PROGRAM DESIGN

Peter Juliff

PHILIP SALMON
P.O. Box 345
DRUMMOYNE NSW 2047
*a/d Phone: (02) 818 1111

0185

PROGRAM DESIGN

PROGRAM DESIGN

Peter Juliff

Head, Department of Computing
Victoria College, Prahran Campus

Prentice-Hall of Australia

Prentice-Hall of Australia Pty Ltd, Sydney
Prentice-Hall International Inc., London
Prentice-Hall Canada Inc., Toronto
Prentice-Hall of India Private Ltd, New Delhi
Prentice-Hall of Japan Inc., Tokyo
Prentice-Hall of Southeast Asia Pte Ltd, Singapore
Editora Prentice-Hall do Brasil Ltda., Rio de Janeiro
Whitehall Books Ltd, Wellington
Prentice-Hall Inc., Englewood Cliffs, New Jersey

Typeset by Abb-Typesetting Pty Ltd,
Collingwood, Vic.
Printed in Australia by
Impact Printing, Brunswick Vic.

2 3 4 5 88 87 86 85 84

National Library of Australia
Cataloguing-in-Publication Data

Juliff, P.L. (Peter Laurence), 1938–.
Program design.

Includes index.
ISBN 0 7248 1006 4.

1. Algorithms. 2. Electronic digital computers — Programming. I. Title.

001.64'2

CONTENTS

PREFACE

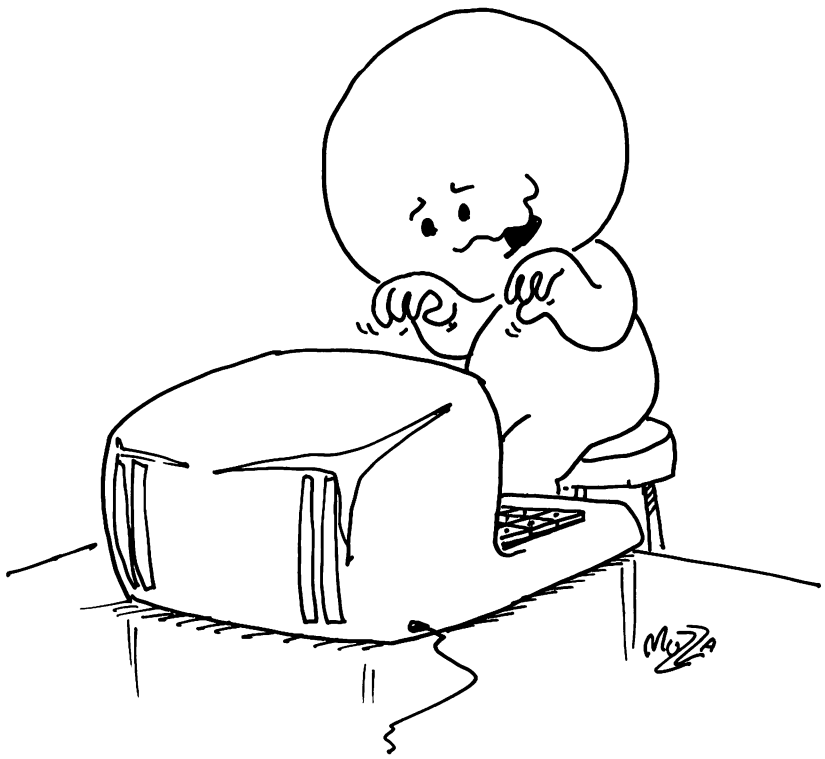
This book has grown from my lecture notes over a period of ten years. I hope it will provide a programming reference that is suitable for a wide range of students and practitioners.

I would like to acknowledge the help that I have received from my colleagues in the compilation of this text. Heinz Dreher (West Australian Institute of Technology) and Tony Watson (West Australian College of Advanced Education) are responsible for much of the consistency of presentation of the material, and Alan Forsyth (Victoria College) provided valuable assistance with the Pascal appendix.

My thanks go particularly to Leah Leck for her indefatigable typing and re-typing and to Maurie Marion for his artistic contribution.

Peter Juliff
Head, Department of Computing
Victoria College
Prahran Campus
Victoria, Australia

WRITING PROGRAMS



WRITING PROGRAMS

Programming is not only a creative and stimulating activity but also an intellectually rigorous discipline. Without wishing to detract from its enjoyment, one must be careful to distinguish between 'amateur' and 'professional' programming. I do not use the term 'amateur' in a perjorative sense but rather to describe the person who writes programs mostly for his own use and enjoyment. Such a person is usually primarily concerned with producing a correct solution to a relatively small problem and not with the elegance of the algorithm, its ability to be maintained or its documentation. The professional programmer, on the other hand, will be concerned with a relatively large and complex problem and must consider many more criteria than simply solving the problem correctly. His work will be primarily for use by other people with whom the programmer has no contact and who may be called on to modify or enhance the programs to meet their changing requirements over a period of many years.

It is my intention to address the professional programmer and to urge the amateur programmer to adopt the same disciplined approach to his work.

I would like to make some assertions about programming:

- Programming is an inherently difficult and demanding task.
- Coding, the expression of the solution in a particular programming language, is only one small facet of programming and arguably not the most important one.
- Programming is a discipline to which certain principles may be applied independent of computer type, language or application.
- There are objective criteria for good programs; it is not a matter of personal aesthetics.
- Despite the appearance of correctness of solution as a *sine qua non*, the prime criteria in judging a program are structure and style rather than 'cleverness' of the algorithm.
- Given that programs process data, an understanding of the nature and structure of the data is an essential prerequisite to the construction of any program.
- A programmer must be familiar with as many techniques as possible, aware of their pros and cons and able to recognize situations for their application (modified where necessary).

The first of these precepts will be self-evident to any programmer. I will elaborate on the rest in this book.

The first section of the book deals with the construction of algorithms. They are first considered at a micro, or individual, level to formulate guidelines for elegance of construction. Consideration is then given to their grouping to comprise a program, i.e. in terms of their structure within the program as a whole and their communication among one another. By this means I hope to illustrate that the process is recursive in that the rules relating to the construction of any individual routine are just as applicable to the overall program itself.

The second section relates to the construction and processing of data structures. Algorithms are inextricably interwoven with the data which they process and their structure arguably should be modelled on that of the processed data.

In this book algorithms are expressed in pseudocode which I regard as the best of the contemporary methods for their formulation. Pseudocode is a method of expressing program algorithms in English statements which are connected by a limited number of control/sequence rules. It is sometimes referred to as Structured English, and its format will be explained in Chapter 1.

NOTATION

In an attempt to free the text from the syntax of any specific programming language, I have adopted a notation which is similar to that used in most contemporary programming texts. The notation also partly conforms to the syntax of many programming languages.

The symbols used are:

+ addition
− subtraction
* multiplication
/ division
← assignment
. . . a point in an algorithm where detailed statements have been omitted for brevity.

Hence instead of

Put zero in TOTAL

I have adopted

TOTAL←0

which may be read as 'TOTAL gets a value of zero', and instead of the more colloquial

Add 1 to COUNTER

I have used

COUNTER←COUNTER + 1

There are places in the text where I have not adhered strictly to these conventions, but the meaning of the non-standard procedures should be clear.

