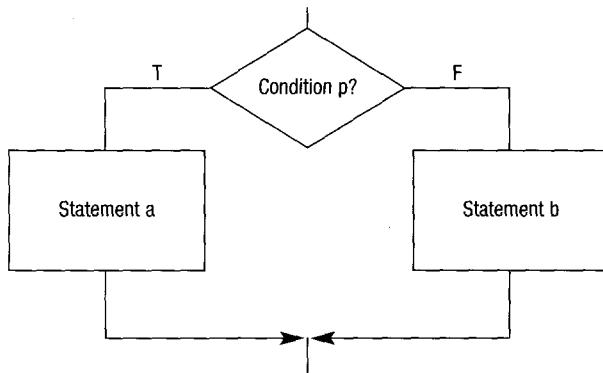
The sequence control structure can be used to represent the first four basic computer operations; namely, to receive information, put out information, perform arithmetic, and assign values. For example, a typical sequence of statements in a flowchart might read:



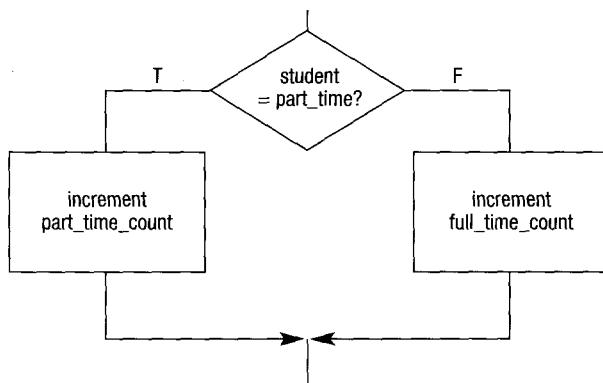
These instructions illustrate the sequence control structure as a straightforward list of steps, written one after the other, in a top-to-bottom fashion. Each instruction will be executed in the order in which it appears.

2 Selection

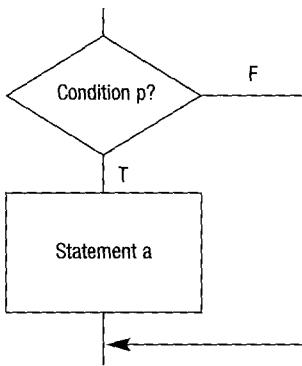
The selection control structure can be defined as the presentation of a condition, and the choice between two actions depending on whether the condition is true or false. This construct represents the decision-making abilities of the computer, and is used to illustrate the fifth basic computer operation; namely, to compare two variables and select one of two alternate actions. A flowchart represents the selection control structure with a decision symbol, with one line entering at the top, and two lines leaving it, following the true path or false path, depending on the condition. These two lines then join up at the end of the selection structure.



If condition p is true, the statement or statements in the true path will be executed. If condition p is false, the statement or statements in the false path will be executed. Both paths then join up to the flowline following the selection control structure. A typical flowchart might look like this:

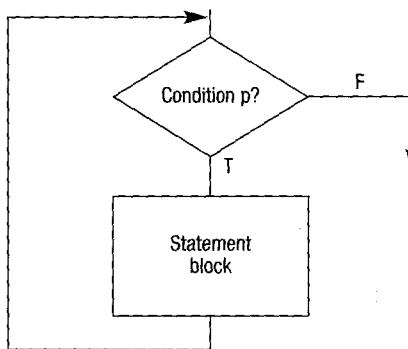


A variation of the selection control structure is the null ELSE structure, which is used when a task is performed only if a particular condition is true. The flowchart that represents the null ELSE construct has no processing in the false path.

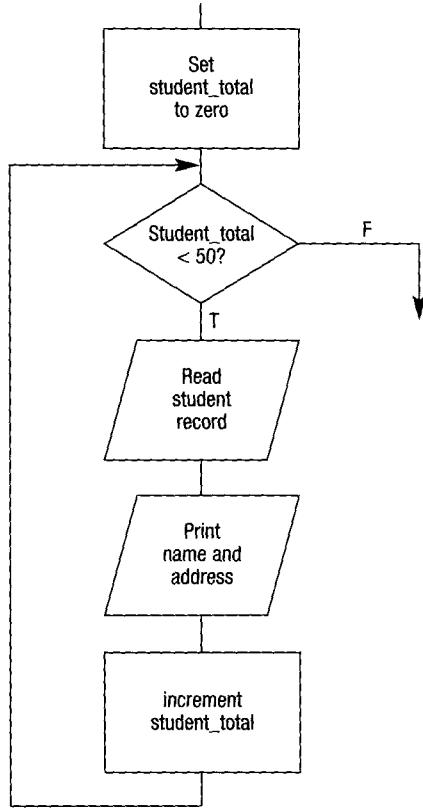


3 Repetition

The repetition control structure can be defined as the presentation of a set of instructions to be performed repeatedly, as long as a condition is true. The basic idea of repetitive code is that a block of statements is executed again and again, until a terminating condition occurs. This construct represents the sixth basic computer operation; namely, to repeat a group of actions. A flowchart represents this structure as a decision symbol and one or more process symbols to be performed while a condition is true. A flowline then takes the flow of control back to the condition in the decision symbol, which is tested before the process is repeated.



While condition p is true, the statements inside the process symbol will be executed. The flowline then returns control upwards to retest condition p. When condition p is false, control will pass out of the repetition structure down the false path to the next statement. We will now look at a flowchart that represents the repetition control structure:



Simple algorithms that use the sequence control structure

The following examples are the same as those represented by pseudocode in Chapter 3. In each example, the problem is defined and a solution algorithm developed using a flowchart. For ease in defining the problem, the processing verbs in each example have been underlined.

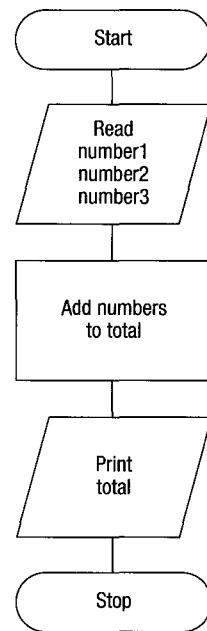
EXAMPLE 3.1 Add three numbers

A program is required to read three numbers, add them together and print their total.

A Defining diagram

Input	Processing	Output
number1	Read three numbers	
number2	Add numbers together	
number3	Print total number	

B Solution algorithm



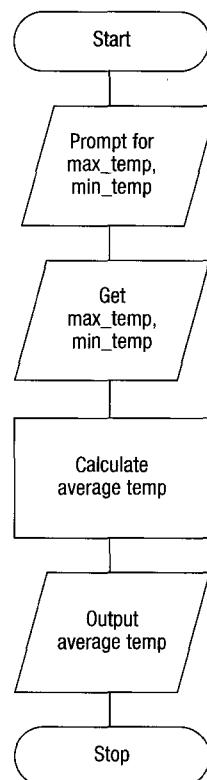
EXAMPLE 3.2 Find average temperature

A program is required to prompt the terminal operator for the maximum and minimum temperature readings on a particular day, accept those readings as integers, and calculate and display to the screen the average temperature.

A Defining diagram

Input	Processing	Output
max_temp	Prompt for temperatures	
min_temp	Get temperatures Calculate average temperature Display average temperature	avg_temp

B Solution algorithm



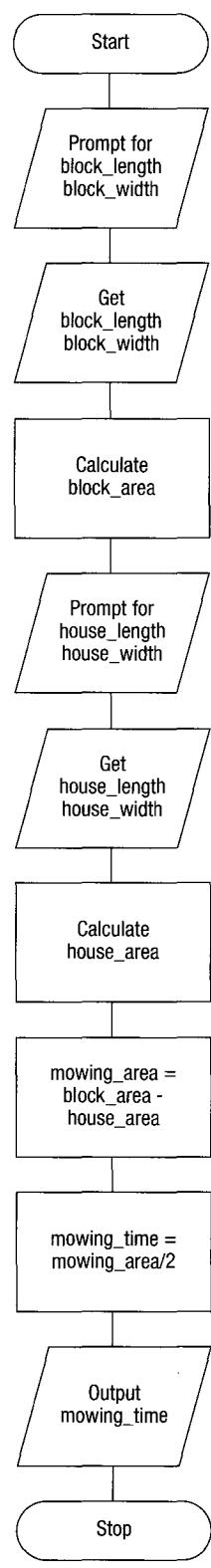
EXAMPLE 3.3 Compute mowing time

A program is required to read from the screen the length and width of a rectangular house block, and the length and width of the rectangular house that has been built on the block. The algorithm should then compute and display the mowing time required to cut the grass around the house, at the rate of two square metres per minute.

A Defining diagram

Input	Processing	Output
block_length	Prompt for block measurements	mowing_time
block_width	Get block measurements	
house_length	Prompt for house measurements	
house_width	Get house measurements Calculate mowing area Calculate mowing time	

B Solution algorithm

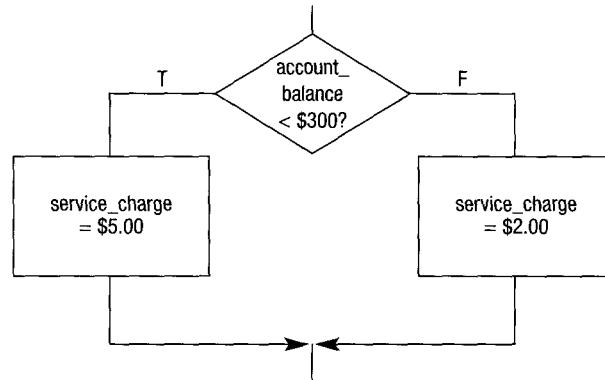


Flowcharts and the selection control structure

Each variation of the selection control structure developed in Chapter 4 in pseudocode can be represented by a flowchart.

Simple IF statement

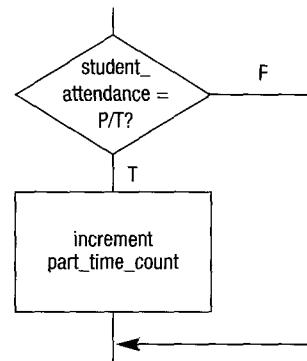
Simple selection occurs when a choice is made between two alternate paths, depending on the result of a condition being true or false. This structure is represented in a flowchart as follows:



Only one of the true or false paths will be followed, depending on the result of the condition in the decision symbol.

Null ELSE statement

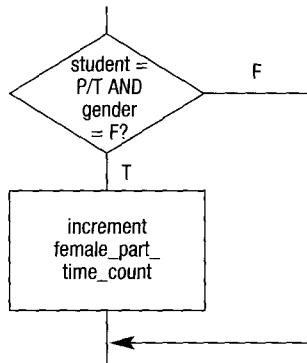
The null ELSE structure is a variation of the simple IF structure. It is used when a task is performed only when a particular condition is true. If the condition is false, no processing will take place, and the IF statement will be bypassed. For example:



In this case, the part_time_count field will only be altered if the true path is followed – that is, when the student's attendance pattern is part-time.

Combined IF statement

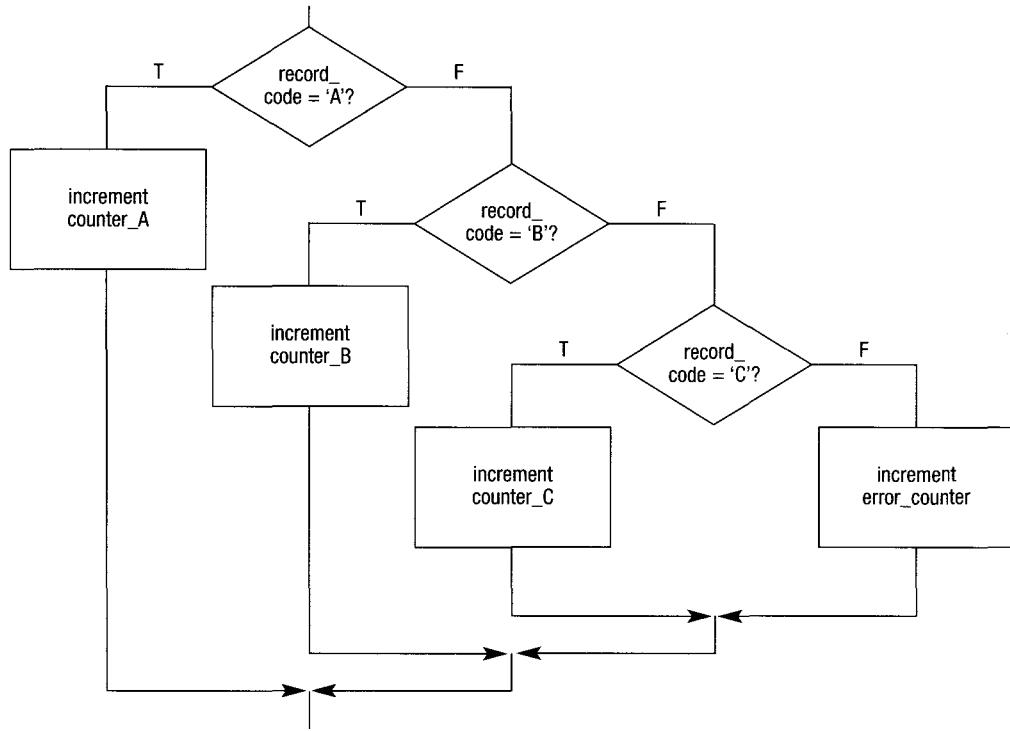
A combined IF statement is one that contains multiple conditions in the decision symbol, each connected with the logical operators AND or OR. If the connector AND is used to combine the conditions, then *both* conditions must be true for the combined condition to be true. For example:



In this case, each student record will undergo two tests. Only those students who are female, and whose attendance pattern is part-time, will be selected, and the variable `female_part_time_count` will be incremented. If either condition is found to be false, the counter will remain unchanged.

Nested IF statement

The nested IF statement is used when a field is being tested for various values, with a different action to be taken for each value. In a flowchart, this is represented by a series of decision symbols, as follows.



Simple algorithms that use the selection control structure

The following examples are the same as those represented by pseudocode in Chapter 4. In each example, the problem is defined and a solution algorithm developed using a flowchart. For ease in defining the problem, the processing verbs in each example have been underlined.

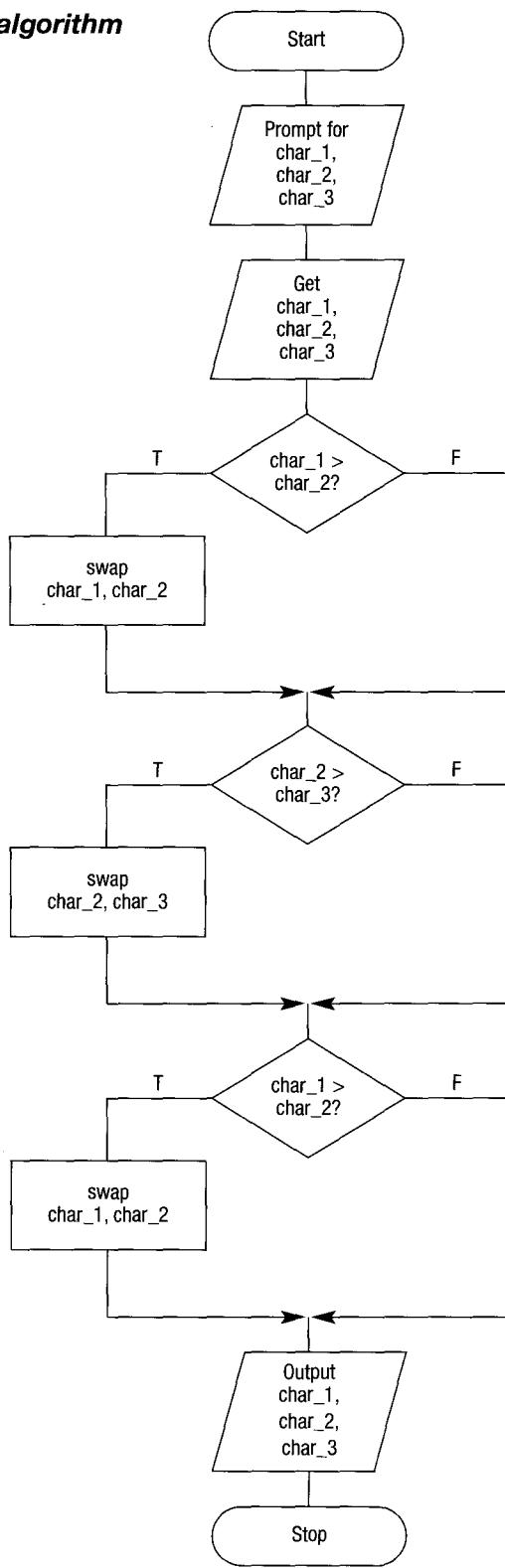
EXAMPLE 4.1 Read three characters

Design an algorithm that will prompt a terminal operator for three characters. ~~accept~~ those characters as input, sort them into ascending sequence and output them to ~~the~~ screen.

