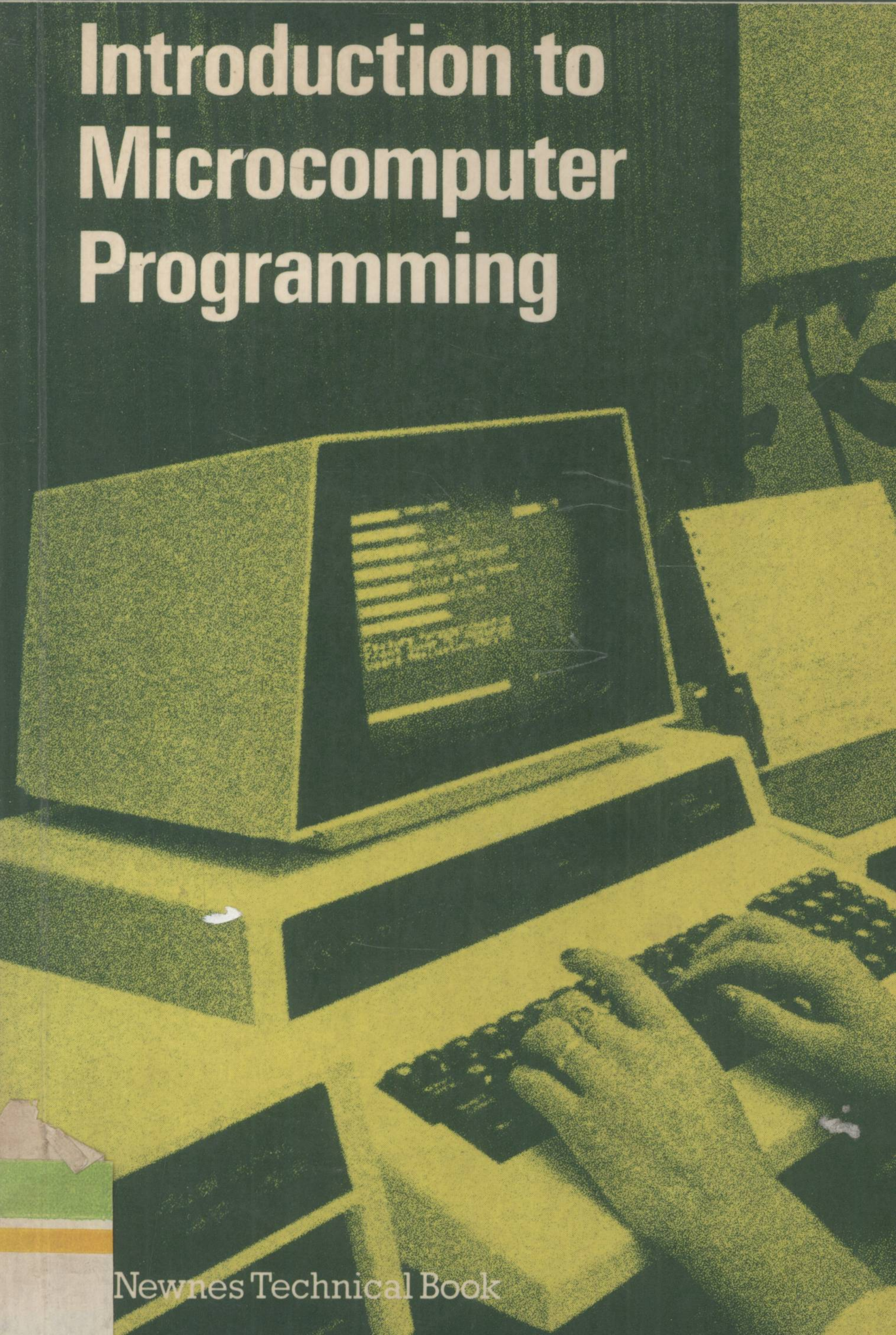
Peter C. Sanderson

Introduction to Microcomputer Programming



Newnes Technical Book

TP31
S216

5

Introduction to Microcomputer Programming

PETER C. SANDERSON, MA

Senior Advisory Officer (Computers), Local Authorities Management Services and Computer Committee