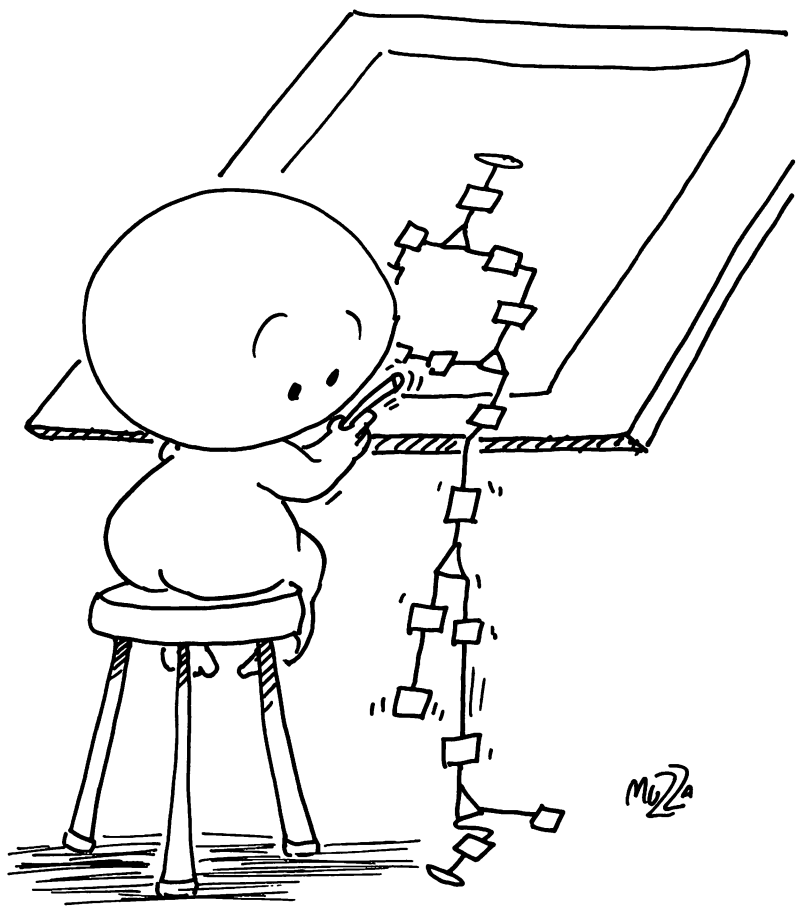
For the sake of readability I have used an underscore character in the names of many variables, believing that

EMPLOYEE_COUNTER

is inherently more readable than

EMPLOYEECOUNTER.

1 ALGORITHM DESIGN



CHAPTER 1

1.1 ALGORITHM DESIGN

An algorithm is a formula, a recipe, a step-by-step procedure to be followed in order to obtain the solution to a problem. To be useful as a basis for writing a program, the algorithm must:

(a) arrive at a correct solution within a finite time,
(b) be clear, precise and unambiguous, and
(c) be in a format which lends itself to an elegant implementation in a programming language.

Algorithm specification has traditionally been done by drawing flowcharts which are a diagrammatic representation of the steps involved. Throughout the book pseudocode will be used instead of flowcharts except for a brief introduction in this chapter to illustrate the relationship between the two techniques.

1.2 STRUCTURED PROGRAMMING/ SOFTWARE ENGINEERING

In the 1960s Professor Edsger Dijkstra of Eindhoven University was warning programmers of the danger of the indiscriminate use of branch instructions (GO TOs) as a means of program control. In a paper published in Italy in 1968, Böhm and Jacopini demonstrated that no program need consist of a combination of any more than the three control constructs of sequence, repetition and selection. The term 'structured programming' was coined to describe a body of technique incorporating and formalizing these concepts which until that time had largely existed only as personal standards of particular programmers. Further academic contributions were made by people such as Niklaus Wirth and David Parnas who stressed that there was a discipline which could be brought to bear in developing programs and led to a more theoretical extension of programming technique into 'software engineering'.

We as programmers owe a great deal both to the academics who contributed to such a body of knowledge and to the popularizers and pragmaticians such as Ed Yourdon, Tom De Marco, Gerry Weinberg and

Michael Jackson who were responsible for translating the theoretical concepts into practical techniques and heuristics which ordinary programmers could understand and follow.

1.3 CONTROL STRUCTURES

The key to elegant algorithm design lies in limiting the control structure to only three constructs. These are illustrated in Figures 1.1, 1.2 and 1.3 in flowchart and statement form.

1. Sequence

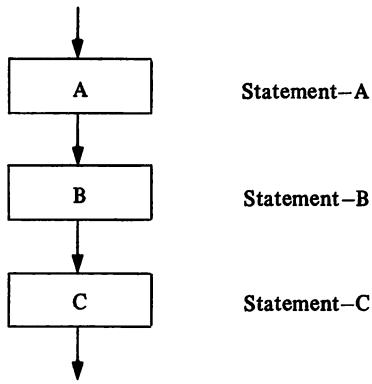


Figure 1.1

2. Repetition (or Iteration)

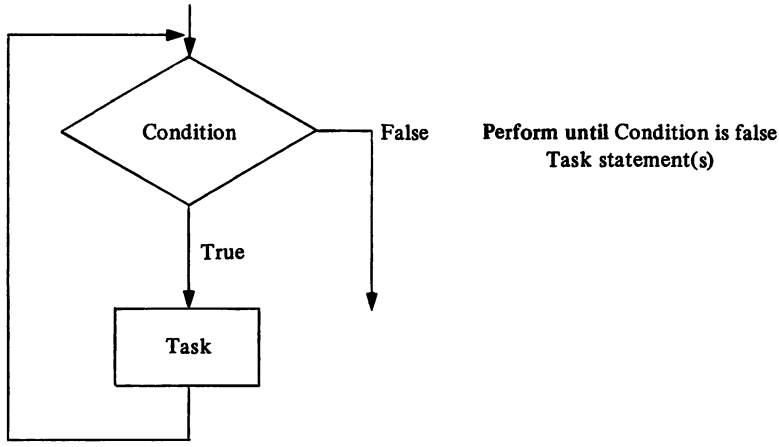
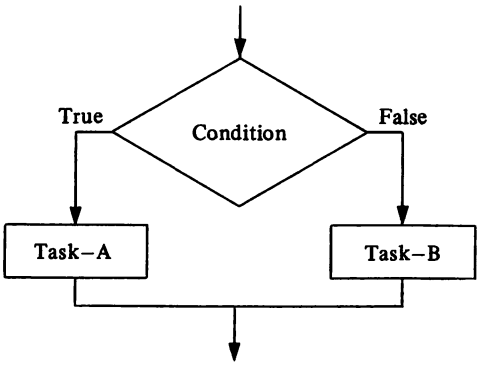


Figure 1.2

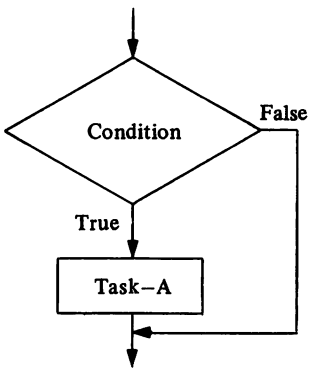
3. Selection




```
If Condition is true
    Do Task–A
else
    Do Task–B
```

Figure 1.3

A variation on the construct of Figure 1.3 is shown in Figure 1.4.




```
If Condition is true
    Do Task–A
```

Figure 1.4

Using only the Selection construct of Figure 1.4 leads to a rather complicated structure for multiple testing, as shown in Figure 1.5.