A Defining diagram

Input	Processing	Output
char_1	Prompt for characters	char_1
char_2	Accept three characters	char_2
char_3	Sort three characters Output three characters	char_3

B Solution algorithm



EXAMPLE 4.2 Process customer record

A program is required to read a customer's name, a purchase amount and a tax code. The tax code has been validated and will be one of the following:

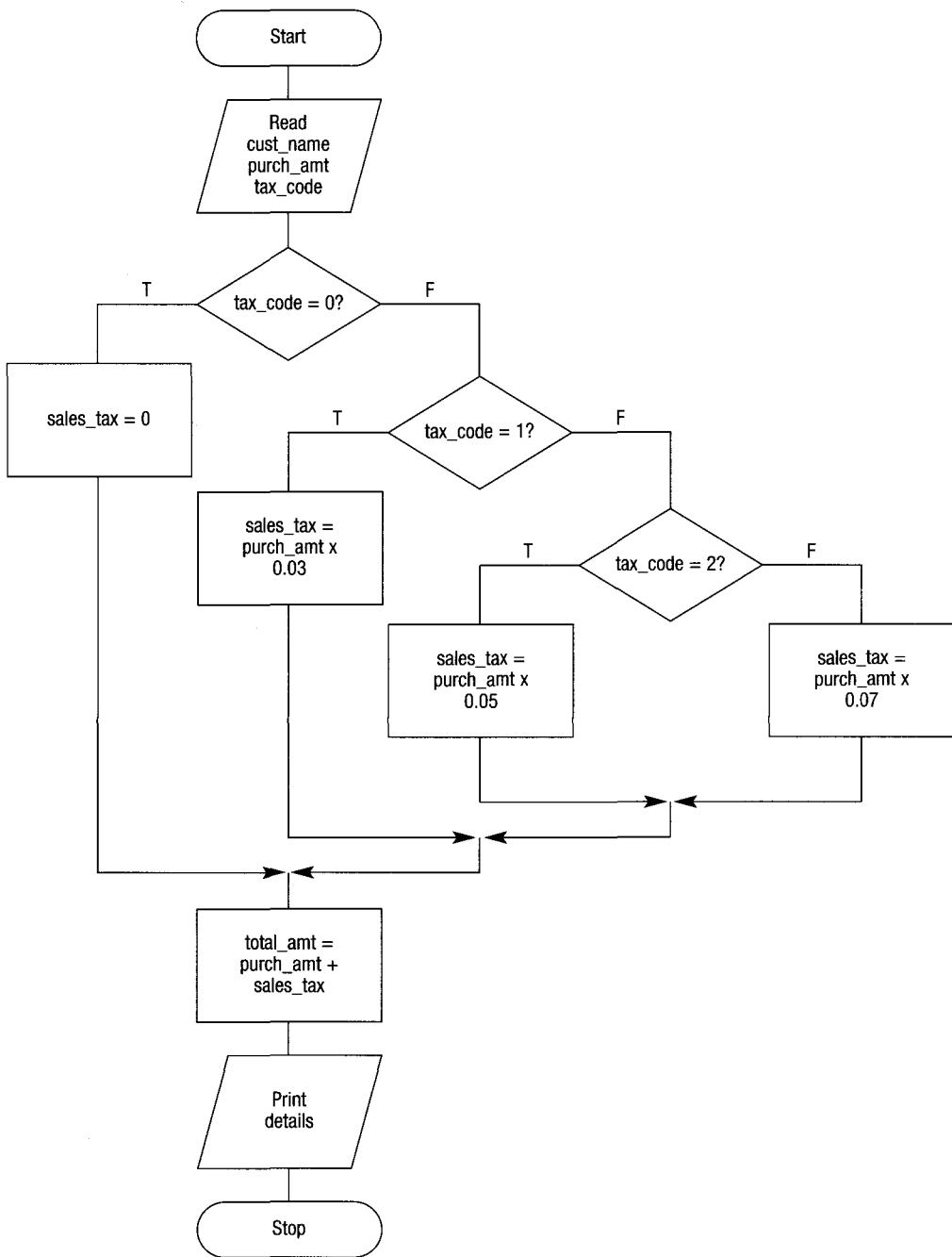
- 0 tax exempt (0%)
- 1 state sales tax only (3%)
- 2 federal and state sales tax (5%)
- 3 special sales tax (7%)

The program must then compute the sales tax and the total amount due, and print the customer's name, purchase amount, sales tax and total amount due.

A Defining diagram

Input	Processing	Output
cust_name	Read customer details	cust_name
purch_amt	Compute sales tax	purch_amt
tax_code	Compute total amount Print customer details	sales_tax total_amt

B Solution algorithm



EXAMPLE 4.3 Calculate employee's pay

A program is required by a company to read an employee's number, pay rate and the number of hours worked in a week. The program is then to validate the pay rate and the hours worked fields and, if valid, compute the employee's weekly pay and print it along with the input data.

Validation: According to the company's rules, the maximum hours an employee can work per week is 60 hours, and the maximum hourly rate is \$25.00 per hour. If the hours worked field or the hourly rate field is out of range, the input data and an appropriate message is to be printed and the employee's weekly pay is not to be calculated.

Weekly pay calculation: Weekly pay is calculated as hours worked times pay rate. If more than 35 hours are worked, payment for the overtime hours worked is calculated at time-and-a-half.

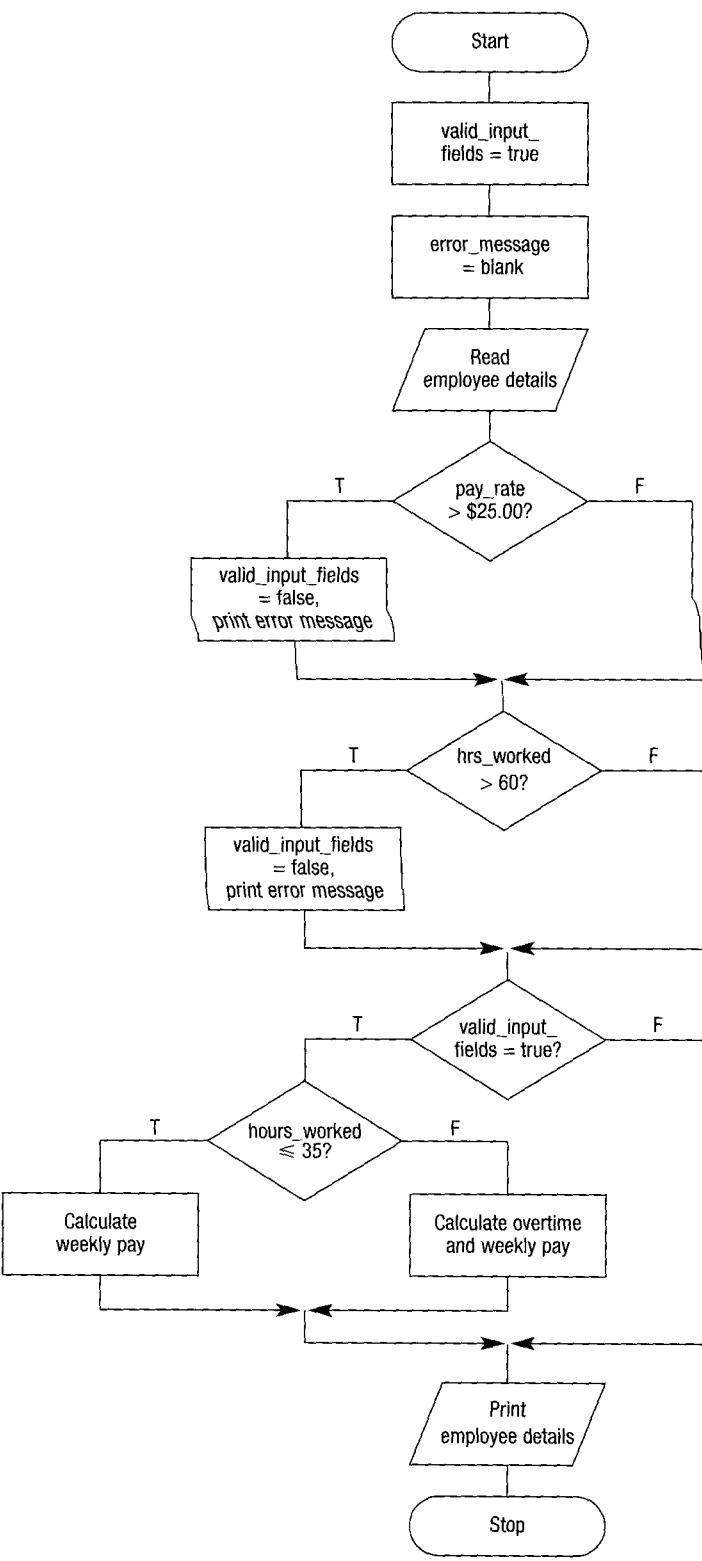
A Defining diagram

Input	Processing	Output
emp_no	Read employee details	emp_no
pay_rate	Validate input fields	pay_rate
hrs_worked	Calculate employee pay Print employee details	hrs_worked emp_weekly_pay error_message

B Solution algorithm

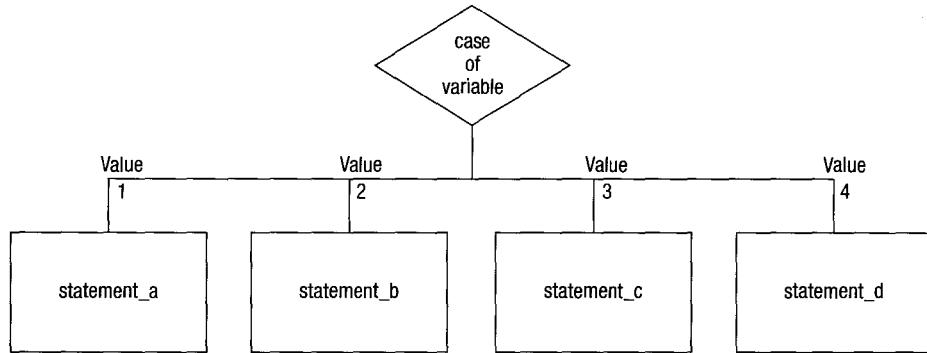
The solution to this problem will require a series of decision symbols. First, the variables pay_rate and hrs_worked must be validated, and if either is found to be out of range, an appropriate message should be printed.

The employee's weekly pay is only to be calculated if the variables pay_rate and hrs_worked are valid, so another variable valid_input_fields will be used to indicate to the program whether or not these input fields are valid.



The case structure expressed as a flowchart

The case control structure is another way of expressing a nested IF statement. It is not really an additional control structure; but one that extends the basic selection control structure to be a choice between multiple values. It is expressed in a flowchart by a decision symbol with a number of paths leading from it, depending on the value of the variable, as follows:



Let us now look again at Example 4.2. The solution algorithm for this example was earlier expressed as a nested IF statement. However, it could equally have been expressed as a CASE statement.

EXAMPLE 4.4 Process customer record

A program is required to read a customer's name, a purchase amount and a tax code. The tax code has been validated and will be one of the following:

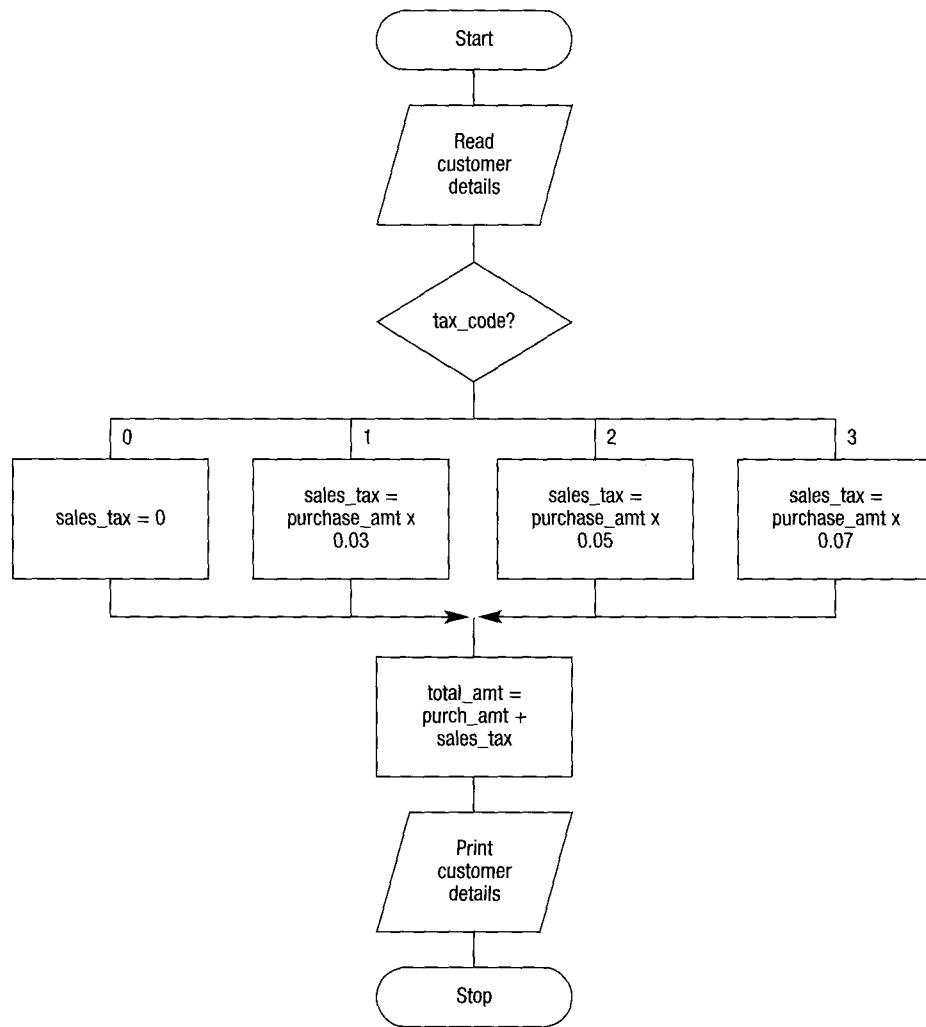
- 0 tax exempt (0%)
- 1 state sales tax only (3%)
- 2 federal and state sales tax (5%)
- 3 special sales tax (7%)

The program must then compute the sales tax and the total amount due, and print the customer's name, purchase amount, sales tax and total amount due.

A Defining diagram

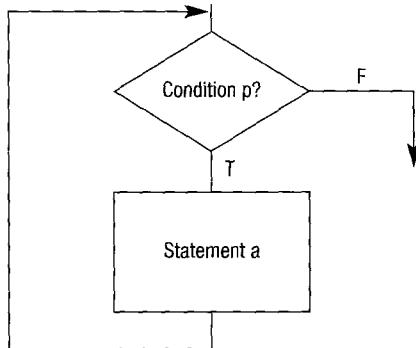
Input	Processing	Output
cust_name	Read customer details	cust_name
purch_amt	Compute sales tax	purch_amt
tax_code	Compute total amount Print customer details	sales_tax total_amt

B Solution algorithm



Flowcharts and the repetition control structure

In Chapter 5 the DOWHILE construct was introduced as the pseudocode representation of a repetition loop. This can be represented in a flowchart as follows:



As the DOWHILE loop is a leading decision loop, the following processing takes place:

- 1 The logical condition p is tested.
- 2 If condition p is found to be true, the statements that follow the true path will be executed once. Control then returns upwards to the retesting of condition p.
- 3 If condition p is still true, the statements that follow the true path will be executed again, and so the repetition process continues until the condition is found to be false.
- 4 If condition p is found to be false, control will follow the false path.