NEWNES TECHNICAL BOOKS

The Butterworth Group

United Kingdom	**Butterworth & Co (Publishers) Ltd** London: 88 Kingsway, WC2B 6AB
Australia	**Butterworths Pty Ltd** Sydney: 586 Pacific Highway, Chatswood, NSW 2067 Also at Melbourne, Brisbane, Adelaide and Perth
Canada	**Butterworth & Co (Canada) Ltd** Toronto: 2265 Midland Avenue, Scarborough, Ontario M1P 4S1
New Zealand	**Butterworths of New Zealand Ltd** Wellington: T & W Young Building, 77–85 Customhouse Quay, 1, CPO Box 472
South Africa	**Butterworths of New Zealand Ltd** Durban: 152–154 Gale Street
USA	**Butterworth (Publishers) Inc** Boston: 10 Tower Office Park, Woburn, Mass. 01801

First published 1980 by Newnes Technical Books,
a Butterworth imprint

British Library Cataloguing in Publication Data

Sanderson, Peter Crawshaw
Introduction to microcomputer programming.
1. Minicomputers – Programming
I. Title
001.6′42 QA76.6 79-42861

ISBN 0-408-00415-0

Typeset by Butterworths Litho Preparation Department
Printed in England by Fakenham Press Ltd.,
Fakenham, Norfolk

Preface

The development of microprocessor-based computer systems has demolished the price barrier that hitherto inhibited computer usage. The use of a computer system is now feasible for a wide range of new users, especially small businesses, teachers and hobbyists. Yet, although systems are cheap, they are useless without programs. Commissioning a special program for individual use is expensive; a suite of business programs is likely to be more costly than the microcomputer system itself. On the other hand, using a 'package' program confines the user within the straitjacket created by the concepts of the program supplier. Many applications for businessman and hobbyist alike are so individual that a package solution would be inappropriate and inadequate. Sooner or later, most users of microcomputer systems will have to consider writing some of their own programs if they are to obtain all the benefits they desire from their system.

This book is a simple introduction to programming for users of microcomputers, whether commercial users, domestic hobbyists or teachers. A slight acquaintance with the functions of a computer is assumed, but no advanced mathematical knowledge or familiarity with the detailed electronic construction or workings of microcomputer systems is at all essential for an understanding of the text. No previous knowledge of programming is necessary. The book is essentially a practical guide to programming and a manual of self-instruction. The examples are chosen from familiar fields, and it is intended that these examples and the exercises will be presented to the reader's microcomputer system. Suggested solutions are provided for the exercises.

The sections on the Basic language stress the features that form a common core and are found in the great majority of microcomputer implementations. Chapter 7 is devoted to some of the common variations of Basic found in microcomputer configurations. The chapters on assembly language and machine code deal with the features of the four most commonly used microprocessor chips. Since there is a great deal more to programming than the mere coding of language statements, an initial chapter is devoted to pre-coding activities and the final chapter to program development and testing.

I would like to express my thanks to all the manufacturers and suppliers who provided material about their versions of Basic for Chapter 7. I would like also to mention Anne Patricia, whose life was so tragically cut short in October 1978, and whose efforts on Kim 1 gave me the original idea of writing the book, which grew out of our many conversations on microcomputer programming. In a sense, this work is a memorial to her.

I would like to record my deepest gratitude to Kathlyn Bell for her expert typing assistance, to my daughter Julia, who provided an impetus when inspiration flagged, and to the staff of the publishers for their help and guidance.

Peter C. Sanderson

Contents

1

Introduction to computer programming

A computer system that you cannot program has been compared to a tool without a handle. If you have spent many hours in assembling and testing your own kit, or have become the proud possessor of a readymade system, you will sooner or later wish to write your own programs.

A *program* is a series of instructions that enable a computer to perform the task you wish it to do, whether displaying football pool permutations or calculating future loan repayments. It is written in a language intelligible to the computer system. When you write computer programs, you have to be familiar with the particular programming code (or *language*) in which you are working. This has some resemblance to the activities of knitting or map reading, where you have to familiarise yourself with the code in which knitting patterns are expressed or the code of the conventional map symbols and contour colours.