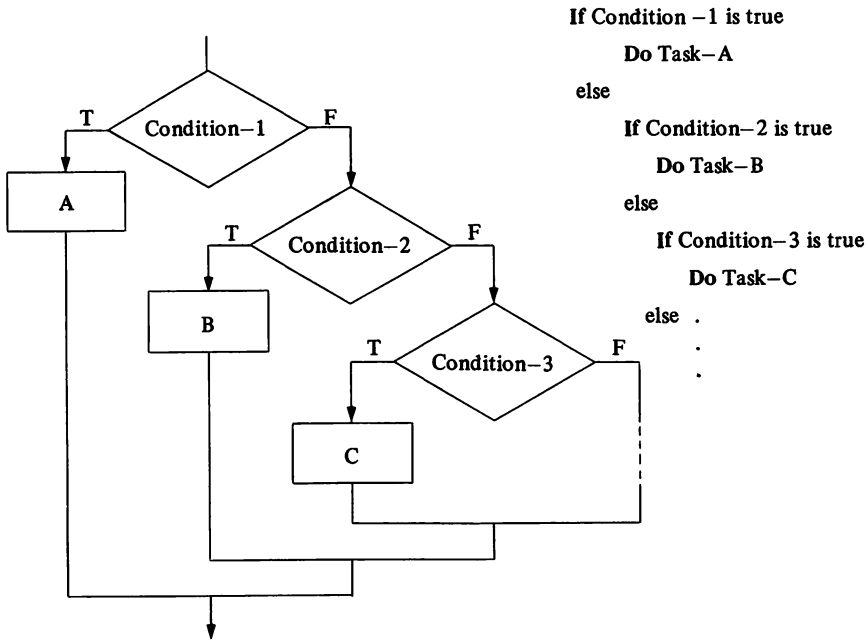


Figure 1.5

Hence a preferable solution is that shown in Figure 1.6.

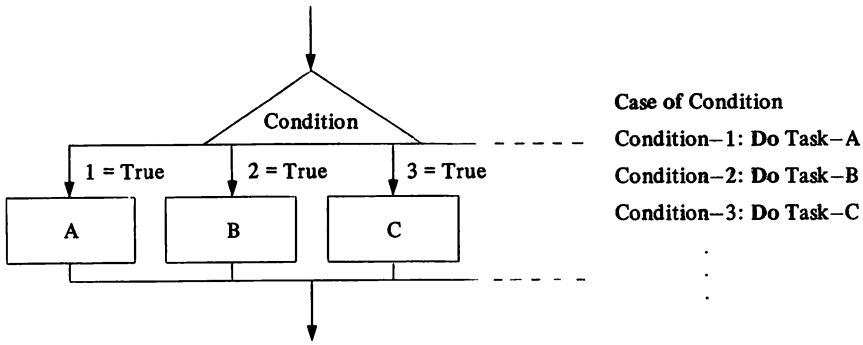


Figure 1.6

At this point I will abandon flowcharts as a means of expressing algorithms. Their main deficiency lies in their allowing the designer of an algorithm to develop structures which do not conform to the three control constructs listed previously and which could therefore lead to inelegant and dangerous implementation in whatever programming language is used to solve the problem. The patterns of logic in Figure 1.7, for example, are seductively easy to depict in flowchart form.

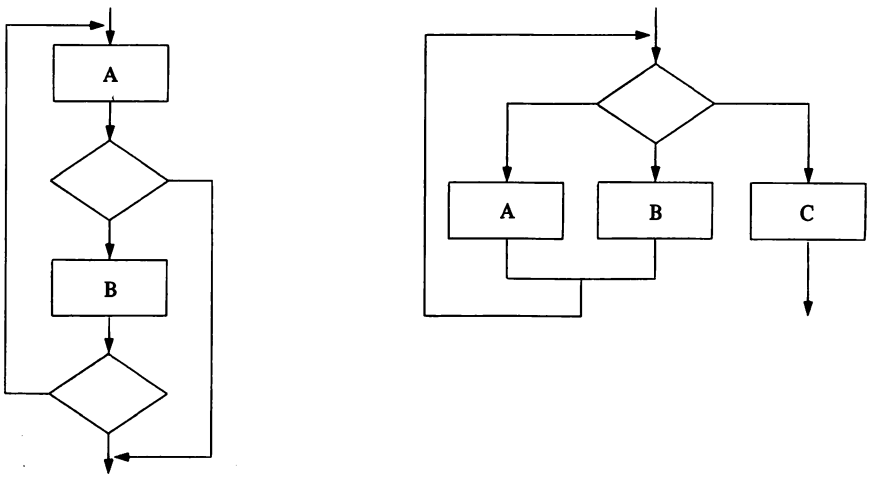


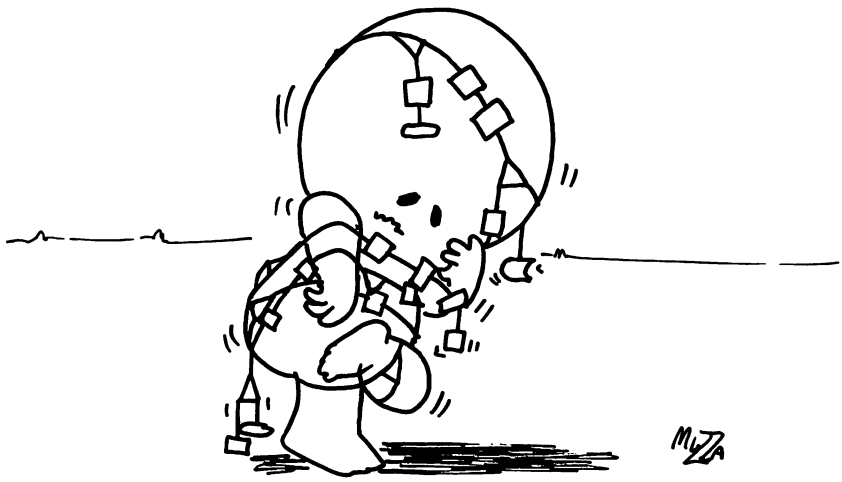
Figure 1.7

They are incapable, however, of being expressed in structured syntax in any programming language. This will not preclude their being implemented in that language, of course, but will make it possible only by the excessive use of GO TOs and their associated labels.

The advantage of using an English-like pseudocode with strictly limited control structures overcomes the two main deficiencies inherent in flowcharts:

(a) for the most part we are unable to write anything meaningful in the boxes, and

(b) as stated previously, we have little effective control over the constructs which we draw, and our transgressions ultimately become embedded in the code generated from that flowchart.



1.4 PSEUDOCODE

In contrast to the shortcomings of flowcharts, the advantages of pseudocode are:

(a) statements are written in free English text and, although brevity is desirable, they may be as long as is needed to describe the particular operation,

(b) by providing only the three structured control concepts of sequence, repetition and selection, the technique does not permit us to use branches to labels, and

(c) many languages, such as Pascal, exist (and more are certain to follow) which have syntax that is almost identical to pseudocode and hence make the transition from design to coding extremely easy.

The conventions followed in this book for pseudocode are as follows:

(a) Individual operations are described briefly and precisely in English statements consisting of nouns and verbs.

(b) Groups of statements may be formed into modules or routines and

the group given a name, for example, Calculate Pay. Such a routine may be executed by the statement:

Do Calculate Pay, e.g.

Get employee's work details
Do Calculate Pay
Print pay envelope details

(c) Sequence of operations is effected by writing instructions on consecutive lines.

(d) Repetition of operations is effected by a '**Perform until** ...' statement which indicates that all of the following indented operations are to be repeated until that condition is true. The condition is tested immediately before executing the group of statements for the first time and thereafter immediately before each prospective iteration. Hence, if the governing condition is true when the '**Perform until**' is first encountered, the subordinate instruction(s) will not be executed at all.

The following task, for example, would be executed four times:

```
Counter←4
Perform until Counter = zero
    statement-1
    statement-2
        .
        .
        .
    Counter←Counter − 1
```

Similarly, the task below would not be executed at all if Counter were already equal to Target:

```
Perform until Counter = Target
    statement-1
    statement-2
        .
        .
        .
    Counter←Counter + 1
```

Note the use of indentations of the subordinate statements to highlight the span of control.

In the second example, if Counter was greater than the value in Target when the **Perform until** was encountered, the group of

statements could be executed indefinitely! A better control statement would have been:

Perform until Counter ⩾ Target
(⩾... greater than or equal to)

Such a repetition or iteration concept is referred to as a program 'loop'.