There are two important considerations about which a programmer must be aware before designing a DOWHILE loop.

First, the testing of the condition is at the beginning of the loop. This means that the programmer may need to perform some initial processing to set up the condition adequately before it can be tested.

Second, the only way to terminate the loop is to render the DOWHILE condition false. This means that the programmer must set up some process within the repeated processing symbols that will eventually change the condition so that the condition becomes false. Failure to do this results in an endless loop.

Simple algorithms that use the repetition control structure

The following examples are the same as those represented by pseudocode in Chapter 5. In each example, the problem is defined and a solution algorithm developed using a flowchart. For ease in defining the problem, the processing verbs in each example have been underlined.

EXAMPLE 5.1 Fahrenheit–Celsius conversion

Every day, a weather station receives 15 temperatures expressed in degrees Fahrenheit. A program is to be written that will accept each Fahrenheit temperature, convert it to Celsius and display the converted temperature to the screen. After 15 temperatures have been processed, the words 'All temperatures processed' are to be displayed on the screen.

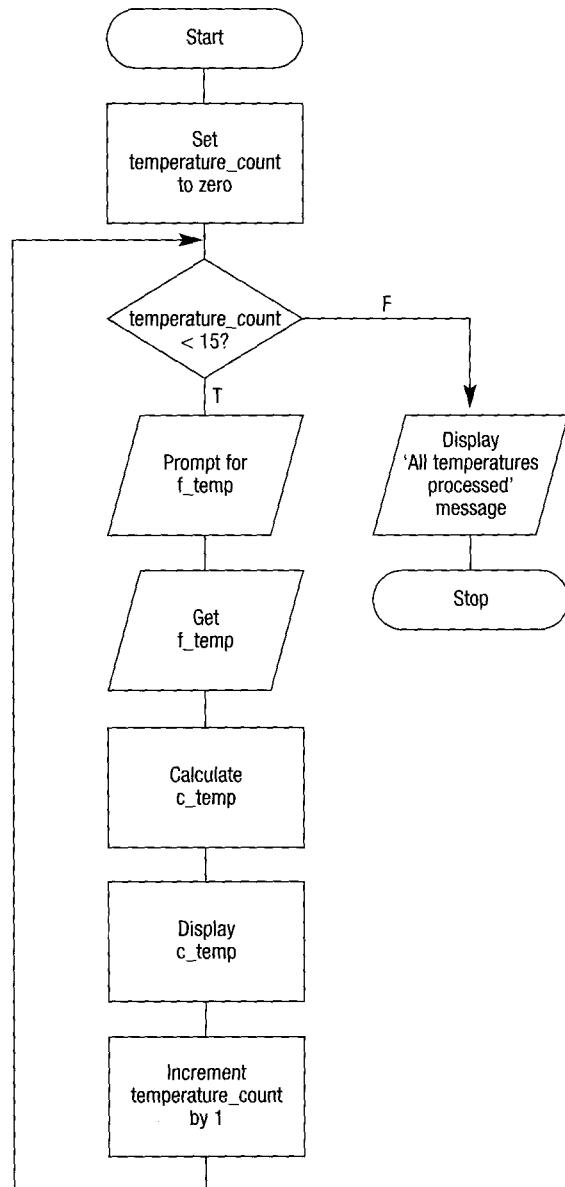
A Defining diagram

Input	Processing	Output
f_temp (15 temperatures)	Get Fahrenheit temperatures Convert temperatures Display Celsius temperatures Display screen message	c_temp (15 temperatures)

In this example, the programmer will need:

- a DOWHILE structure to repeat the necessary processing
- a counter, called temperature_count, initialised to zero, that will control the 15 repetitions.

B Solution algorithm



EXAMPLE 5.2 Print examination scores

A program is required to read and print a series of names and exam scores for students enrolled in a mathematics course. The class average is to be computed and printed at the end of the report. Scores can range from 0 to 100. The last record contains a blank name and a score of 999 and is not to be included in the calculations.

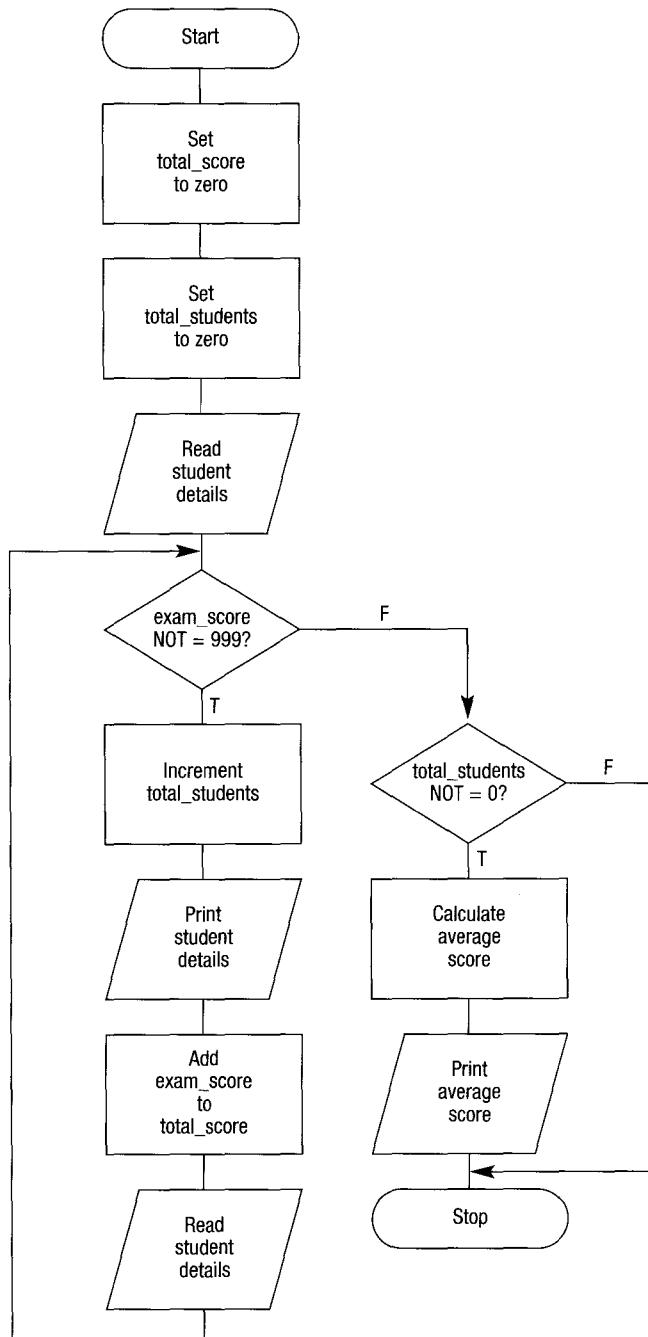
A Defining diagram

Input	Processing	Output
name	Read student details	name
exam_score	Print student details Compute average score	exam_score average_score
	Print average_score	

The programmer will need to consider the following requirements:

- a DOWHILE structure to control the reading of exam scores, until it reaches a score of 999
- an accumulator for total scores, namely total_score
- an accumulator for the total students, namely total_students.

B Solution algorithm



EXAMPLE 5.3 Process student enrolments

A program is required that will read a file of student records, and select and print only those students enrolled in a course unit named Programming I. Each student record contains student number, name, address, postcode, gender and course unit number. The course unit number for Programming I is 18500. Three totals are to be printed at the end of the report: total females enrolled in the course, total males enrolled in the course, and total students enrolled in the course.

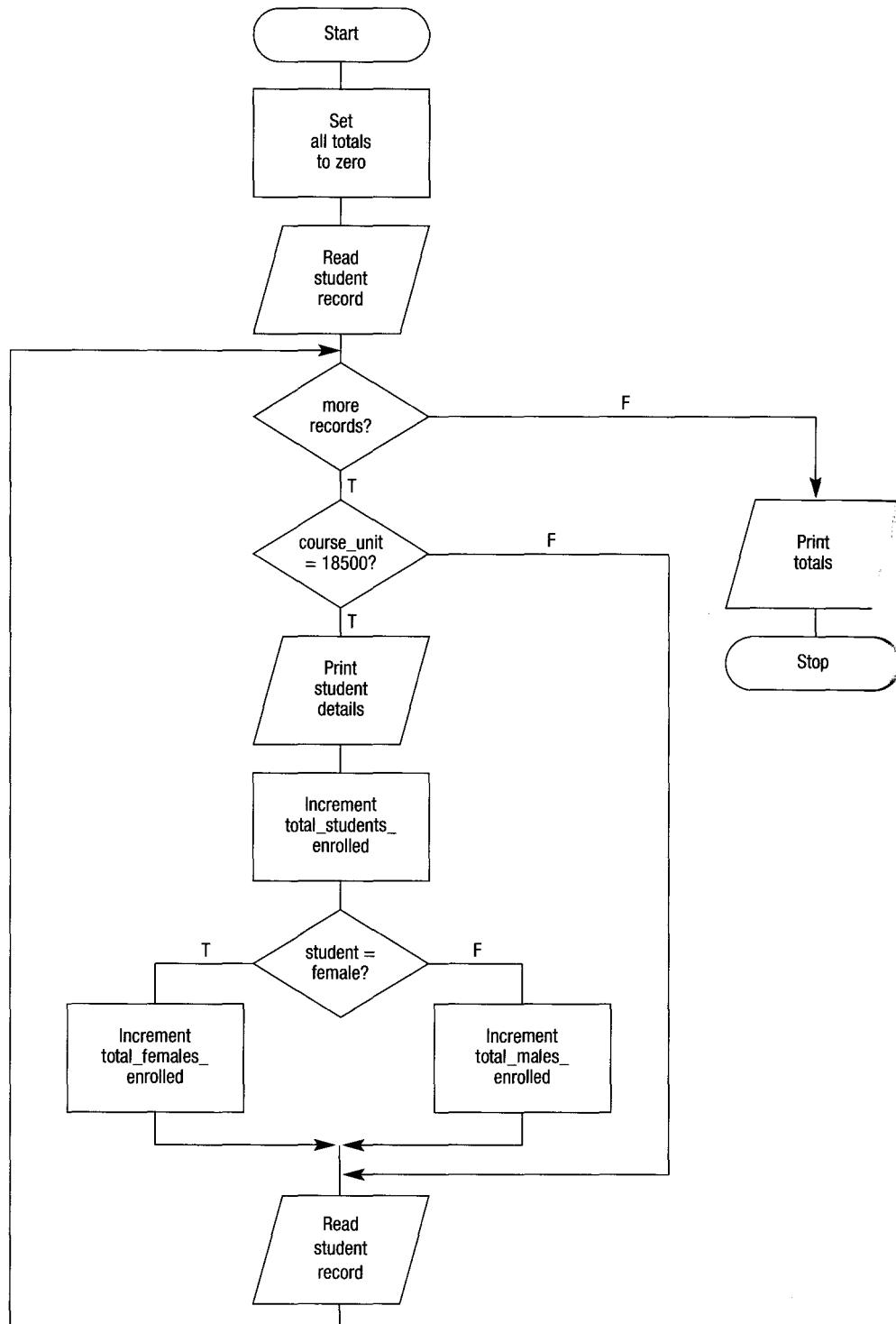
A Defining diagram

Input	Processing	Output
student_record • student_no • name • address • postcode • gender • course_unit	Read student records Select student records Print selected records Compute total females enrolled Compute total males enrolled Compute total students enrolled Print totals	selected student records total_females_enrolled total_males_enrolled total_students_enrolled

The programmer will need to consider the following requirements:

- a DOWHILE structure to perform the repetition
- decision symbols to select the required students
- accumulators for the three total fields.

B Solution algorithm



EXAMPLE 5.4 Process inventory items

A program is required to read a series of inventory records that contain item number, item description and stock figure. The last record in the file has an item number of zero. The program is to produce a low stock items report, by printing only those records that have a stock figure of less than 20 items. A heading is to be printed at the top of the report and a total low stock item count printed at the end.

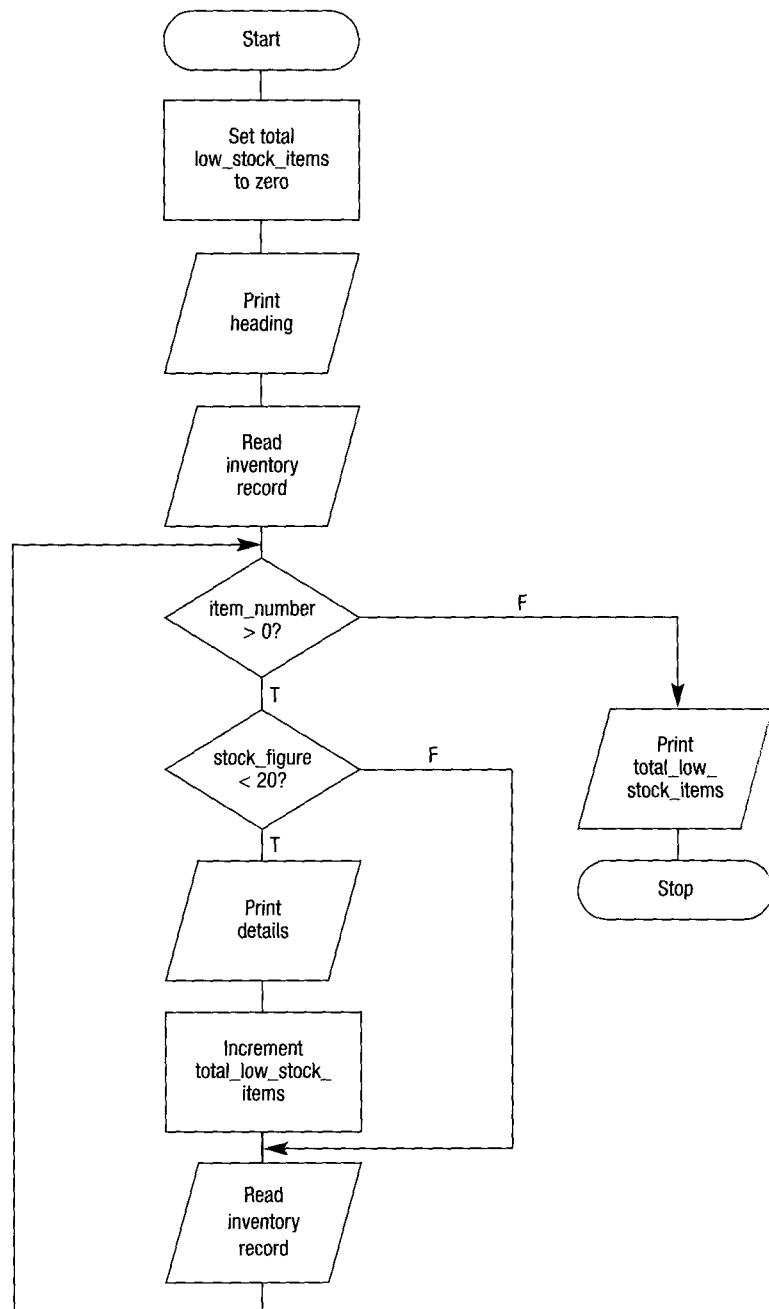
A Defining diagram

Input	Processing	Output
inventory record • item_number • item_description • stock_figure	Read inventory records Select low stock items Print low stock records Print total low stock items	heading selected records • item_number • item_description • stock_figure total_low_stock_items

The programmer will need to consider the following requirements:

- a DOWHILE structure to perform the repetition
- a decision symbol to select stock figures of less than 20
- an accumulator for total_low_stock_items.

B Solution algorithm



Further examples using flowcharts

The following examples are the same as those represented by pseudocode in Chapters 6 and 7. In each example, the problem is defined and a solution algorithm developed using a flowchart.