The word 'program' is the first of the many technical terms that will be used in this book. Some element of jargon is unfortunately inevitable in a work about computers. In this chapter unfamiliar terms are italicised when they are introduced, and are then defined. At the end of the book there is an alphabetical glossary of the special terms that you will meet in this book and may encounter elsewhere in your reading about computers.

If you are using your microcomputer system as a student, you will almost certainly need to write your own programs. If you are a hobbyist, you will sooner or later find that the readymade or 'package' programs supplied by the microcomputer system manufacturer or taken from computer magazines will fail to meet your individual needs. Domestic diaries and budgetary systems vary from household to household, so standardised programs are unlikely to meet individual requirements. If you are using a microcomputer system for control of some external device you are not likely to find an exact, tailor-made system to connect to your equipment for synthesising music or controlling central heating or a model railway system. The computer games that you can buy ready-programmed may pall, and sooner or later you may wish to introduce your own variations into them or to design some game of your own invention, for which you will have to compose the necessary programs.

Programming a microcomputer system is essentially the same task as programming a conventional large computer (these are usually called *mainframes*) or the medium-sized computers known as *minicomputers*. Indeed, if you use certain programming languages for your micro-computer, you will find that your programs will be able to run on a conventional mainframe such as an IBM 370/158. Certainly the preparatory work before writing down the statements in the appropriate programming language is common to all types and sizes of computer system.

All computers follow slavishly the instructions in the program. They are not telepathic and cannot tell if you have omitted an instruction because, to the human mind, it seemed too obvious. If you forget to instruct some computers to halt or stop at the end of the program, or forget to ensure that you do not give an instruction to perform division when one of the numbers involved may be zero, you will have a non-sensical answer displayed or printed. When writing computer programs, you have to abandon all human intellectual pretensions and look at instructions at the level of the machine! All instructions in your program will be meticulously obeyed, but the computer will make no attempt to discover whether they are sensible or comprehensive.

Thus programming can be frustrating and demands close attention to detail. Yet it is not insufferably difficult; no more so than learning to play simple tunes from conventional musical notation on a piano or recorder. Highly successful computer programmers have sprung from the most unlikely non-mathematical backgrounds. The rapidly growing number of computer hobbyists shows that there is nothing fundamentally formidable in writing programs and that it can add challenge and exhilaration to your hobby.

The most important rule in designing successful programs is to avoid rushing into writing the actual programming language instructions until a great deal of preparatory work, which will be described in the rest of this chapter, has been done. There is a fundamental distinction between writing the actual instructions (known in the computer world as *coding*) and the real work of program design. Except for the simplest programs, coding is less than a quarter of the work involved. As in decorating a house or a room, the more time spent in preparation, the better, more elegant and longer-lasting is the result. As paintwork on uncleaned or unprimed woodwork soon needs renewal, so hastily coded programs soon need rewriting or drastic alteration. In fact there is a pronounced likelihood that they will not even run.

There are five main steps in preparing a program, prior to entering it on the keyboard or switches of the microcomputer system in the appropriate code or computer language.

1. Ascertain whether the problem is feasible for solution on your microcomputer system.
2. Define the problem precisely.
3. Consider possible methods of solution.
4. Break down the problem into small steps suitable for representation in programming language statements or instructions, and express these in visual form. (This is known as *flowcharting* and will be defined in greater detail in the course of this chapter.)
5. Document the statements so that it will be easy to operate the program from this documentation, which will also assist when you wish to extend, or make alterations in, the original system.

These steps will be described in detail in the rest of this chapter.

Ascertaining the feasibility of your problem

Most problems are capable of being solved on a computer system if they can be expressed in a logical series of finite steps. In theory, any problem that contains no irrational or intuitive elements can be programmed for solution on a computer, although in some cases the program would be too complex to write in a reasonable amount of time. You can design a program to analyse past form to predict the winner of a football game or a horse race, but you cannot blame either the program or the microcomputer if they fail to select the actual winner, since elements that defy logic are invariably involved in sporting competitions.

Some problems, which in theory can be solved by a computer, will be incapable of being solved on the particular equipment you possess, or can only be solved with a great deal of difficulty. If your system does not have a typewriter keyboard, it is best to avoid applications where you have to enter letters of the alphabet, and of course elaborate graphical displays are not possible if you only have an LED display. If you wish to use a large file of data in your program, it may be very time-consuming and cumbrous to have to change many cassettes of tape during the operation of the program.