(e) Selection is effected by the **If** ... **else** ... for simple tests and the **Case** statement for multiple tests. The **else** is not essential in a test where the action to be taken upon the condition being false is merely to avoid the action if the condition were true. The following examples should illustrate the constructs.

(i) **If** employee's age ⩾ 18
Adult Counter←Adult Counter + 1
else
Junior Counter←Junior Counter + 1

(ii) **If** employee's sex is female
Female Counter←Female Counter + 1
else
If employee's sex is male
Male Counter←Male Counter + 1
else
Error Counter←Error Counter + 1

(iii) **If** hours worked > 40
Overtime Counter←Overtime Counter + 1

(iv) **Case of** employee's Skill Code:
"E": Engineer Counter←Engineer Counter + 1
"A": Accountant Counter←Accountant Counter + 1
"P": Programmer Counter←Programmer Counter + 1

Note again the use of statement indentation to indicate the span of control of each of the statements.

In all of these examples above, the key words associated with the constructs are in bold type when used, for example, **Do**, **Perform until**, **If** ... **else** ..., **Case of**.

1.5 MODULES

One of the perennial decisions facing the designer of an algorithm is how much work constitutes one module? There are no hard and fast rules, but the following guidelines should be noted:

(a) A module consists of a group of statements which logically belong together to accomplish one comprehensible and indivisible task.

(b) Where any group of statements needs to be incorporated in more than one place in an algorithm, that group should constitute a module which is then invoked from as many points in the algorithm as required.

(c) When in doubt, err on the side of too many modules rather than too few.

1.6 ALGORITHM CONSTRUCTION

At this point we will consider only simple algorithms, essentially only those needed for a single task or a task requiring only a few simple modules. In a later chapter we will consider the means of assembling a large number of tasks to form a complex structure.

The best method of illustrating the procedure of algorithm construction is by the use of a series of examples of increasing complexity.

Example 1.1

Construct an algorithm which will place in AREA the area of a triangle bounded by sides of length A, B and C. Should the dimensions given be such that they would be impossible to construct a triangle (e.g. A = 10, B = 3, C = 2) the content of AREA is to be zero.

Area of Triangle

S ← ½ * (A + B + C)
If (S − A) ⩽ 0
or (S − B) ⩽ 0
or (S − C) ⩽ 0
    AREA ← 0
else
    AREA ← $\sqrt{S*(S-A)*(S-B)*(S-C)}$

Example 1.2

Construct an algorithm to read a series of integers, the first of which indicates the number of integers to follow (e.g. 5, 3, 19, 6, 82, 4) and compute and print their total and average. Exclude the first number from the total and average calculations.

Total and Average

```
Read 1st number and place in COUNTER_A and
    COUNTER_B
TOTAL←0
Perform until COUNTER_A=0
   Read INTEGER
   TOTAL←TOTAL+INTEGER
   COUNTER_A←COUNTER_A−1

If COUNTER_B>0
   AVERAGE←TOTAL/COUNTER_B
else
   AVERAGE←0
Print TOTAL and AVERAGE
```

Example 1.3

Construct an algorithm to read a series of integers until the number 99 is encountered. Compute and print the total and average of these numbers, excluding the 99 from these calculations.

Total and Average

```
TOTAL←0
COUNTER←0
Perform until INTEGER=99
   Read INTEGER
   If INTEGER is not=99
     TOTAL←TOTAL+INTEGER
     COUNTER←COUNTER+1

If COUNTER>0
   AVERAGE←TOTAL/COUNTER
else
   AVERAGE←0
Print TOTAL and AVERAGE
```

Example 1.4

Alter the operation of Example 1.3 to compute the total and average only of integers which lie between 100 and 200 inclusive. This will require modifying only the **Perform until** loop.

Total and Average (part)

```
Perform until INTEGER = 99
  Read INTEGER
  If INTEGER is not = 99
    If INTEGER is not < 100 and not > 200
      TOTAL←TOTAL + INTEGER
      COUNTER←COUNTER + 1
```

The need to make all processing within the loop dependent upon the end-of-data test (i.e. 'If integer is not = 99') will make the generation of code more clumsy as the complexity of the processing increases, as can be seen from the previous algorithm.

It should also be noted that at the point at which the statement

Perform until INTEGER = 99

was included, the variable INTEGER had not initially had any value assigned to it and hence the first test of its value would provide an unpredictable result.

Below, the algorithm is re-cast using a 'priming read' technique in which the first data item is read outside the loop and thereafter each following item is read immediately before the end of the code comprising the loop.

Total and Average (part)

```
Read INTEGER
Perform until INTEGER = 99
  If INTEGER is not < 100 and not > 200
    TOTAL←TOTAL + INTEGER
    COUNTER←COUNTER + 1
  Read INTEGER
```

Thus, if the first number read was 99, the loop would not be executed at all.

This concept of a priming read is extremely important in processing any serial collection of data, such as records on a file or elements in an array.

Example 1.5

Construct an algorithm to compute the tax payable on an employee's gross pay according to Table 1.1. The gross pay is in GROSS_PAY, the number of dependants is in DEPENDANTS and the result of the calculation is to be placed in TAX.

Table 1.1

Gross Wage	Fewer than 3 dependants	3 or more dependants
\$100 or less	Tax = nil	Tax = nil
\$100.01 to \$250	Tax = 15% of Gross	Tax = 10% of Gross
Over \$250	Tax = 35% of Gross	Tax = 25% of Gross

Tax Calculation

```
If GROSS_PAY≤$100
   TAX_RATE←0
else
   If GROSS_PAY≤$250
     If DEPENDANTS<3
       TAX_RATE←15
     else
       TAX_RATE←10
   else
     If DEPENDANTS<3
       TAX_RATE←35
     else
       TAX_RATE←25
TAX←GROSS_PAY*TAX_RATE/100
```

Example 1.6

Construct an algorithm to check the validity of a calendar date, that is:

(a) check that the month lies between 1 and 12,
(b) check that the day is correct for the month concerned, including a leap year check for February.

When an algorithm would become too complex or voluminous if it were constructed as a single entity, it should be decomposed into smaller logical

units. In this instance, the leap year check should be removed from the main algorithm and placed in a separate module or subroutine.

The algorithm below assumes that the date to be validated is in the format DDMMYYYY e.g. 25041983 and that the three components DD, MM and YYYY are separately accessible.

If the date is found to be correct, ERROR_FOUND will be set to a value of zero, otherwise it will be set to a value of 1.

```
Validate Date
    ERROR_FOUND←0
    Case of MM:
      4, 6, 9 or 11: LAST_DAY←30
      1, 3, 5, 7, 8, 10 or 12: LAST_DAY←31
      2: Do Check For Leap Year
        If LEAP_YEAR_FOUND = 1
          LAST_DAY←29
        else
          LAST_DAY←28
      else: ERROR_FOUND←1
      If ERROR_FOUND = 0
        If DD < 1 or DD > LAST_DAY
          ERROR_FOUND←1

Check For Leap Year
    LEAP_YEAR_FOUND←0
    Divide YYYY by 4, ignore quotient but place remainder in
        REMAINDER
    If REMAINDER = 0
      Divide YYYY by 400, ignore quotient but place remainder in
          REMAINDER
      If REMAINDER > 0
        LEAP_YEAR_FOUND←1
```

Comment: On leaving this subroutine LEAP_YEAR_FOUND contains 1 if the year was a leap year, otherwise it contains zero.

Example 1.7

Construct an algorithm to examine 20 numbers in a table, or array, and place in TALLY a count of those over 100.

This example requires the use of a form of notation which will allow the items to be referenced by their position in the table. This is normally done by the use of an index or subscript, for example:

NUMBER (4), NUMBER (16), etc. refer to specific table elements;

NUMBER (INDEX) refers to whichever table element is currently indicated by the value held in the variable INDEX.