EXAMPLE 6.1 Process number pairs

Design an algorithm that will prompt for and receive pairs of numbers from an operator at a terminal and display their sum, product and average on the screen. If the calculated sum is over 200, an asterisk is to be displayed beside the sum. The program is to terminate when a pair of zero values is entered.

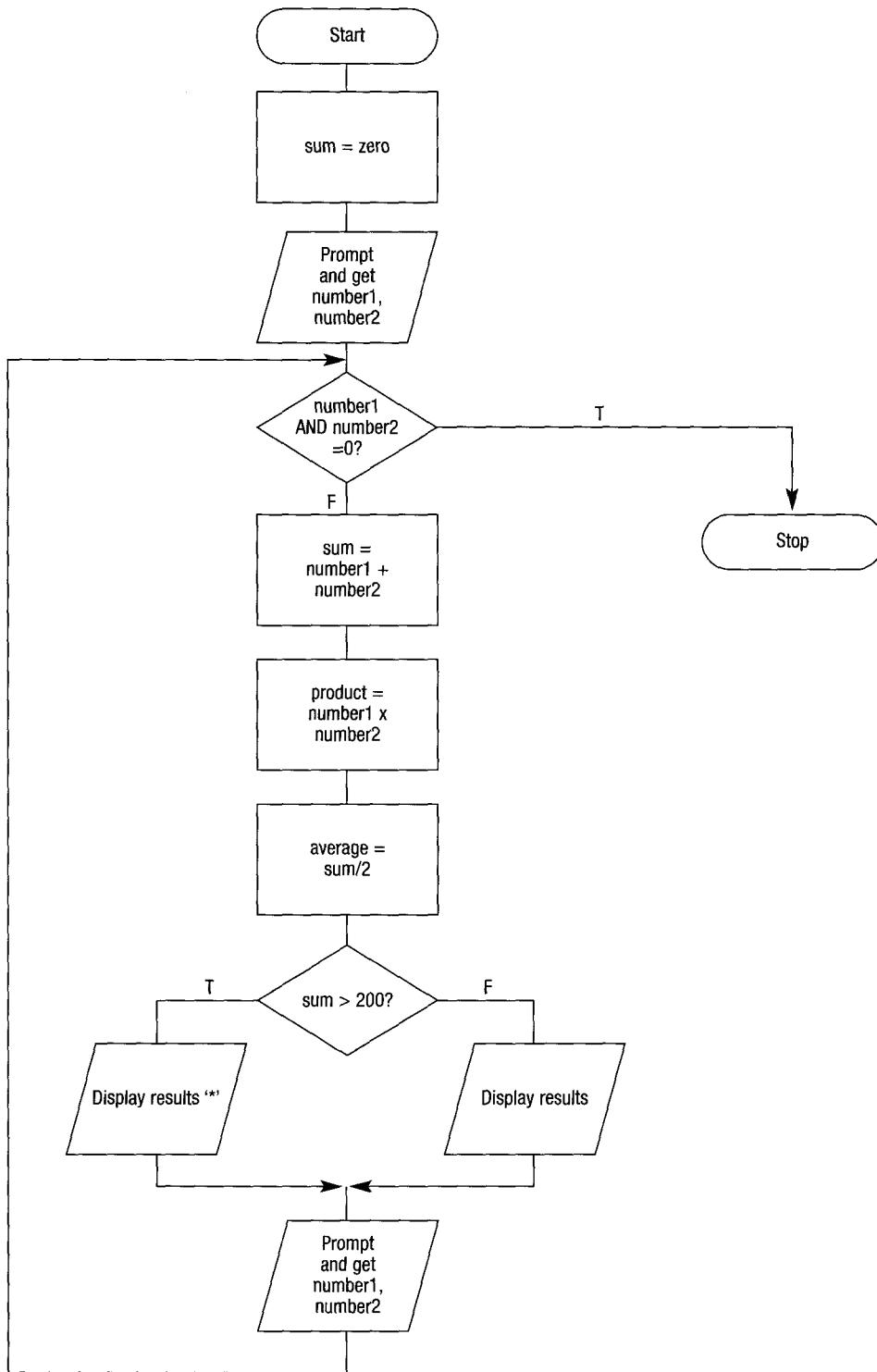
A Defining diagram

Input	Processing	Output
number1	Prompt for numbers Get numbers Calculate sum Calculate product Calculate average Display sum, product, average Display '*' if sum > 200	sum product average '*'
number2		

B Control structures required

- 1 A DOWHILE loop to control the repetition.
- 2 A decision symbol to determine if an asterisk is to be displayed.

C Solution algorithm



EXAMPLE 6.2 Print student records

A file of student records consists of 'S' records and 'U' records. An 'S' record contains the student's number, name, age, gender, address and attendance pattern; full-time (F/T) or part-time (P/T). A 'U' record contains the number and name of the unit or units in which the student has enrolled. There may be more than one 'U' record for each 'S' record. Design a solution algorithm that will read the file of student records and print only the student's number, name and address on a 'STUDENT LIST'.

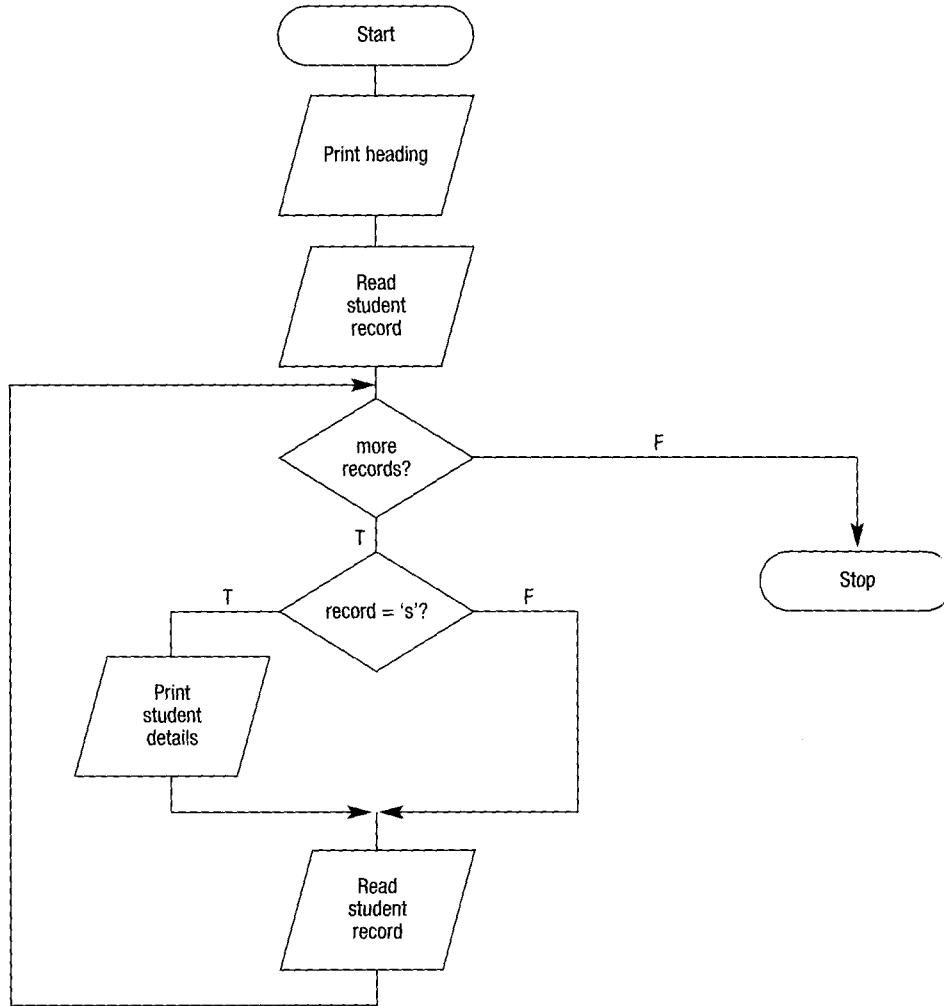
A Defining diagram

Input	Processing	Output
's' records <ul style="list-style-type: none">• number• name• address• age• gender• attendance_pattern 'u' records	<p>Print heading Read student records Select 's' records Print selected records</p>	<p>Heading line selected student records<ul style="list-style-type: none">• number• name• address</p>

B Control structures required

- 1 A DOWHILE loop to control the repetition.
- 2 A decision symbol to select 'S' records.

C Solution algorithm



EXAMPLE 6.3 Print selected students

Design a solution algorithm that will read the same student file as in Example 6.2, and produce a report of all female students who are enrolled part-time. The report is to be headed 'PART TIME FEMALE STUDENTS' and is to show the student's number, name, address and age.

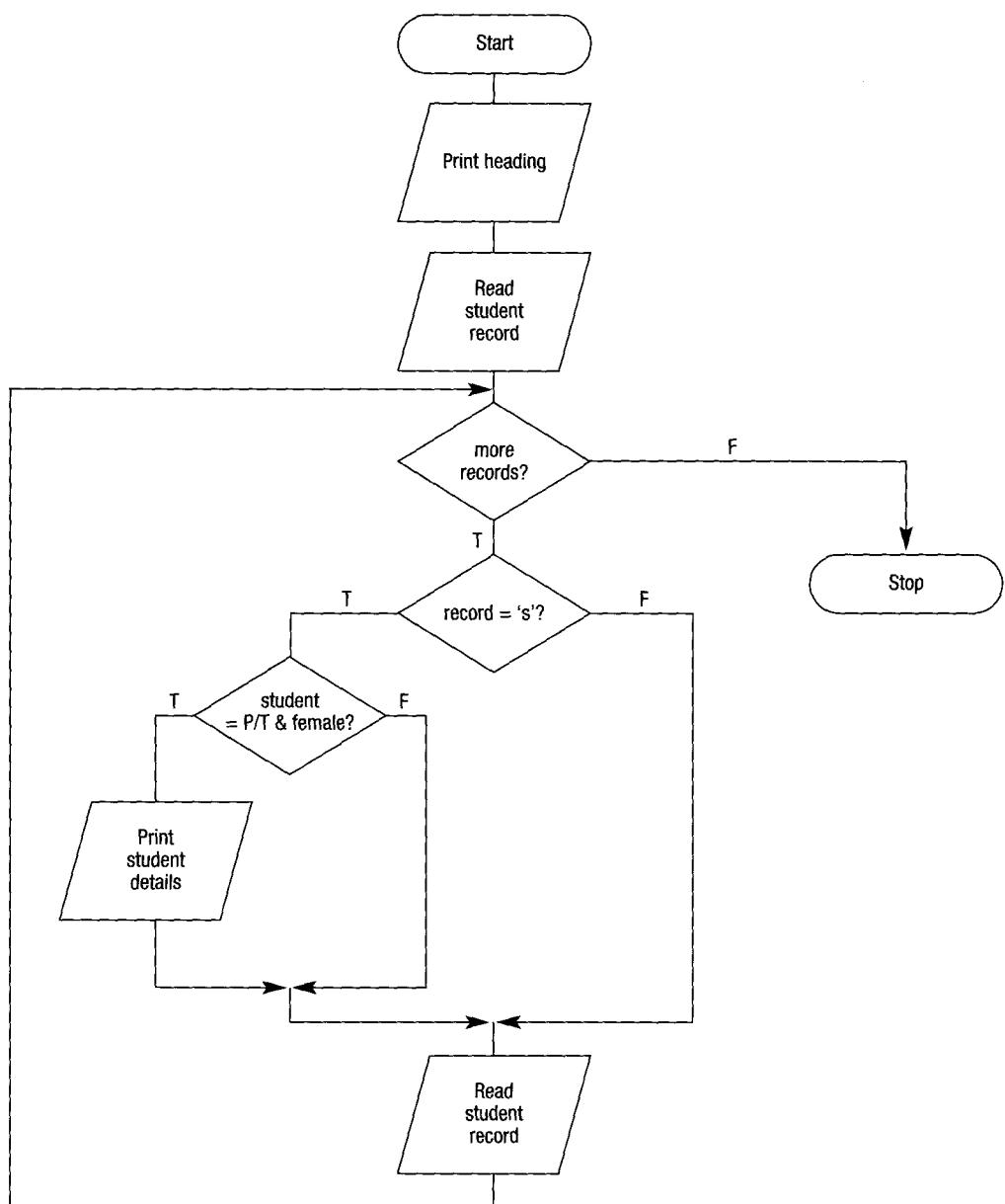
A Defining diagram

Input	Processing	Output
's' records • number • name • address • age • gender • attendance_pattern 'u' records	Print heading Read student records Select P/T female students Print selected records	Heading line selected student records • number • name • address • age

B Control structures required

- 1 A DOWHILE loop to control the repetition.
- 2 Decision symbols to select 'S', female and part-time (P/T) students.

C Solution algorithm



EXAMPLE 6.4 Print and total selected students

Design a solution algorithm that will read the same student file as in Example 6.3 and produce the same 'PART TIME FEMALE STUDENTS' report. In addition, you are to print at the end of the report the number of students who have been selected and listed, and the total number of students on the file.

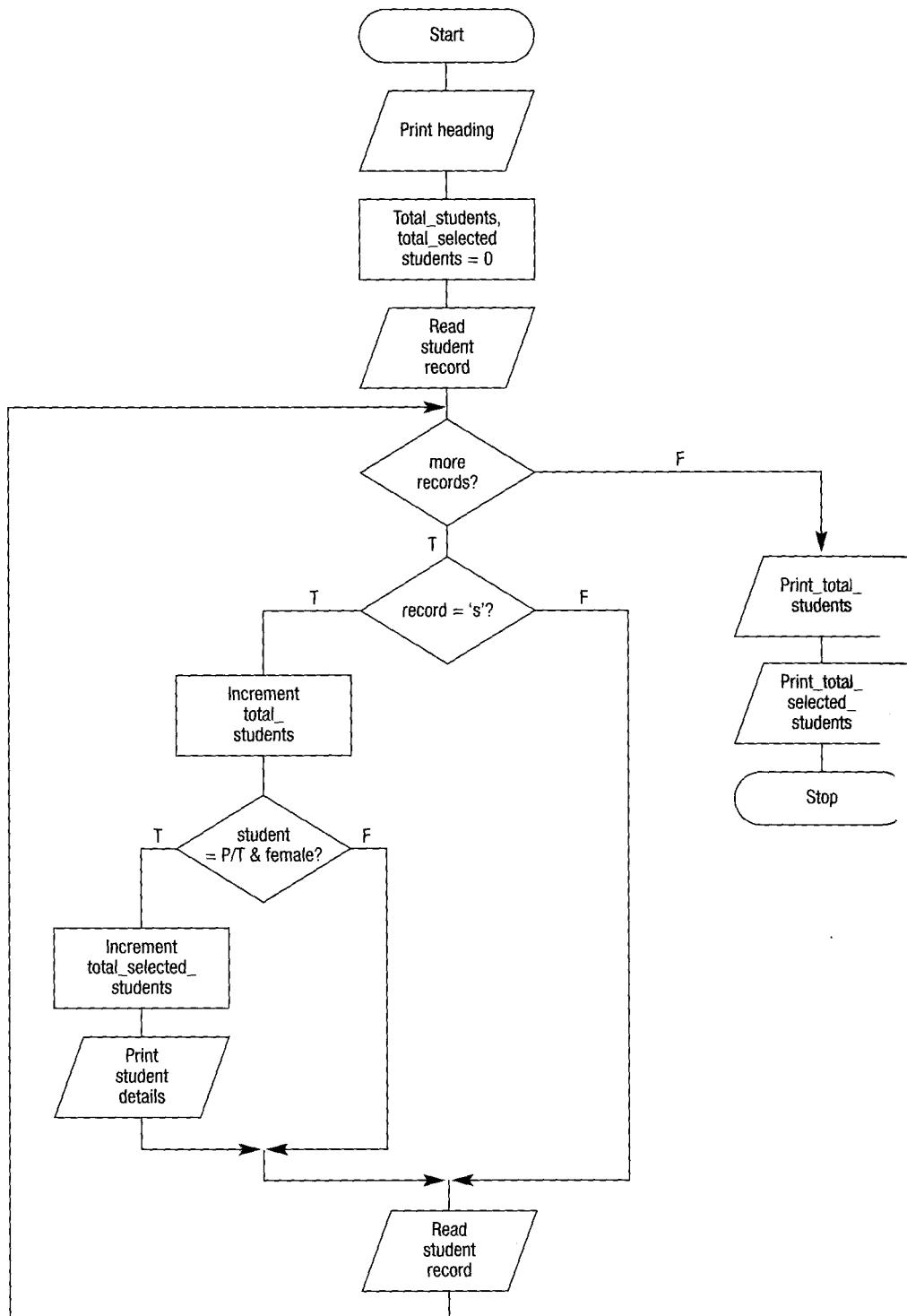
A Defining diagram

Input	Processing	Output
's' records • number • name • address • age • gender • attendance_pattern 'u' records	Print heading Read student records Select P/T female students Print selected records Compute total students Compute total selected students Print totals	Heading line selected student records • number • name • address • age total_students total_selected_students

B Control structures required

- 1 A DOWHILE loop to control the repetition.
- 2 Decision symbols to select 'S', female and P/T students.
- 3 Accumulators for total_selected_students and total_students.