You may find you have a program too large for the *store* or *memory* of your equipment. You will easily be able to find out the store size from the manual that accompanies a readymade system, and if you have built your own system you will know exactly how much store you have attached to it. You will soon become adept at estimating the size of a program in the early stages of design. You can then decide whether the problem should be abandoned (or left until more store is purchased) or whether it can be conveniently broken down into a series of smaller problems, so that you can enter results produced by one program into the next program in the sequence.

In the spirit of a fledgeling pianist attempting a Brahms sonata, you may decide that, although a projected program would be feasible, it would be too difficult until you have gained more experience. This difficulty should not cause you to think of yourself as a coward or a slow learner. It has been estimated in the USA that a standard output of correct *machine code* (the most difficult form of computer language, which will be fully explained in the next chapter) for a programmer is ten instructions daily. Therefore you may not be able to spare the time to program a complex problem.

If you are a small business user of microcomputers you will probably also consider the cost savings and potential benefits of introducing a specific microcomputer-based system before embarking upon the programming. Such considerations will also determine which projects are programmed first.

Definition of the problem to be programmed

The easiest problems to define for solution by programming a microcomputer are numerical. There are many published *algorithms* (rules of a procedure for solving a specific problem, often applied to rules expressed in a computer language) for numerical problems, which may need only slight alteration for your microcomputer system. Whether you use these or whether you completely design your own program, you should ensure that:

- care is taken if any number or intermediate result is likely to be zero or negative;
- no number or intermediate result is likely to become too large or small for your system (the processor manual will provide the range of numbers that can be represented);
- you can obtain the accuracy to the number of decimal digits you want;
- you are inserting satisfactory checks on numbers you insert from the keyboard or switches: 'finger-trouble' can easily occur, and is not always easy to detect at the moment when a wrong insertion is made.

If you are programming a problem where you are working with interrupt signals from outside the microcomputer, such as from a model-train layout, you should make sure that in the program definition the precise timing requirements are defined. You will also have to think carefully about what you are going to insert in the program for unusual and error conditions.

Domestic programs need to be as precisely defined as mathematical problems. You must try to visualise the various ways in which your family will enter certain items from the keyboard. If you are working on a calendar or diary application, for instance, you will have to decide whether you will accept 11 June, 11 Je, 11.6 as being equally acceptable entries, and whether you will accept numeric '0' and alphabetic 'O' as being interchangeable. The insertion of checks on keyboard data can be overdone, but it is as well to cater for major variants in the early stages of system design.

In both domestic and business programs you are likely to be involved in the processing of a cassette tape file against information entered from the keyboard. A typical example of this type of application would be the reconciliation of credits and debits that you have entered on cassette tape with entries from a monthly bank statement that are being entered, item by item, from the keyboard. You will have to program for appropriate action to be taken for unmatched items, which will be indicated by reaching the end of the file when there are still items to be entered on the keyboard or by completing the entry of keyboard items before reaching the end of the file. If your expenditure and income entries are on more than one cassette, you will have to program to display a message to insert another cassette. In this type of program you will have to be liberal in displaying or printing messages for the guidance of data entry through the keyboard and for explaining unusual or error conditions to the person operating the microcomputer system. In all programs, you should avoid displaying a result as a string of digits if your microcomputer system can print or display explanatory headings and text with the figures.

Flowcharting

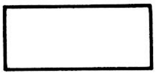
When you have defined the problem, you will have to consider methods for solution and ultimately select one of these methods. You will then have to break the method down gradually into small steps; eventually each step will be equivalent to a single program statement in the language you are using. To a certain extent, the processes of selecting a method and decomposing it into program statements are complementary, since often a detailed examination of a method originally selected will prove its unsuitability. Then it is necessary to start again with the consideration of an alternative method for solving the problem that you have previously defined.

A flowchart has been previously defined as an expression of program steps in a visual form. Invariably you do not leap into writing the detailed program steps, but start with the broad stages in solving the problem. We can therefore extend our definition of a flowchart to include a visual representation of the stages of the process to solve a particular problem on a computer.