Examine Table
TALLY←0
INDEX←1
Perform until INDEX > 20
If NUMBER (INDEX) > 100
TALLY←TALLY + 1
INDEX←INDEX + 1

Example 1.8

Construct a coinage analysis algorithm to determine the number of notes/coins of each denomination needed to be put in an employee's pay envelope so as to pay him the value of NET_PAY in as few notes/coins as possible. Assume that a table exists of the format shown containing the value in cents of each note or coin to be used and a corresponding counter to contain the number of that note or coin to be placed in the pay envelope. The coinage denominations and counters may be addressed as DENOMINATION and COUNTER respectively, with a suitable subscript. Assume that all COUNTERs start with zero as their content.

DENOMINATION	2000	1000	500	200	100	20	10	5	2	1
COUNTER	0	0	0	0	0	0	0	0	0	0
	1	2	3	4	5	6	7	8	9	10

Coinage Analysis
REMAINDER←NET_PAY
INDEX←1
Perform until REMAINDER = 0
Divide REMAINDER by DENOMINATION (INDEX) placing quotient in COUNTER (INDEX) and remainder in REMAINDER
INDEX←INDEX + 1

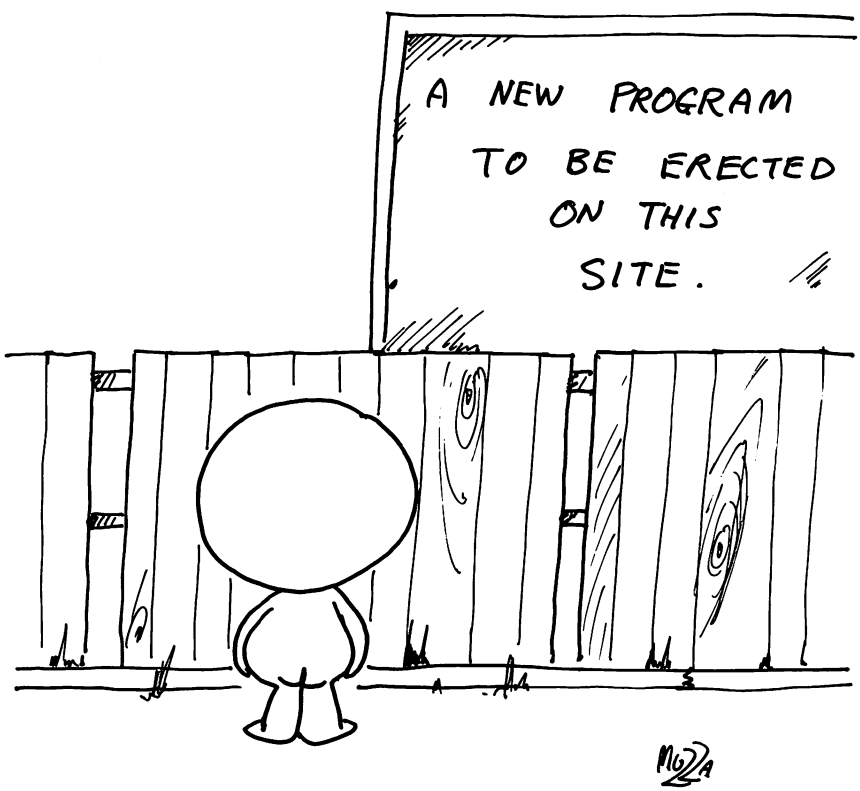
1.7 CONCLUSION

Algorithm design consists of specifying the steps required to reach the solution of a problem. If the algorithm is to lead to an elegant program, it must be specified by the use of a limited number and type of control structures, those of sequence, repetition and selection.

The use of these constructs allows any module to be constructed without resorting to branches and internal labels.

The specification of the algorithm using pseudocode rather than a diagrammatic format such as a flowchart will allow for freer expression of the tasks involved and ensure that the structured programming concepts are adhered to.

2 PROGRAM STRUCTURE



CHAPTER 2

2.1 PROGRAM STRUCTURE

In this chapter, we will consider the techniques concerned with the specification of a complete program, each portion of which will be implemented by means of the procedures for algorithm design already covered.

2.2 PROGRAM DEVELOPMENT

The steps involved in the development of any program are:

1 Define the problem.
2 Construct a program design document, or 'blueprint'.
3 Code the program in a suitable programming language.
4 Convert the coded program to a form able to be executed by the computer.
5 Test the program with data designed to detect any errors in its operation.
6 Prepare whatever additional documentation is needed to facilitate its comprehension and maintenance over a period of time.

In this chapter we will deal with the first two stages; in subsequent chapters we will cover the others. The 'blueprint' referred to in step 2 will comprise a structure diagram and its associated pseudocode algorithm.

2.3 PROBLEM DEFINITION

Before any program can be constructed, the programmer must have a clear understanding of precisely what the program must do. To this end it is useful to focus on three aspects of the program: input, processing and output.

Input

What is the structure, content and format of the data which the program is to process? Is it to be created by the program itself, is it to be input via a

VDU, is it to be read from a file on a storage device such as a disk or tape? Is the data to be in the form of a continuous stream of characters, is it to be entered item by item by an operator at a screen, is it to be read record by record from a file? Is the data to be alphanumeric or numeric, in which latter case, are the numbers to be integers or may they contain decimal fractions?

Processing

What is the algorithm required to operate on the input to produce the desired output? What calculations are needed? Does the format of the data need to change between its input form and that in which it is to be output? Do any validation checks need to be applied to the input data to ensure that it is acceptable? Will any additional data areas, such as tables or temporary storage items, need to be constructed? Do any counters or totals need to be accumulated?

Output

What is the structure, content and format of the required output? Is the data to be printed, displayed on a screen or written to a storage peripheral? In the case of printed output, what page headings or descriptive column headings are required? For VDU output, what is the format of the screen? How many lines? How many columns? For file output, what is the format of items, or fields, within records?

It is only after clarifying these considerations that the design of the program may be started.

2.4 TOP-DOWN DECOMPOSITION

The key to program construction is the decomposition of the overall procedure into modules of a single, comprehensible task and the structuring of those modules to provide an elegant implementation of the solution to the problem. Unfortunately there is no single formula which will decompose a complex program into individual tasks. The strategy, however, is one of top-down reduction of the processing until a level is reached where each of the individual processes consists of one self-contained task which is understandable and would be able to be programmed in relatively few instructions.

Without suggesting that a complete payroll procedure could be encompassed within the span of a single program, the following example should illustrate the process of decomposition. A payroll example has been chosen because of its familiarity to most readers.

The framework of the payroll situation is that an employee's work details are entered into the factory's computer system at the end of each week via a VDU, certain checks are made on the reasonableness of the data (e.g. is the employee on the payroll?, does the number of hours worked appear excessive?, etc.), the employee's pay is calculated and a coinage analysis made (i.e. how many \$20, \$10, \$5 ... 5¢, 2¢, 1¢ notes/coins are needed for his pay envelope) and the employee's history record is updated with that week's earnings added to year-to-date totals.

A list is to be printed showing the gross pay, tax and net pay for each employee and, at the end of the procedure, totals of each of these three items together with a total coinage analysis for the factory as a whole.

An examination of the overall problem would suggest the following main tasks, or modules:

1 Obtain employee's work details from VDU operator.
2 Check employee's details for reasonableness.
3 Read employee's history record from the main payroll file (i.e. the master file).
4 Calculate the pay and coinage analysis.
5 Update the employee's history record.
6 Print the employee's pay details.
7 Print factory payroll totals.

The modules required to handle these functions need not be performed in the order listed and it is clear that those numbered 1 to 6 will be repeated as many times as there are employees while 7 will be performed once only. These are considerations which will emerge as the solution progresses.