C Solution algorithm



EXAMPLE 6.5 Print student report

Design an algorithm that will read the same student file as in Example 6.4 and, for each student, print the name, number and attendance pattern from the 'S' records (student records) and the unit number and unit name from the 'U' records (enrolled units records) as follows.

STUDENT REPORT

Student name
Student number
Attendance
Enrolled units
.....
.....

At the end of the report, print the total number of students enrolled.

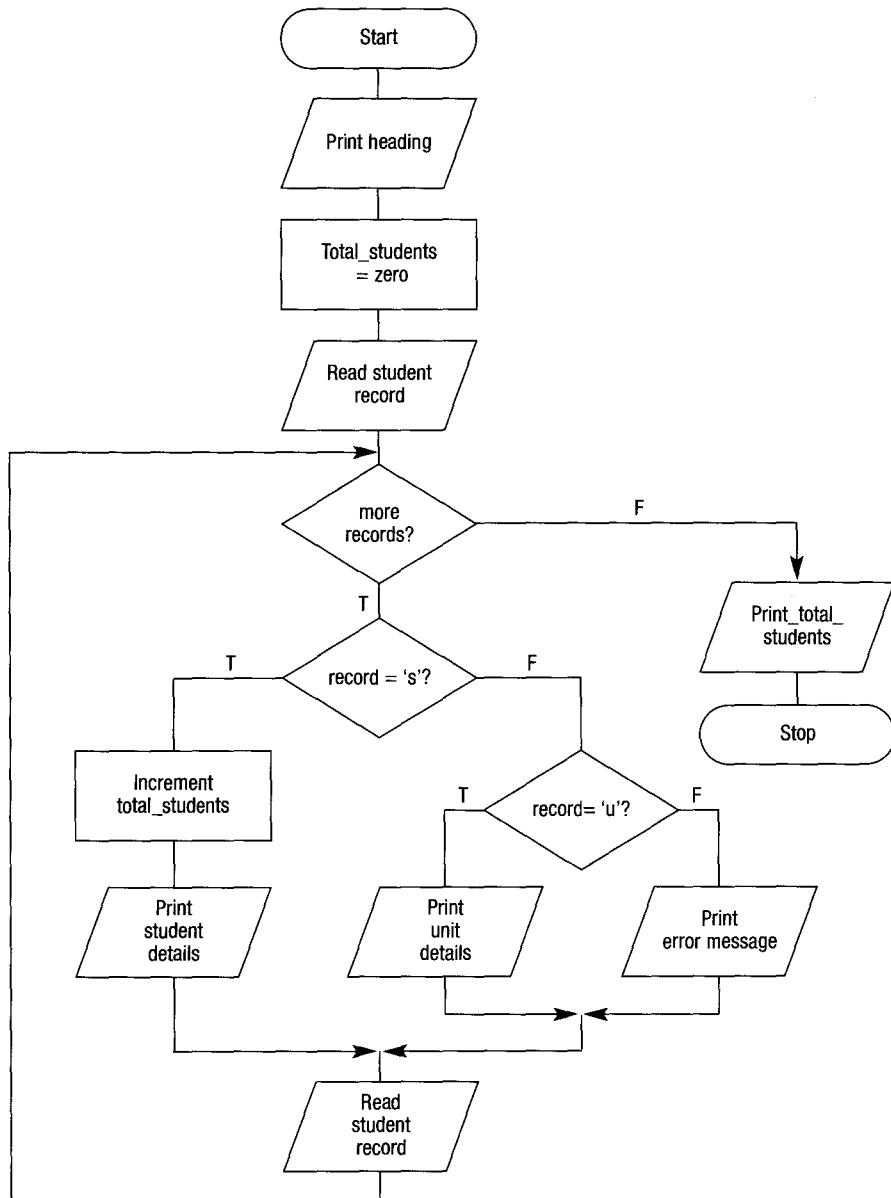
A Defining diagram

Input	Processing	Output
's' records • number • name • attendance_pattern	Print heading Read student records Print 's' record details	Heading line detail lines • name • number • attendance_pattern
'u' records • unit_number • unit_name	Print 'u' record details Compute total students Print total students	• unit_number • unit_name total_students

B Control structures required

- 1 A DOWHILE loop to control the repetition.
- 2 Decision symbols to select 'S' or 'U' records.
- 3 An accumulator for total_students.

C Solution algorithm



EXAMPLE 6.6 Produce sales report

Design a program that will read a file of sales records and produce a sales report. Each record in the file contains a customer's number, name, a sales amount and a tax code. The tax code is to be applied to the sales amount to determine the sales tax due for that sale, as follows:

Tax code	Sales tax
0	tax exempt
1	3%
2	5%

The report is to print a heading 'SALES REPORT', and detail lines listing the customer number, name, sales amount, sales tax and the total amount owing.

A Defining diagram

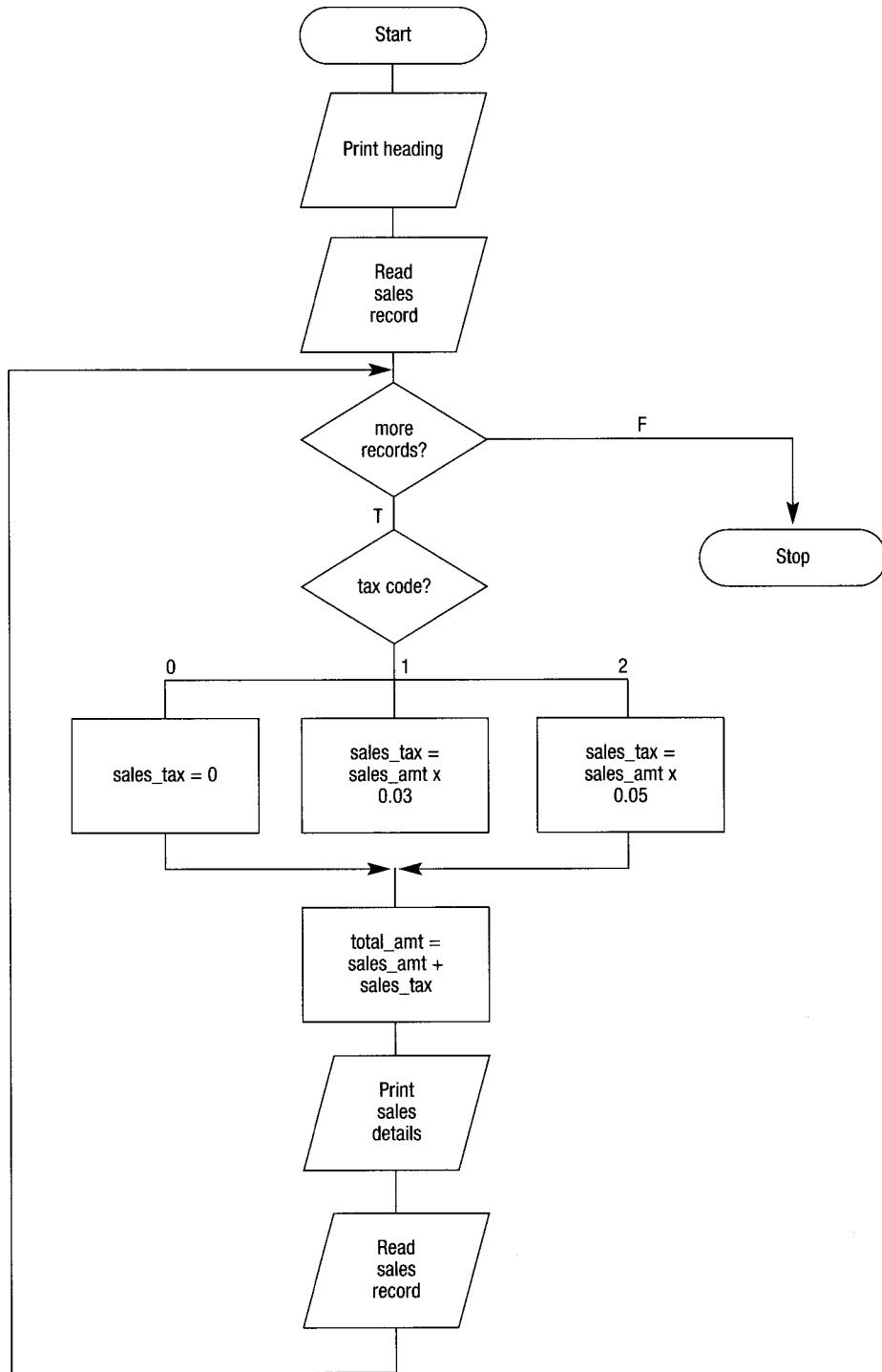
Input	Processing	Output
sales record • customer_number • name • sales_amt • tax_code	Print heading Read sales records Calculate sales tax Calculate total amount Print customer details	Heading line detail lines • customer_number • name • sales_amt • sales_tax • total_amount

B Control structures required

- 1 A DOWHILE loop to control the repetition.
- 2 A case structure to calculate the sales_tax.

Assume that the tax_code field has been validated and will contain only a value of 0, 1 or 2.

C Solution algorithm



EXAMPLE 6.7 Student test results

Design a solution algorithm that will read a file of student test results and produce a student test grades report. Each test record contains the student number, name and test score (out of 50). The program is to calculate for each student the test score as a percentage and to print the student's number, name, test score (out of 50) and letter grade on the report. The letter grade is determined as follows:

A = 90-100%

B = 80-89%

C = 70-79%

D = 60-69%

F = 0-59%

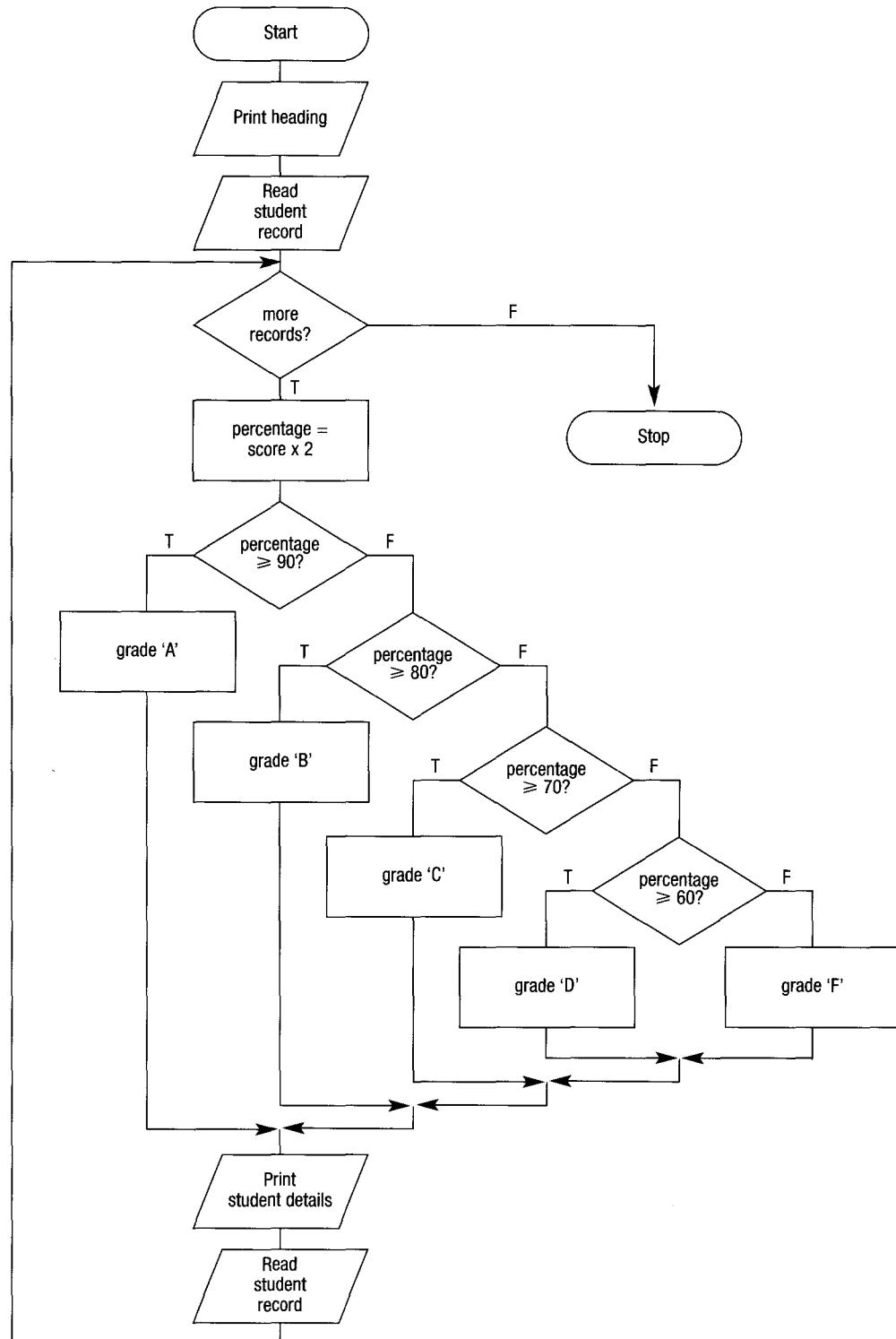
A Defining diagram

Input	Processing	Output
Student test records • student_number • name • test_score	Print heading Read student records Calculate test percentage Calculate letter grade Print student details	Heading line student details • student_number • name • test_score • grade

B Control structures required

- 1 A DOWHILE loop to control the repetition.
- 2 Decision symbols to calculate the grade.

C Solution algorithm



EXAMPLE 6.8 Gas supply billing

The Domestic Gas Supply Company records its customers' gas usage figures on a customer usage file. Each record on the file contains the customer number, customer name, customer address and gas usage expressed in cubic metres. Design a solution algorithm that will read the customer usage file, calculate the amount owing for gas usage for each customer, and print a report listing each customer's number, name, address, gas usage and the amount owing.

The company bills its customers according to the following rate: if the customer's usage is 60 cubic metres or less, a rate of \$2.00 per cubic metre is applied; if the customer's usage is more than 60 cubic metres, then a rate of \$1.75 per cubic metre is applied for the first 60 cubic metres and \$1.50 per cubic metre for the remaining usage.

At the end of the report, print the total number of customers and the total amount owing to the company.