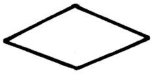
The expression of the steps or procedure for solving a problem in visual form is a valuable aid to programming. When a formula is written down, or a series of instructions (like a recipe) is studied, it is difficult to realise that they are an expression of a number of discrete steps that follow one another in time, because we tend to see the formula or instructions as a whole. However, the computer only obeys one instruction at a time, so a flowchart is ideal for the representation of steps in a computer program since it illustrates the flow of steps in time sequence. A flowchart often makes the subsequent steps to an operation in solving a specific problem more obvious and assists in avoiding repetition of steps. It is easier to alter a flowchart than detailed computer language statements in a program, especially for the beginner.

Flowcharts are by no means confined to computer programming. They are used in a growing variety of applications including mechanical assembly, chemical production and fault-finding in machinery. A set of flowcharts for non-computer procedures has been published for general use in British local government.

The shapes of the 'boxes' in which the steps of a flowchart are written have been standardised in BS 4058. Only the two chief symbols will be used in this book, as it is perfectly feasible for an amateur programmer to manage by using only these two. They are:



for a process or a calculation



for a decision (e.g. is a number zero?); it has two exits, 'yes' and 'no'

The use of these symbols is shown in a simple flowchart for the game of Snakes and Ladders (Figure 1.1). This example illustrates several important features in the construction of flowcharts:

1. Boxes are provided for 'start' and 'stop'.
2. The amount of detail in a step is entirely up to you. In this diagram, we have included both landing on a 'snake' and landing on a 'ladder' in one box (I).