The following points would need to be covered, when considering the generation of each module.

1 Obtain employee's work details from VDU operator.
   - Obtain employee's identification (ID) number.
   - Obtain the hours worked each day of the pay period.
   - Request any additional data (e.g. bonus amounts, etc.) which may be relevant.

2 Check employee's details for reasonableness.
   - Ensure that items which are numeric by nature do not have non-numeric contents.
   - Ensure that the hours worked in any day do not exceed some pre-determined figure (e.g. 10 hours).

3 Read employee's history record and validate the ID number.
   - Using the employee's ID number obtained from the VDU, retrieve the associated history record and display the employee's name on the VDU to enable the operator to confirm that he entered the correct ID number.

4 Calculate pay and coinage analysis.
   This module is an obvious candidate for further decomposition:
   (a) Calculate pay.
      - Determine the number of hours to be paid, taking into consideration overtime allowances, etc.
      - Compute gross pay by multiplying the payable hours by the employee's hourly rate of pay according to his history record.
      - Calculate tax payable. (This would be the task of a separate module which applied the current tax formula to the gross pay already calculated.)
      - Calculate net pay by deducting tax from gross pay and adding or deducting any other relevant allowances or deductions.
   (b) Coinage analysis.
      - Determine from the net pay the number of each denomination of note and coin needed to be placed in the employee's pay envelope.
   (c) Accumulate factory totals.
      - Anticipating the needs of module 7 which must print the factory totals, these must be accumulated as each employee is dealt with.

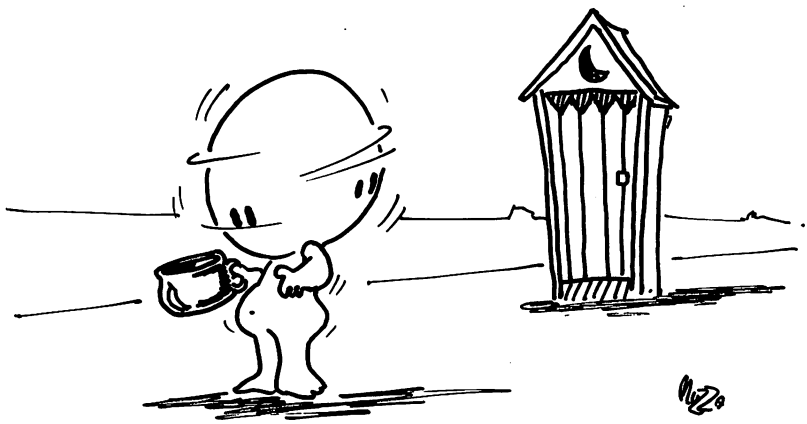
5 Update the employee's history record.
   - Add to the respective year-to-date fields in the employee's master record the gross pay, tax, net pay, etc. computed for this period.
   - Rewrite the history record to its file.

6 Print the employee's pay details.
   - Print one or more lines as required for the employee's ID, name, gross pay, tax, net pay, etc.

7 Print factory payroll totals.
   - Skip to a new page, print a suitable heading and print a summary of the gross pay, tax, net pay, etc., and a total coinage analysis for the factory as a whole.

This process of top-down decomposition must be continued until each module is clearly and simply defined and is felt to be of a size small enough to be implemented in a program in a module the purpose and method of operation of which would be apparent to a reader other than the original author.

To some extent, developing a feeling for how much or how little constitutes a comprehensible module in a coded program is an iterative procedure. It presupposes some familiarity with programming languages which may be used and hence a programmer must expect to err initially in placing too much or too little work in an individual module. Although this skill will improve as more programs are written, it will always be better to err on the side of using too many simple modules than too few modules which are then too complex.



An example of this dilemma occurs in relation to searching a table in an attempt to find an item which may or may not be in it. A COBOL programmer anticipating the use of a SEARCH instruction may regard this task as a single operation. A BASIC or FORTRAN programmer, however, may regard it as a separate module to be implemented by a FOR ... NEXT or a DO loop, respectively.

Although it is tempting to design an algorithm with a specific target language in mind, it is preferable to avoid the temptation and to describe

the process in general terms which will then be capable of implementation in any programming language.

Having decomposed the problem into a number of individual modules, two further steps are required before a program may be coded. The first is the design of the structure of the program, that is, the relationship of the modules with one another in terms of their hierarchy and their sequence of operation. This will be dealt with in the following section of this chapter. The second step, to be covered in Chapter 3, is the concept of inter-module communication.

It may be profitable at this point to reconcile any seeming contradiction between the assertion in Chapter 1 that pseudocode is to be preferred over diagrammatic methods of presenting algorithms and the return in this chapter to representing programs diagrammatically. The purpose of a structure diagram is two-fold:

(a) the main purpose is to enable the programmer to arrive at an acceptable structure for a program by identifying its component modules and their mutual interaction, and

(b) a by-product of the diagram is the semi-automatic generation of a program algorithm by implementing in pseudocode the control structures indicated in the structure diagram.

The strength of structure diagrams lies in the fact that by concentrating on structure, logic is derived automatically whereas the weakness of flowcharts lies in the fact that by concentrating on logic, structure is not derived at all.

2.5 STRUCTURE DIAGRAMS

Of the program design tools currently available, the best for determining and illustrating program structure is that of the structure diagram.

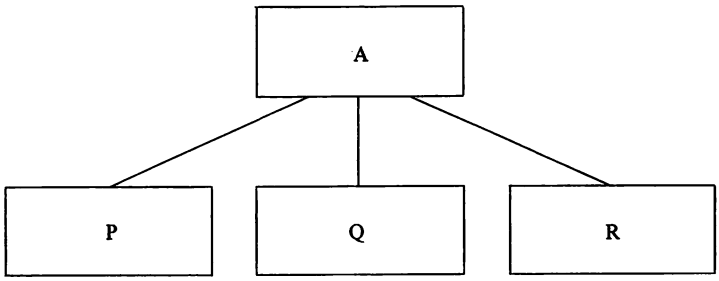
A structure diagram depicts a program's structure in terms of its hierarchy and operating sequence. It does not attempt to displace the design of individual algorithms as a description of the logic involved in the solution of the problem.

The diagram comprises a tree-like structure with the program name in the top-most box and each module in a box subordinate to its 'manager' or 'controller'. A repetition or loop construct is indicated by an arrow encompassing the lines of control drawn from a controller to its subordinates. A selection construct is obtained by the use of the diamond symbol traditionally used to represent decisions in flowcharts. Apart from

these symbols, the only rules of interpretation are that hierarchy is represented from top to bottom and sequence is read from left to right.

Figures 2.1, 2.2, 2.3, 2.4 and 2.5 illustrate the relationship between the structure of an algorithm and the method of its implementation in a program.