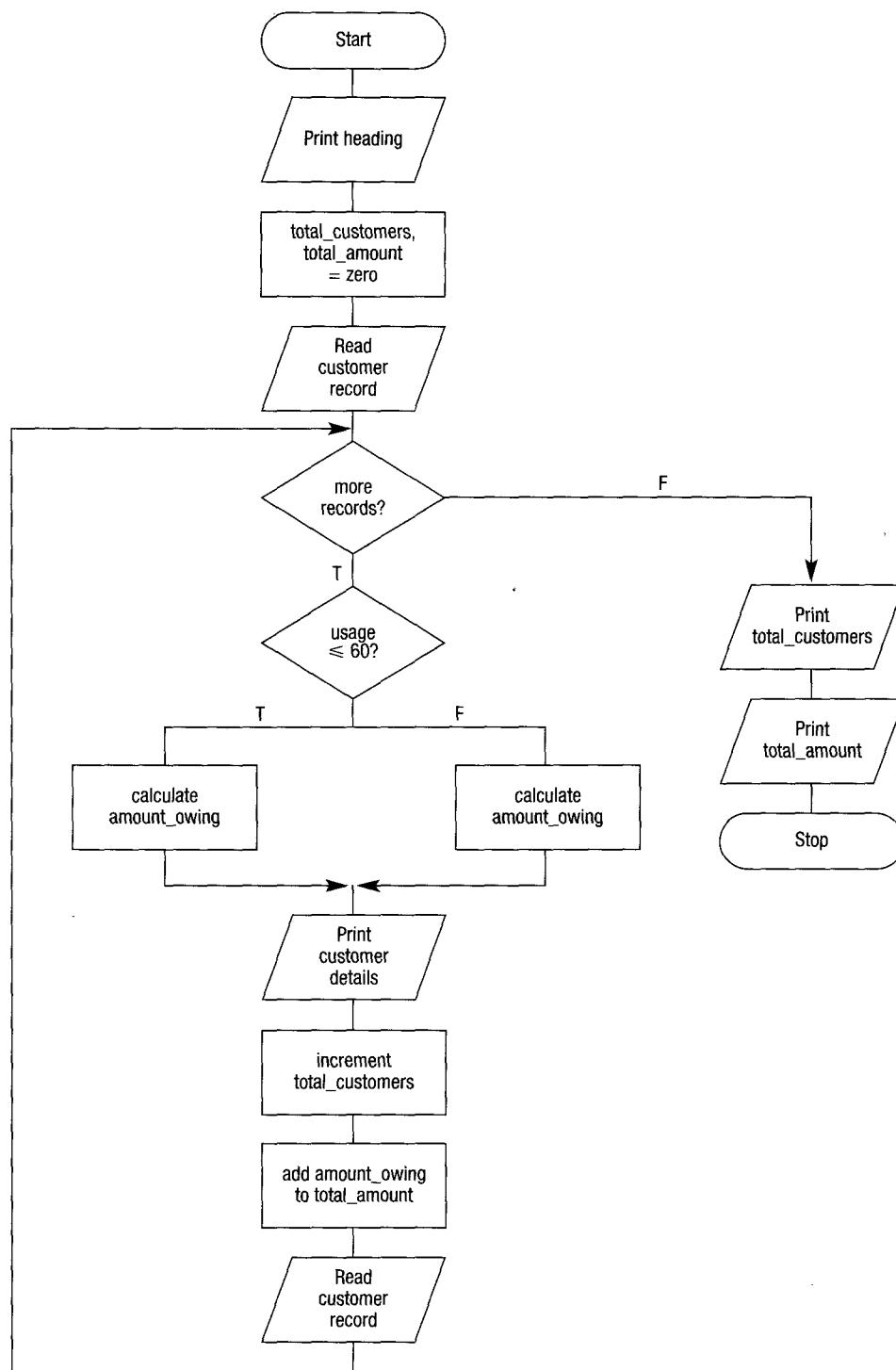
A Defining diagram

Input	Processing	Output
customer usage records <ul style="list-style-type: none">• customer_number• name• address• gas_usage	Print heading Read usage records Calculate amount owing Print customer details Compute total customers Compute total amount owing Print totals	Heading line customer details <ul style="list-style-type: none">• customer_number• name• address• gas_usage• amount_owing total_customers total_amount_owing

B Control structures required

- 1 A DOWHILE loop to control the repetition.
- 2 A decision symbol to calculate the amount_owing.
- 3 Accumulators for total_customers and total_amount_owing.

C Solution algorithm



EXAMPLE 7.6 Process exam scores

Design a program that will prompt for and receive 18 examination scores from a mathematics test, compute the class average, and display all the scores and the class average to the screen.

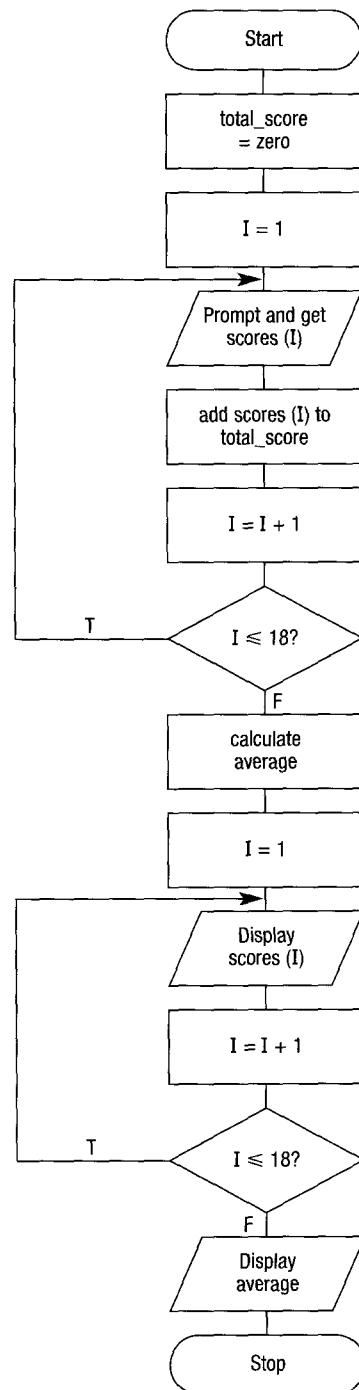
A *Defining diagram*

Input	Processing	Output
18 exam scores	Prompt for scores Get scores Compute class average Display scores Display class average	18 exam scores class_average

B *Control structures required*

- 1 An array to store the exam scores – that is, ‘scores’.
- 2 An index to identify each element in the array.
- 3 A DO loop to accept the scores.
- 4 Another DO loop to display the scores to the screen.

C Solution algorithm



EXAMPLE 7.7 Process integer array

Design an algorithm that will read an array of 100 integer values, calculate the average integer value, and count the number of integers in the array that are greater than the average integer value. The algorithm is to display the average integer value and the count of integers greater than the average.

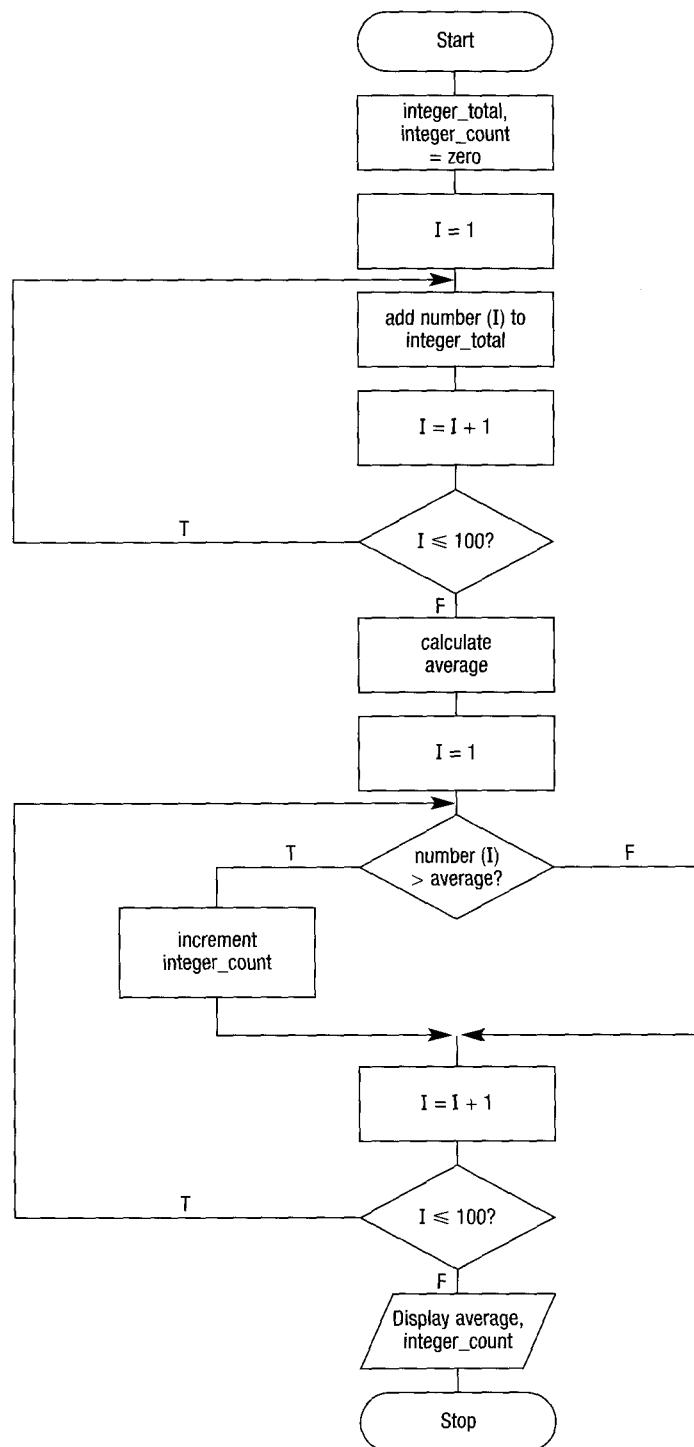
A Defining diagram

Input	Processing	Output
100 integer values	Read integer values Compute integer average Compute integer count Display integer average Display integer count	integer_average integer_count

B Control structures required

- 1 An array of integer values – that is, numbers.
- 2 A DO loop to calculate the average of the integers.
- 3 A DO loop to count the number of integers greater than the average.

C Solution algorithm



EXAMPLE 7.8 Validate sales number

Design an algorithm that will read a file of sales transactions and validate the sales numbers on each record. As each sales record is read, the sales number on the record is to be verified against an array of 35 sales numbers. Any sales number not found in the array is to be flagged as an error.

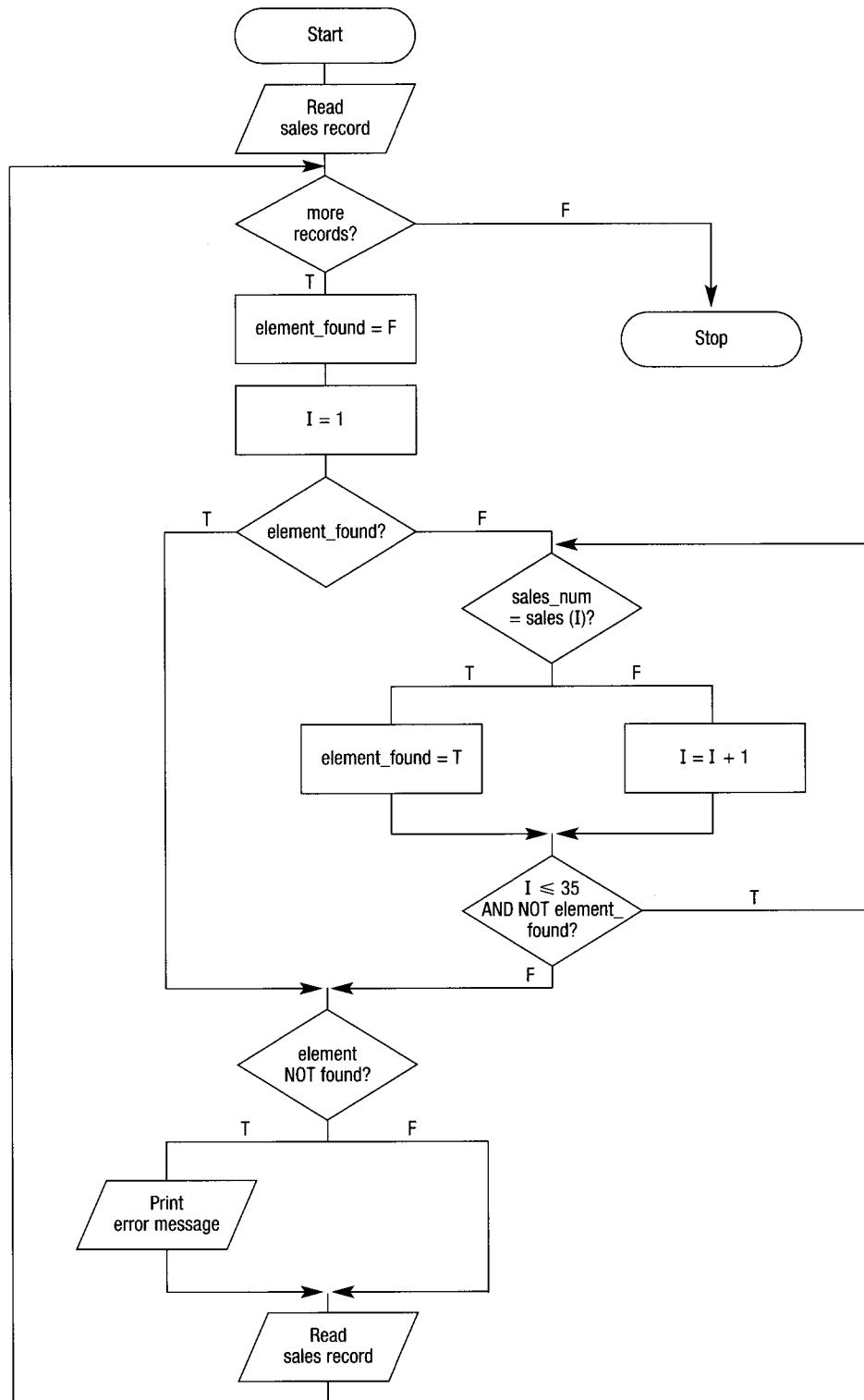
A Defining diagram

Input	Processing	Output
sales records • sales_number	Read sales records Validate sales numbers Print error message	error_message

B Control structures required

- 1 A previously initialised array of sales numbers – that is, sales_numbers.
- 2 A DOWHILE loop to read the sales file.
- 3 A DOWHILE loop to perform a linear search of the array for the sales number.
- 4 A variable element_found that will stop the search when the sales number is found.

C Solution algorithm



EXAMPLE 7.9 Calculate freight charge

Design an algorithm that will read an input weight for an item to be shipped, search an array of shipping weights and retrieve a corresponding freight charge. In this algorithm, two paired arrays, each containing six elements, have been established and initialised. The array, `shipping_weights`, contains a range of shipping weights in grams, and the array, `freight_charges`, contains a corresponding array of freight charges in dollars, as follows.