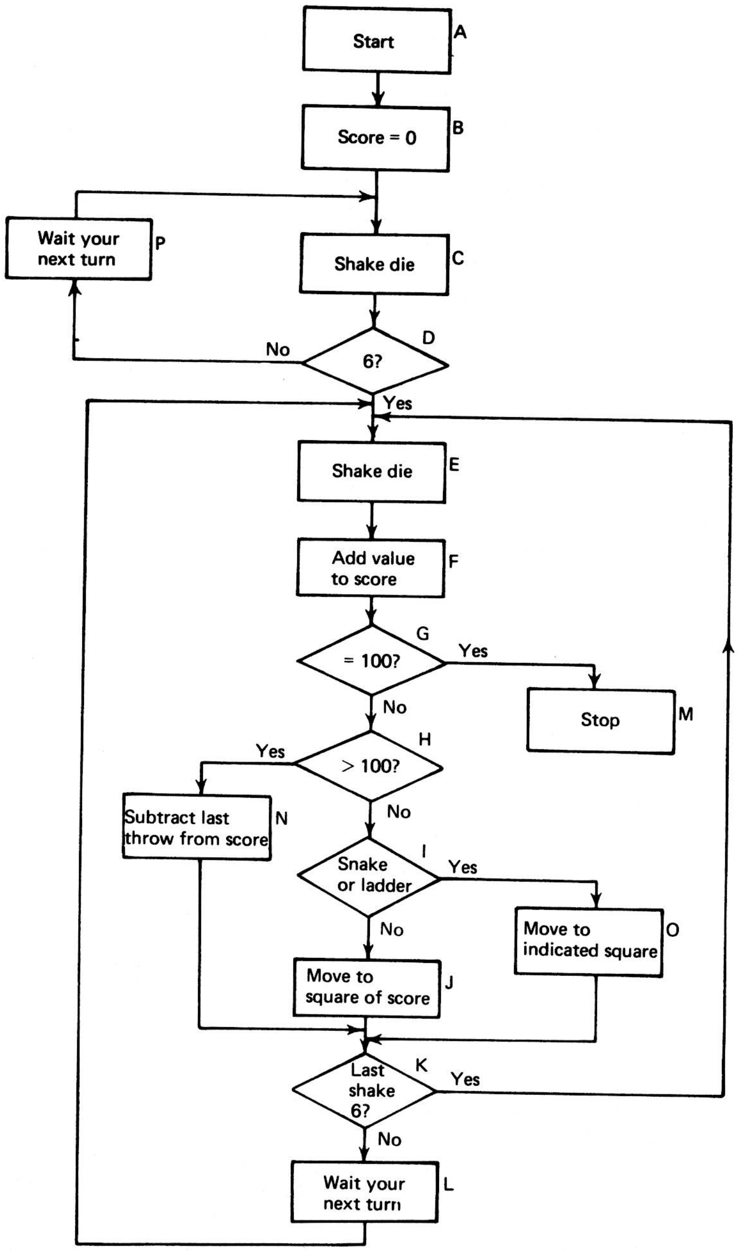


Figure 1.1 Flowchart for Snakes and Ladders

3. There is rarely a single correct solution for a specific flowchart or program. You may doubtless be able to find a different, but equally valid, solution to the snakes and ladders problem. For instance, in box G you could make the test '<100?' and make the 'yes' exit to this the main path of the program.
4. The diagram shows the complexity of a problem that at first sight seems trivial. It also shows the importance of putting the decision-boxes in their correct order so that you can avoid writing identical steps in many different branches of the flowchart or program and (hopefully) avoid any combinations of events where the wrong action is taken.

You may by now be convinced that time spent in planning a program in the flowchart stage will save time and confusion when you are entering the program on the keyboard or switches of your microcomputer system.

Usually you initially flowchart the problem in outline, and then produce successive detailed breakdowns of it until you approach the detailed individual statements of the programming language in which

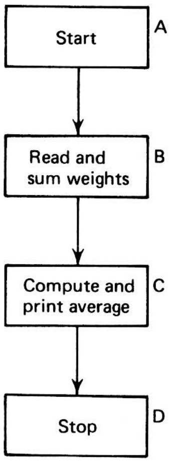


Figure 1.2 Outline flowchart for averaging weights

you are writing. A problem to read-in your monthly weight twelve times and display or print the average over the year is flowcharted in broad outline in Figure 1.2. This represents the way in which humans would solve the problem. When you commence breaking it down into more detailed steps for programming you will get closer to solving the problem as the computer would see the instructions to solve it.

You usually have a method of reference from one level of flowchart to another, so that you can more easily understand the detailed levels –

perhaps some months after the program has been written – by referring back to the broad outline. Often the first-level boxes are A, B, C etc. Then at the second level the boxes referring to A start at A1 and can go up to A9; at the third level they can go from A10 to A99 (A1 expanding to A10–A19, A2 expanding to A20–A29, etc.); at the fourth level from A100 to A999 (A100–A109, A110–A119, etc.) – and so on, as necessary. It is convenient to limit the number of substeps relating to a specific step at the previous level of detail to ten.

The above numbering convention is used in the more detailed breakdown of the weights problem (Figure 1.3). The flowchart has been expanded to include the display of a title and of a message indicating that all weights have been entered. This flowchart introduces the important concept of a *loop*, i.e. a section of program that you want to repeat many times but only write once. All box B of the broad-outline flowchart (boxes B1–B5 of the more detailed one) is the loop here. This type of loop is a very common technique in programming when you know the exact number of items you wish to process. The loop is controlled by a count, which is (usually) set to zero before the loop is entered. When you have come to the end of the processes you wish to perform on an item of the loop, one is added to the count, which is tested against the number of times you wish the loop to be obeyed. If the count is less, the loop is repeated again.

To illustrate the loop in the weights problem, we will reduce the number of weights entered to three (if it works with three it will work with 12, or any number you wish) and observe various totals. This method of checking a flowchart with a very small number of items is a very suitable technique in estimating how correct your program design really is.

Count at B1	*Total at B1*	*Weight entered (kg)*	*Total at B3*	*Count at B4*	*Result of B5 test*
0	0	61	61	1	Yes
1	61	63	124	2	Yes
2	124	64	188	3	No

Sometimes the number of times you wish a loop to be obeyed varies each time you run the program, so that you cannot use a count against a fixed number. The usual way of catering for this type of problem is to enter, at the end of the data, a number (called a 'sentinel') that cannot possibly occur in valid entries. An appropriate one for the weight problem would be 0 or –1, which even the most dedicated weight-watcher could hardly hope to achieve. The sentinel technique is very useful and more flexible than using a count against a fixed number. You must, however, ensure that the sentinel value (which