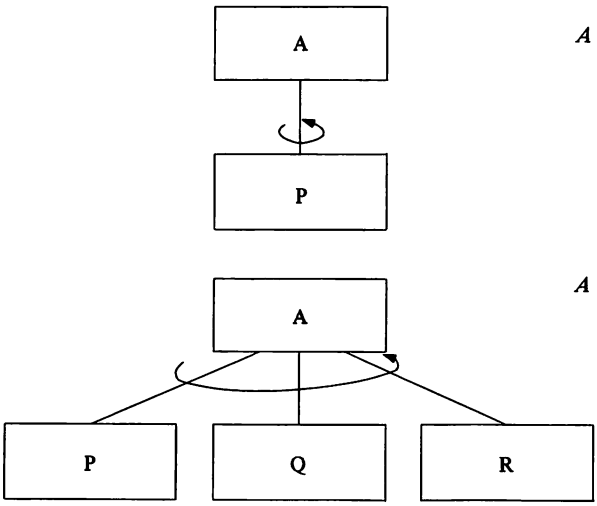
Sequence



A
Do P
Do Q
Do R

Figure 2.1

Repetition

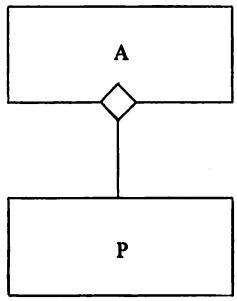


A
Perform until condition
Do P

A
Perform until condition
Do P
Do Q
Do R

Figure 2.2

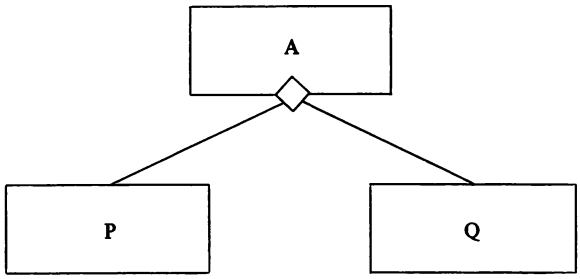
Selection



A

If Condition
Do P

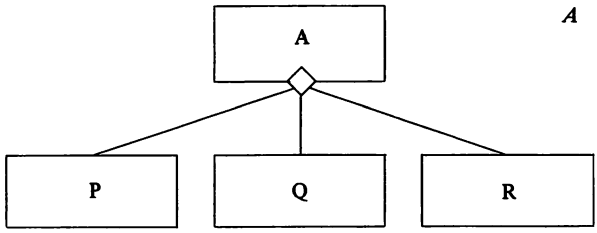
Figure 2.3



A

If condition is true
Do P
else
Do Q

Figure 2.4



A

Case of condition
Condition–1: Do P
Condition–2: Do Q
Condition–3: Do R

Figure 2.5

2.6 A GENERAL EXAMPLE

As a general example, Figure 2.6 represents the structure of an algorithm consisting of a number of modules arranged hierarchically. Such a structure may depict the solution to a complete program, as is the case in Figure 2.6, or of any portion of a program.

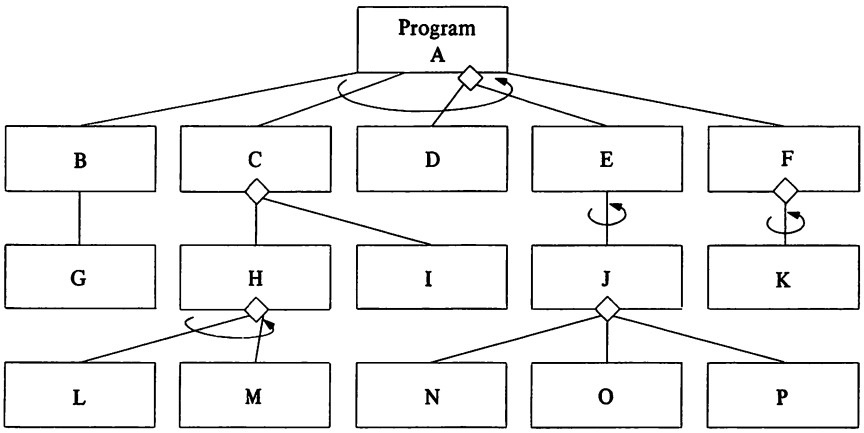


Figure 2.6

The algorithm required to implement the control structure indicated by Figure 2.6 would be:

Program A
Do B
Perform until condition-a
Do C
If condition-b is true
Do D
else
Do E
Do F
Stop

B

```
.
.
.
Do G
.
.
.
```

C

```
.
.
.
If condition-c is true
    Do H
else
    Do I
.
.
.
```

E

```
.
.
.
Perform until condition-e
    .
    .
    .
    Do J
    .
    .
    .

.
.
.
```

F

```
.
.
.
If condition-f is true
    Perform until condition-g
        Do K
.
.
.
```

H

.
.
.

If condition–h is true
  Perform until condition –1m is true
    Do L
    Do M

.
.
.

J

.
.
.

Case of condition–j
  j–1: **Do** N
  j–2: **Do** O
  j–3: **Do** P

The principles of constructing structure diagrams and their associated algorithms are illustrated by Examples 2.1, 2.2 and 2.3.

Example 2.1 Motor Vehicle Enquiries

A program is required to retrieve motor vehicle registration records from a file upon receipt of a request from an operator at a VDU. The operator will supply a vehicle registration number and the program will display the details of the vehicle and its owner. An error message will be displayed if the program is unable to locate the vehicle's record.

The modules required to implement the solution are:

1 Accept the vehicle registration number from the VDU operator.
2 Using the registration number, attempt to retrieve the vehicle's record from its file.
3 A successful retrieval will allow the details of the vehicle to be displayed on the screen.
4 An unsuccessful attempt at retrieval will indicate the absence of the vehicle from the file and a suitable error message may be displayed.

On the assumption that the procedure continues until the operator initiates an abort-and-log-off procedure, Figure 2.7 illustrates a simple structure diagram for this program.

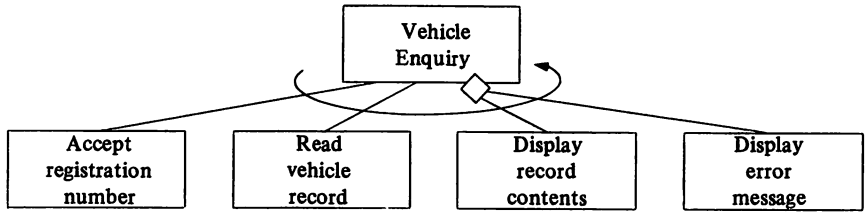


Figure 2.7

An algorithm for the implementation of this program would be constructed as:

Motor Vehicle Enquiries
Perform until operator logs off
Accept vehicle registration number
Retrieve vehicle record from file
If record is successfully read
Display record contents on VDU
else
Display error message

Example 2.2 Motor Vehicle Enquiries (continued)

The situation described in Example 2.1 is extended to cover the case in which a vehicle registration record, if retrieved successfully, may contain registration numbers of other vehicles belonging to the owner of the vehicle in the original enquiry. If this is the case, the VDU operator is to be given the option to display the details of these additional vehicles.

This will involve the addition of further processing modules:

5 Following a successful retrieval of a vehicle's record, if the vehicle's owner has other vehicles registered in his or her name the VDU operator must be asked whether those details are to be displayed.

6 For each additional vehicle to be displayed (if any) there will need to be:

(a) Retrieval of the vehicle's record.

(b) Display of the record details.

Figure 2.8 shows an amended structure diagram to reflect these additions.