Shipping weights (grams)	Freight charges
1-100	3.00
101-500	5.00
501-1000	7.50
1001-3000	12.00
3001-5000	16.00
5001-9999	35.00

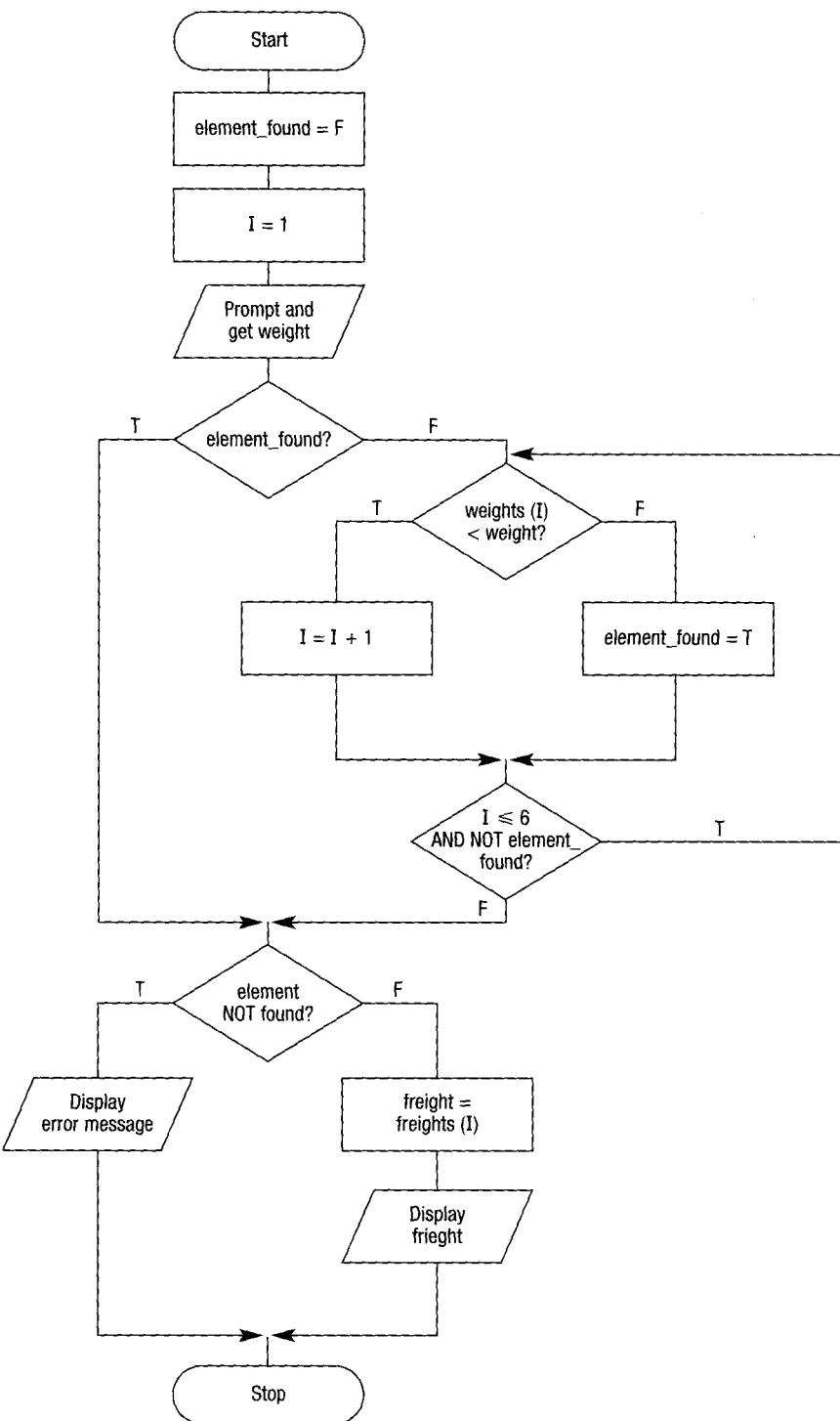
A Defining diagram

Input	Processing	Output
entry weight	Prompt for entry weight Get entry weight Search shipping weights array Compute freight charges Display freight charge	freight_charge error_message

B Control structures required

- 1 Two arrays, `shipping_weights` and `freight_charges`, already initialised.
- 2 A DOWHILE loop to search the `shipping_weights` array and hence retrieve the freight charge.
- 3 A variable `element_found` that will stop the search when the entry weight is found.

C Solution algorithm



Flowcharts and modules

When designing a modular solution to a problem, using a flowchart, the predefined process symbol is used to designate a process or module. This keeps flowcharts simple, because, as in pseudocode, the main flowchart contains the name of the process, or module, and each process has its own separate flowchart.

Let's look at the examples from Chapter 8, to see how flowcharts that contain modules are drawn.

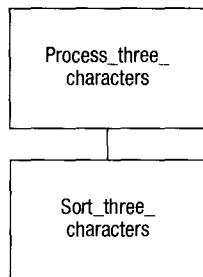
EXAMPLE 8.1 Read three characters

Design a solution algorithm that will prompt a terminal operator for three characters, accept those characters as input, sort them into ascending sequence and output them to the screen. The algorithm is to continue to read characters until 'XXX' is entered.

A Defining diagram

Input	Processing	Output
char_1	Prompt for characters	char_1
char_2	Accept three characters	char_2
char_3	Sort three characters Output three characters	char_3

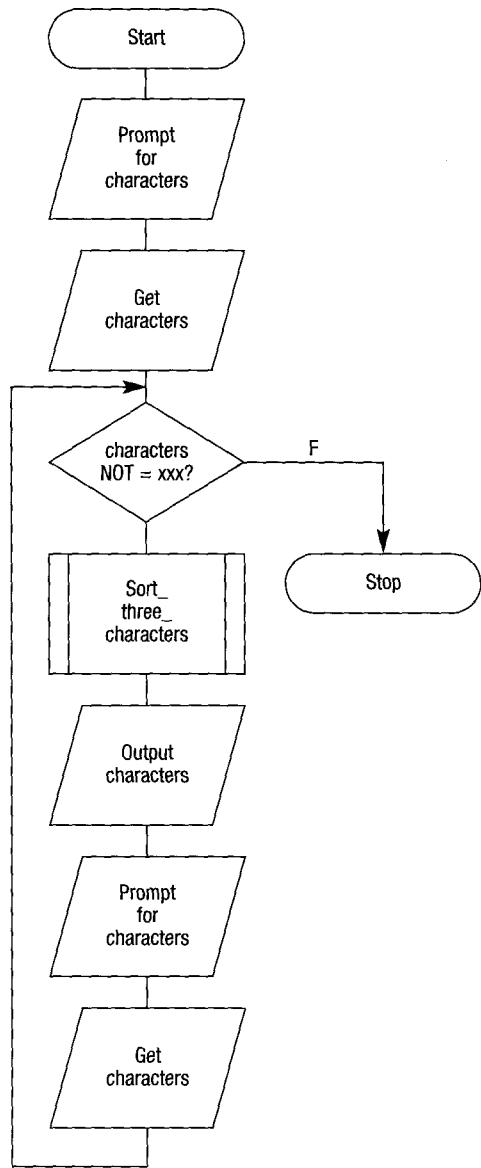
B Hierarchy chart



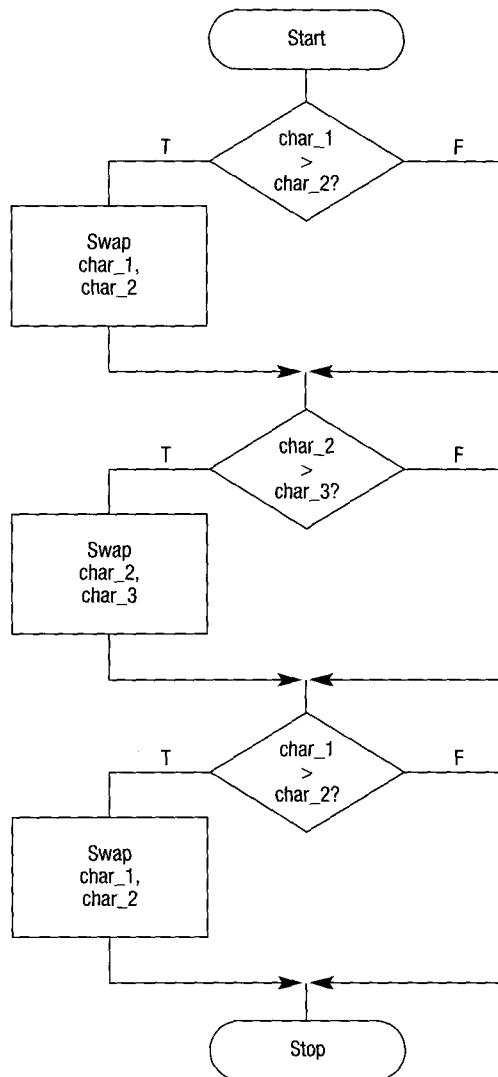
C Solution algorithm using a predefined process symbol

The flowchart solution consists of two flowcharts: a main flowchart called Process_three_characters and a process flowchart called Sort_three_characters. When the main flowchart wants to pass control to its process module, it simply names that process in a predefined process symbol. Control then passes to the process flowchart, and when the processing in that flowchart is complete, the module will pass control back to the main flowchart. The solution flowchart is simple and easy to read.

Process_three_characters



Sort_three_characters



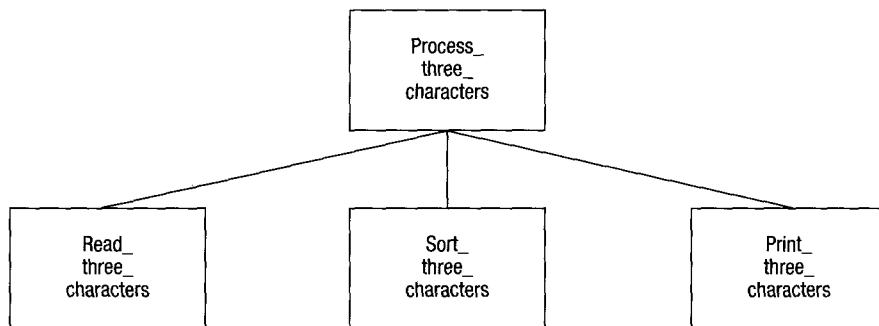
EXAMPLE 8.2 Process three characters

Design a solution algorithm that will prompt a terminal operator for three characters, accept those characters as input, sort them into ascending sequence and output them to the screen. The algorithm is to continue to read characters until 'XXX' is entered.

A Defining diagram

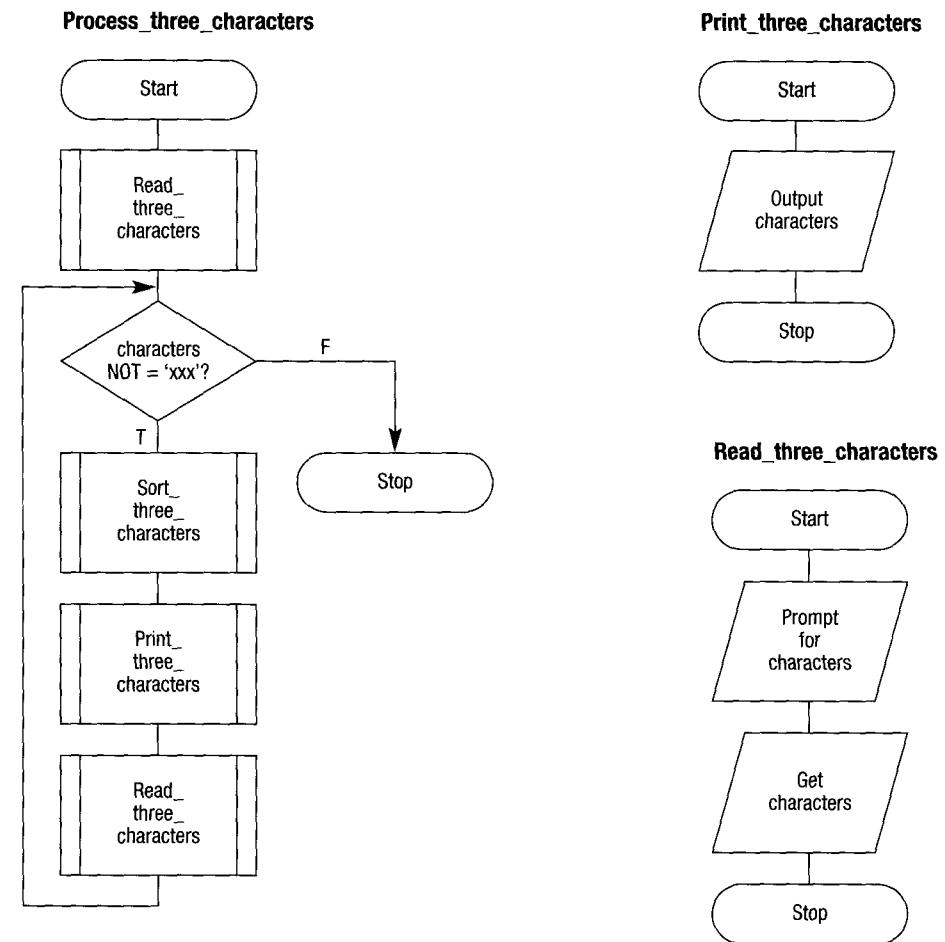
Input	Processing	Output
char_1	Prompt for characters	char_1
char_2	Accept three characters	char_2
char_3	Sort three characters Output three characters	char_3

B Hierarchy chart



C Solution algorithm

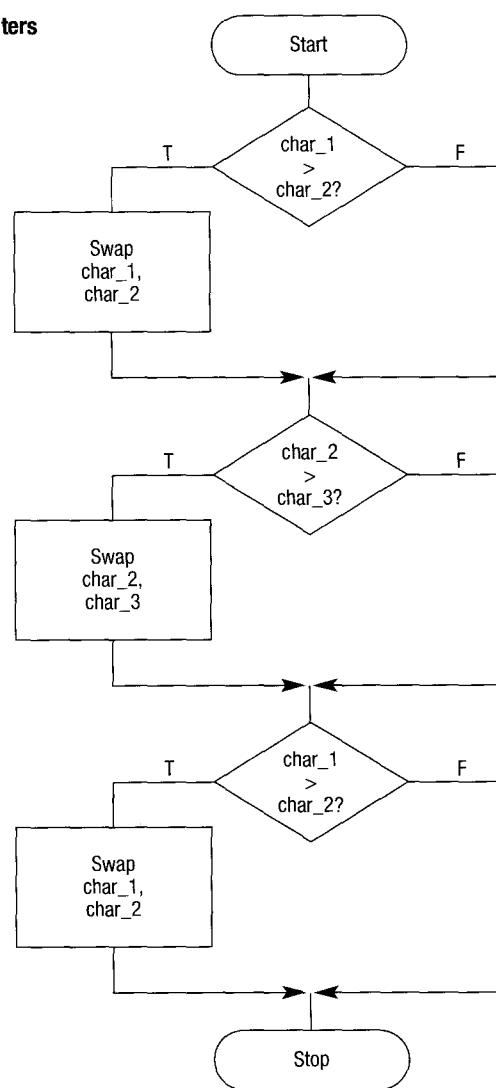
There are four modules in this algorithm, a mainline module and three subordinate modules, which will be represented by a flowchart, as follows:



(Example 8.2 continued next page)

(Example 8.2 continued from previous page)

Sort_three_characters



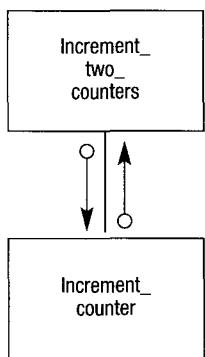
EXAMPLE 8.3 Increment two counters

Design an algorithm that will increment two counters from 1 to 10 and then output those counters to the screen. Your program is to use a module to increment the counters.