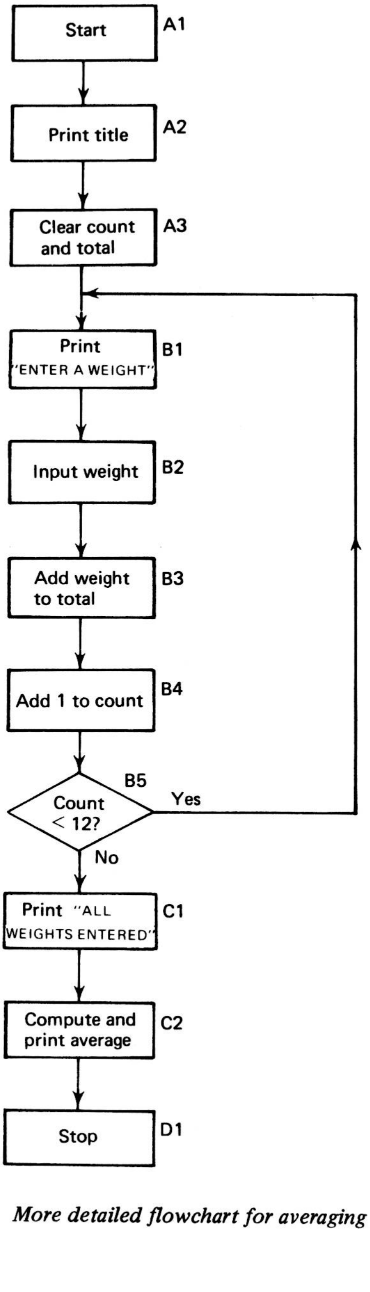


Figure 1.3 More detailed flowchart for averaging 12 weights

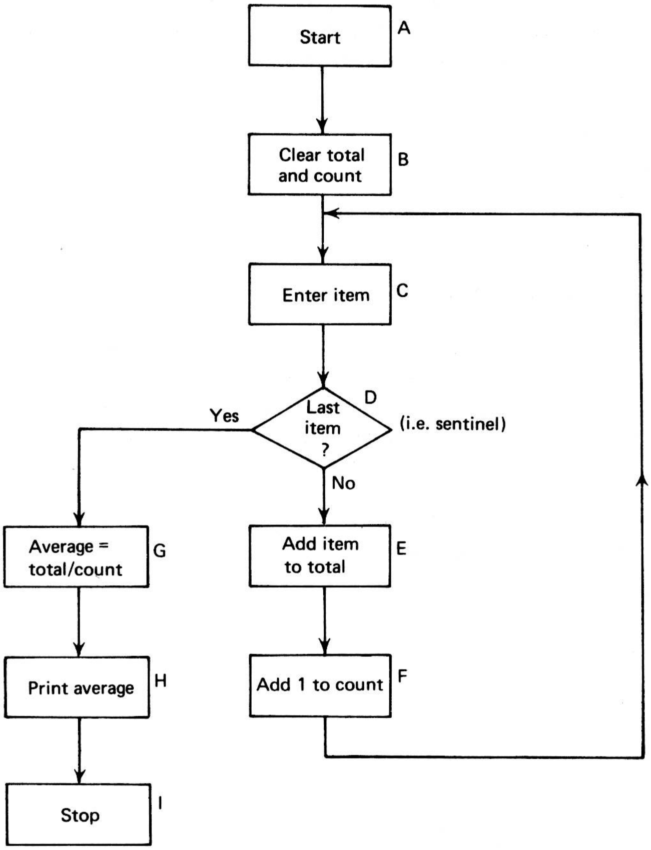


Figure 1.4 Flowchart of 'sentinel' loop technique

could be −1) is tested *before* each data entry is added to the total. Otherwise the sentinel itself would be added to the total.

The flowchart in Figure 1.4 illustrates a solution of the problem to calculate the average of a variable number of items ended by a sentinel. The loop is tested below for data entries of 70, 74 and 0 (sentinel)

Item	*Test at D*	*Total at F*	*Count after F*
70	No	70	1
74	No	144	2
0	Yes		

The previous flowcharts have obliterated each data entry as soon as the next one is read. Often you wish to store each entry for use in a