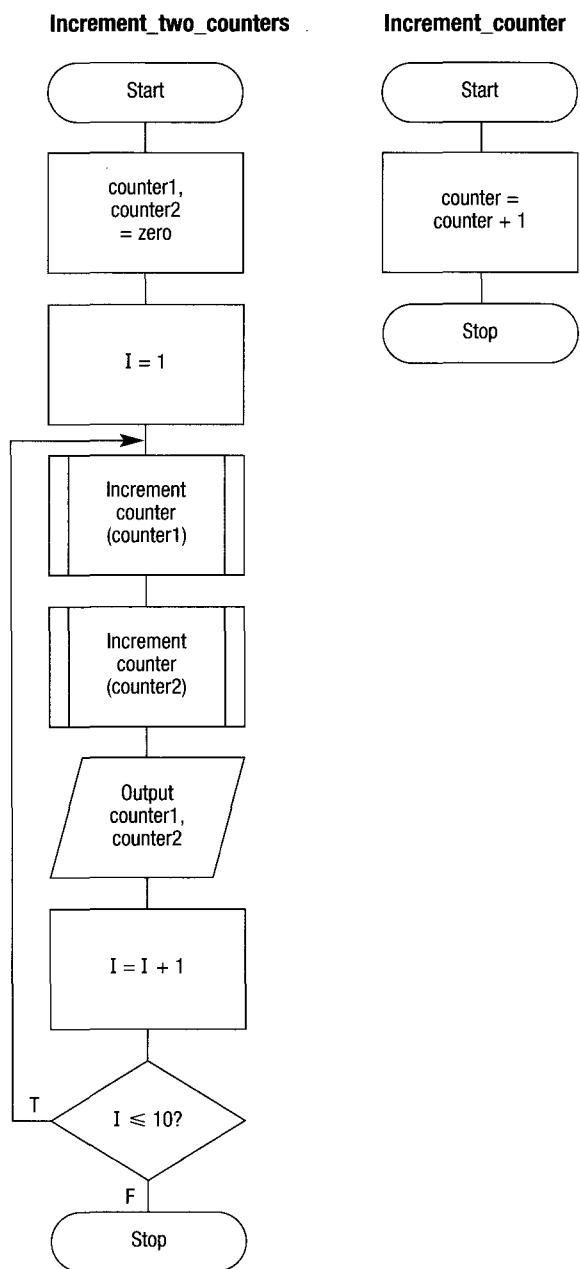
A Defining diagram

Input	Processing	Output
counter1	Increment counters	counter1
counter2	Output counters	counter2

B Hierarchy chart



C Solution algorithm



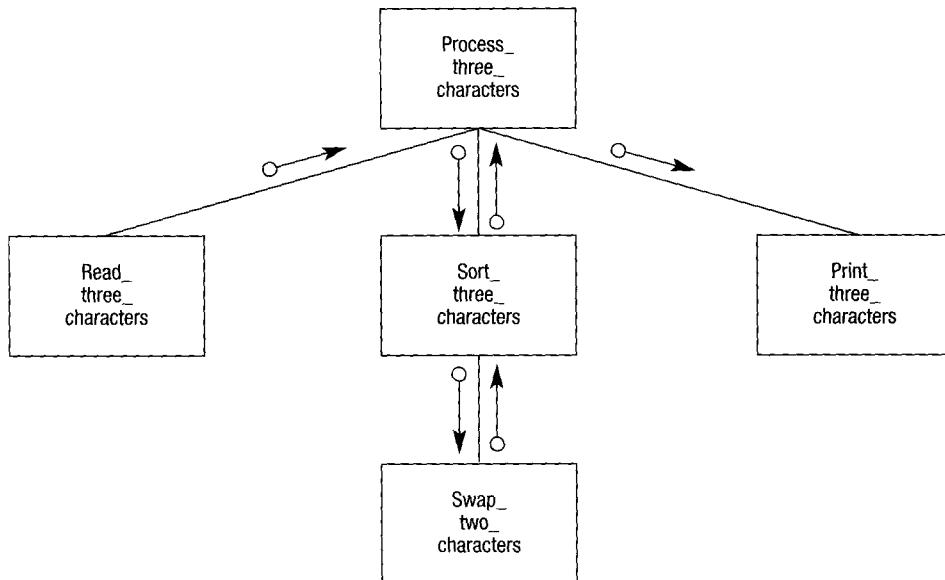
EXAMPLE 8.4 Process three characters

Design a solution algorithm that will prompt a terminal operator for three characters, accept those characters as input, sort them into ascending sequence and output them to the screen. The algorithm is to continue to read characters until 'XXX' is entered.

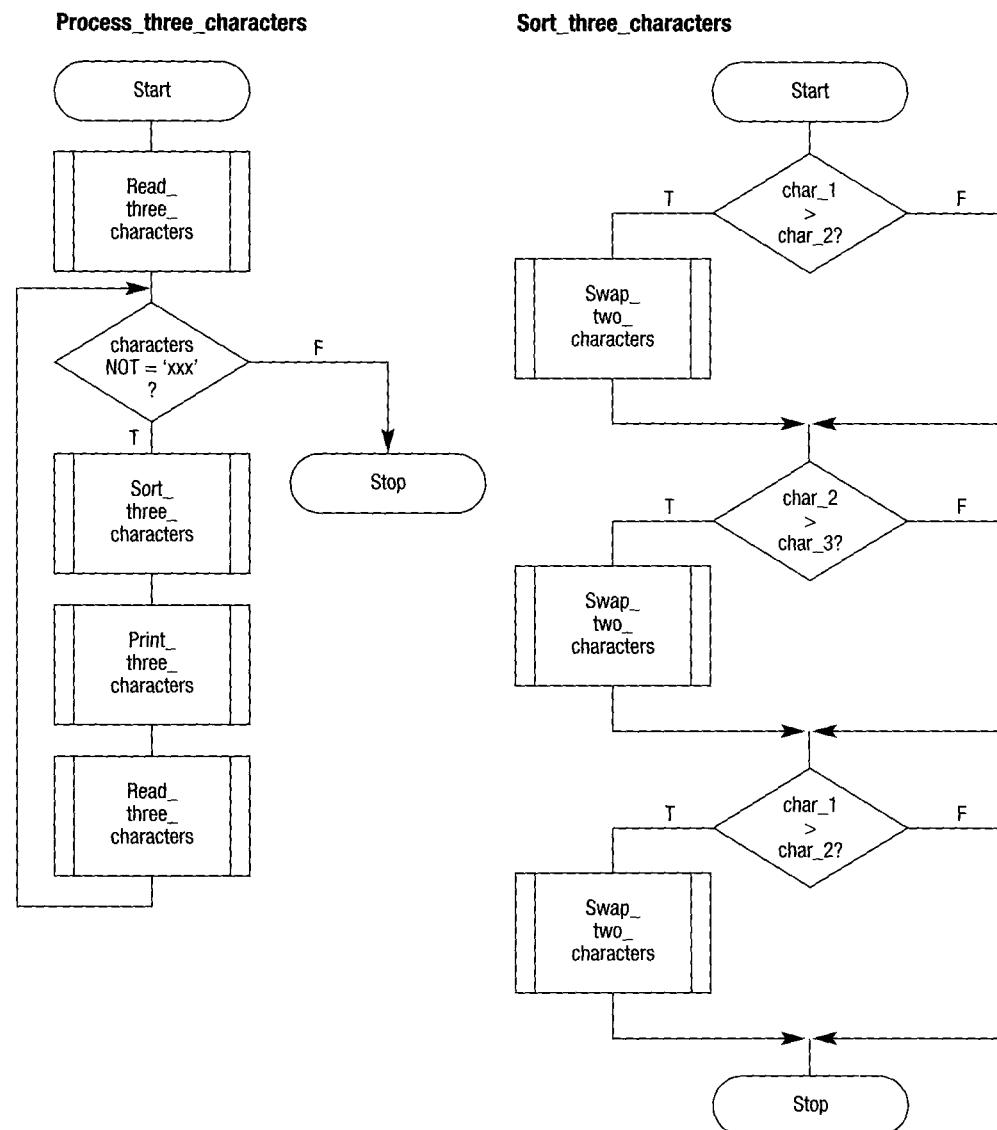
A Defining diagram

Input Processing Output		
char_1	Prompt for characters	char_1
char_2	Accept three characters	char_2
char_3	Sort three characters	char_3
	Output three characters	

B Construct a hierarchy chart



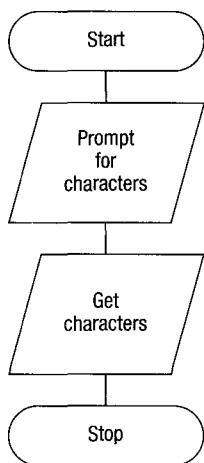
D Solution algorithm



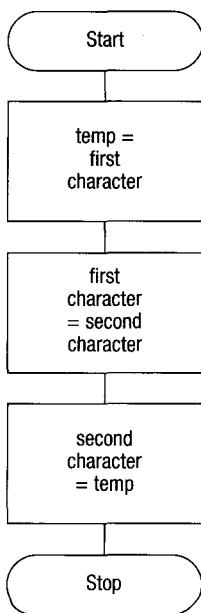
(Example 8.4 continued next page)

(Example 8.4 continued from previous page)

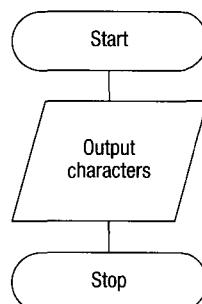
Read_three_characters



Swap_two_characters



Print_three_characters



EXAMPLE 8.5 Calculate employee's pay

A program is required by a company to read an employee's number, pay rate and the number of hours worked in a week. The program is then to validate the pay rate and the hours worked fields and, if valid, compute the employee's weekly pay and print it along with the input data.

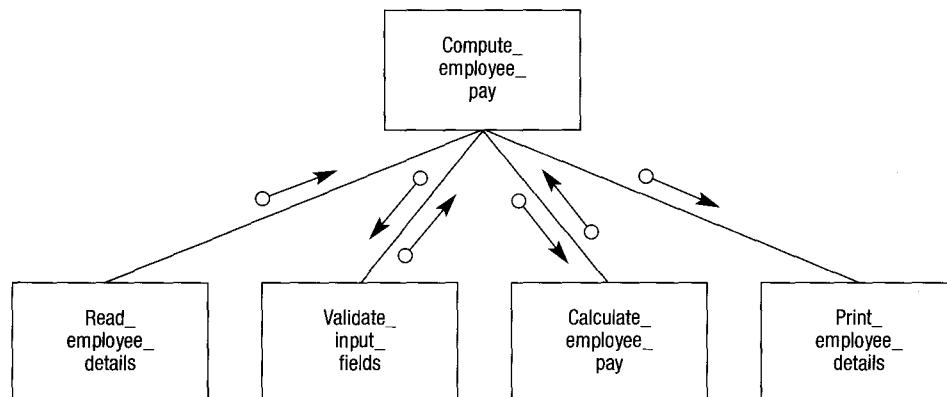
Validation: According to the company's rules, the maximum hours an employee can work per week is 60 hours, and the maximum hourly rate is \$25.00 per hour. If the hours worked field or the hourly rate field is out of range, the input data and an appropriate message is to be printed and the employee's weekly pay is not to be calculated.

Weekly pay calculation: Weekly pay is calculated as hours worked times pay rate. If more than 35 hours are worked, payment for the overtime hours worked is calculated at time-and-a-half.

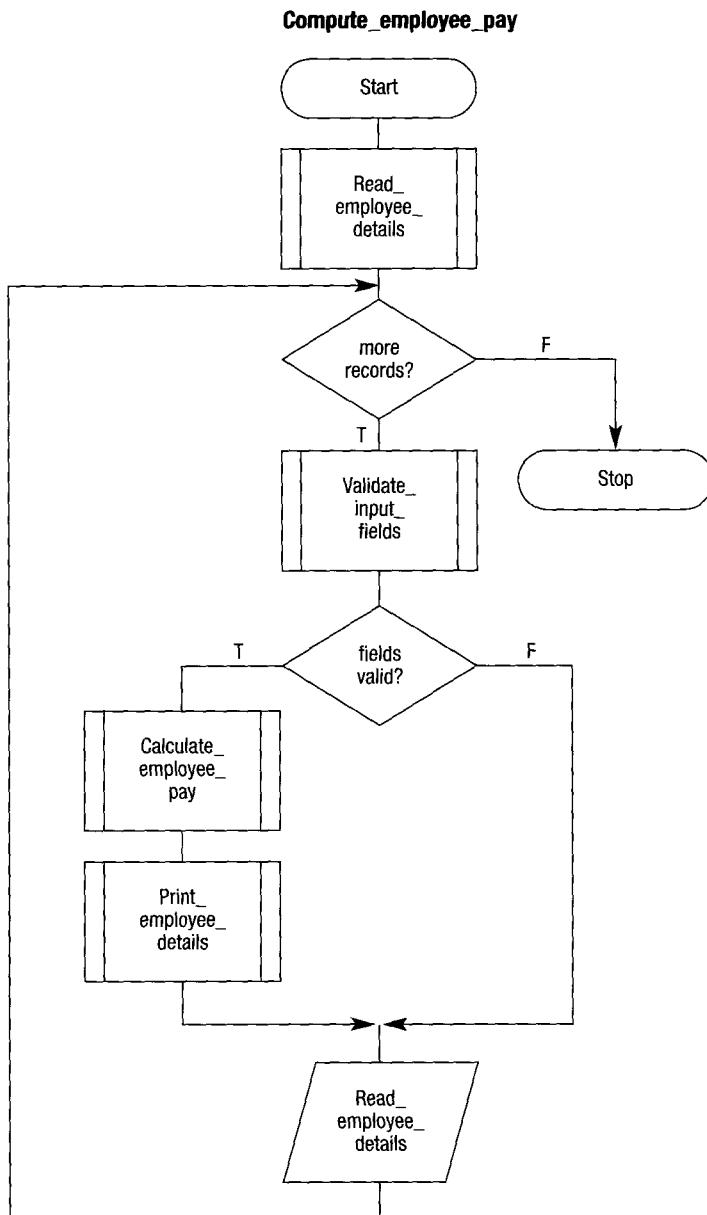
A Define the problem

Input	Processing	Output
emp_no	Read employee details	emp_no
pay_rate	Validate input fields	pay_rate
hrs_worked	Calculate employee pay	hrs_worked
	Print employee details	emp_weekly_pay
		error_message

B Hierarchy chart



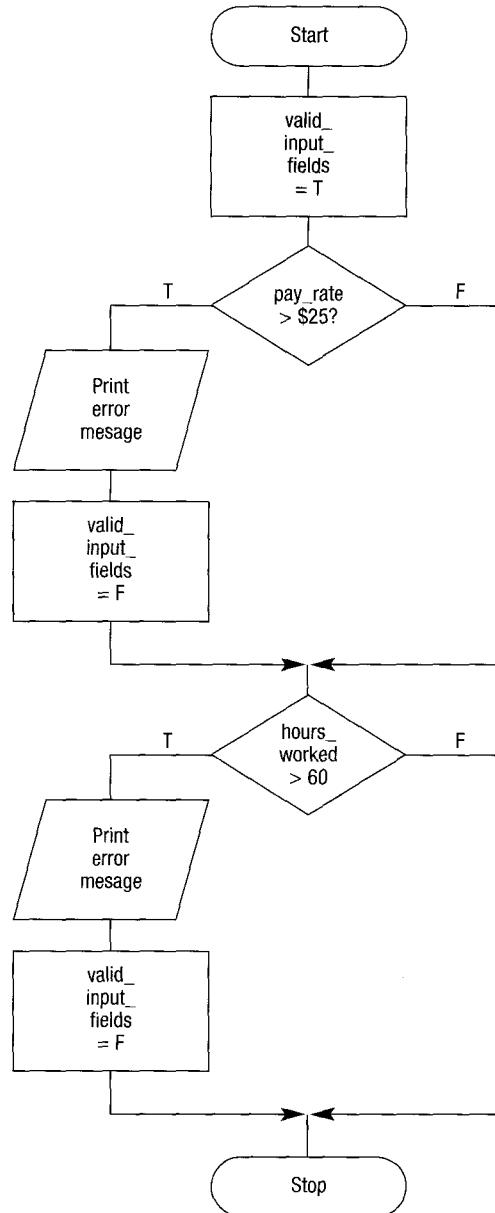
C Solution algorithm



(Example 8.5 continued next page)

(Example 8.5 continued from previous page)

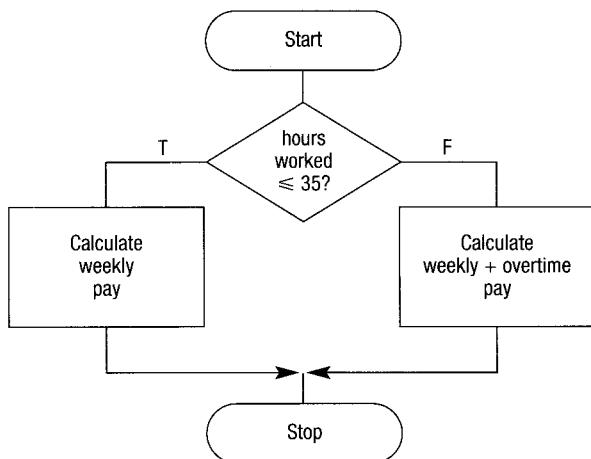
Validate_input_fields



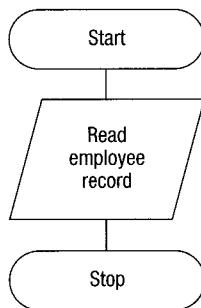
(Example 8.5 continued next page)

(Example 8.5 continued from previous page)

Calculate_employee_pay



Read_employee_details



Print_employee_details