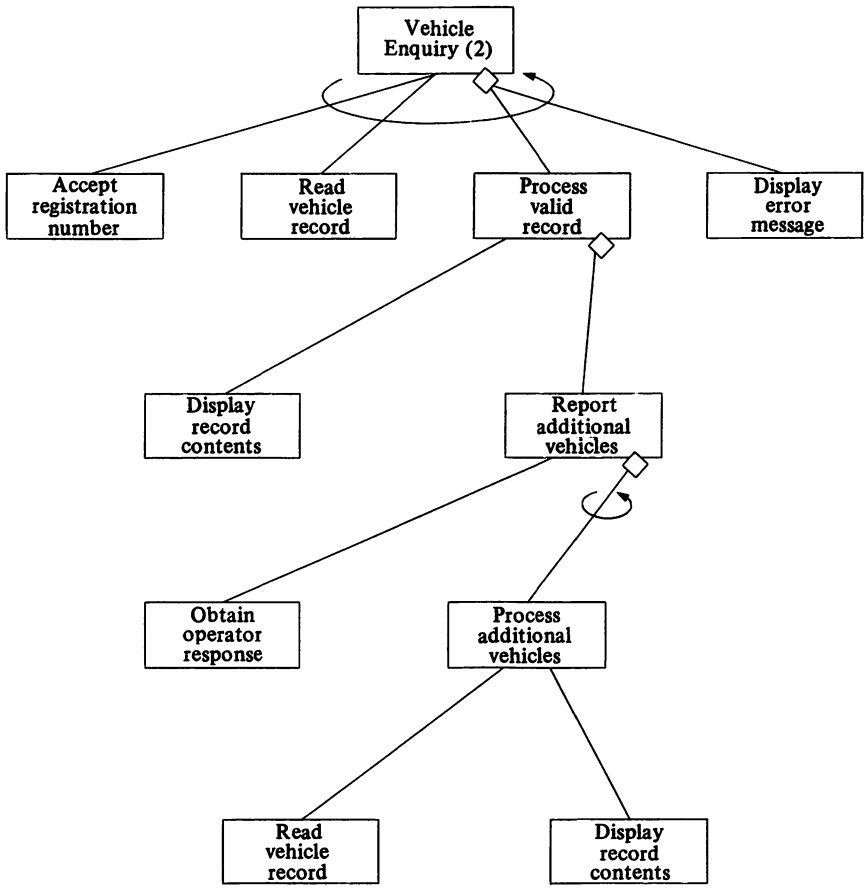


Figure 2.8

The algorithm to implement the extended version of the program shown in Figure 2.8 would be:

Motor Vehicle Enquiries (2)
Perform until operator logs off
    Accept vehicle registration number
* Retrieve vehicle record from file
    If record is successfully read
        Do Process Valid Record
    else
        Display error message

Process Valid Record
* Display record contents
    If owner has additional registered vehicles
        Ask operator if further details are required
        If operator requests further details
            Perform until all vehicles are reported
*   Retrieve vehicle record from file
*   Display record contents

**Note* that in this algorithm, the operations 'Retrieve vehicle record from file' and 'Display record contents' each appear in two places. Suspecting that these may each involve subsequent duplication of coding, we may have the foresight to place these operations in modules of their own and replace the current statements respectively by:

Do Retrieve Vehicle Record, and
Do Display Record Contents

Example 2.3 Factory Payroll

Before drawing a structure diagram for the payroll example used previously in this chapter, consider one modification to the algorithm given:

> As module 3, 'Read employee's history record', is to display the employee's name on the VDU for operator confirmation, it would be preferable if this operation were performed as a subordinate of module 2.

A tentative structure diagram for the problem is shown in Figure 2.9.

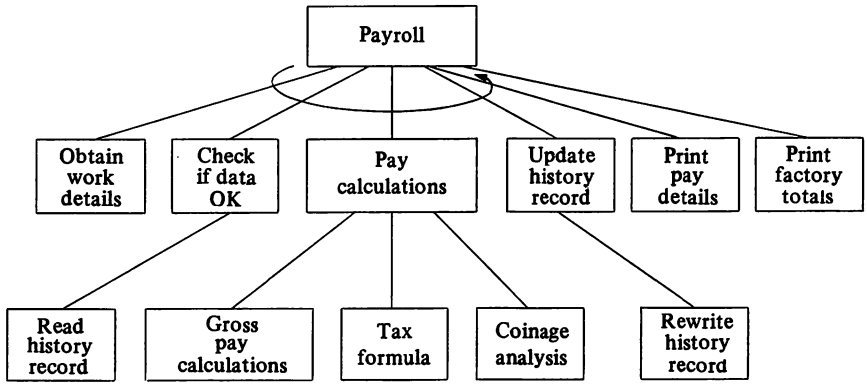


Figure 2.9

Further consideration of the diagram in Figure 2.9 reveals a weakness: there appears to be no mechanism for preventing the algorithm from proceeding to the pay calculations if the data entered via the VDU is incorrect. A better structure for the initial procedure in the algorithm is that shown in Figure 2.10. There the operator can be requested to supply an employee's details as often as is needed until the correct data is obtained.

To avoid making the overall structure diagram too complex, a separate diagram may be drawn for any portion of the problem. Hence, Figure 2.10 would be an appendix to the main diagram.

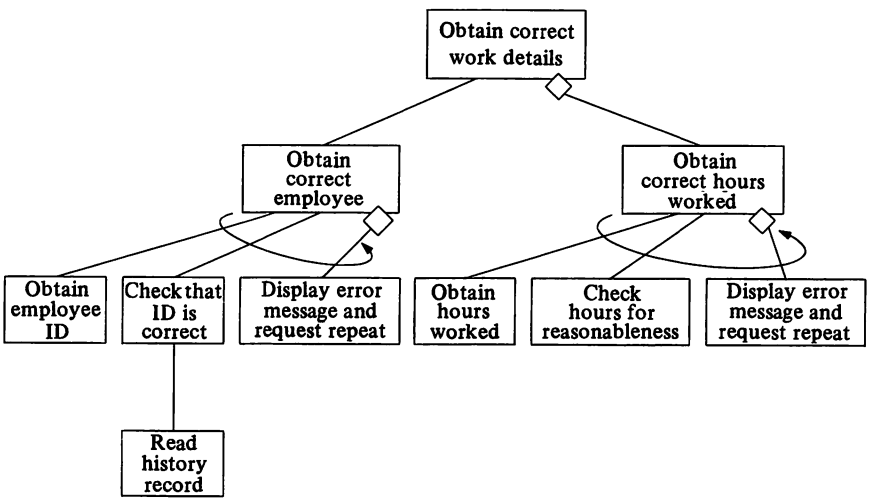


Figure 2.10

One last requirement remains to be specified before an algorithm can be constructed, and that is an answer to the question: how does the procedure terminate? Let us assume that the termination of processing occurs when the VDU operator inputs a response to the prompt for a further employee's details which indicates that no more employees remain to be processed. This could be by a blank response or a predetermined dummy employee ID as a response to the request for the next employee, for example, 99999 or -1.

This will mean that the performance of all subsequent modules will be conditional upon the operator not entering the end-of-data signal in response to the Employee ID prompt.

This is now a serial processing environment and will benefit from a 'priming read' as described in Chapter 1. Figure 2.11 shows the final structure diagram for the solution of the problem.

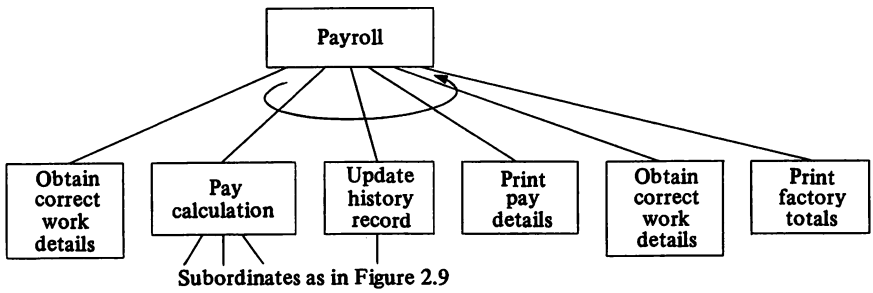


Figure 2.11

Note that the modules 'Obtain correct work details' need no elaboration on their subordinates as these are shown in an appendix as illustrated in Figure 2.10.

2.7 SKELETON PSEUDOCODE

An algorithm may now be constructed, in skeleton form (or pseudocode) to implement a solution to the problem:

Payroll
Perform until correct details are obtained or end-of-data is signalled
Do Obtain Correct Work Details

```
    Perform until end of data is signalled
        Do Pay Calculation
        Do Update History Record
        Do Print Pay Details
        Perform until correct details are obtained or end-of-data is
                signalled
            Do Obtain Correct Work Details

    Do Print Factory Totals
    Stop

Obtain Correct Work Details
    Perform until correct employee ID is obtained or end-of-data
            signal is entered
        Do Obtain Correct Employee

    If end-of-data has not been signalled
        Perform until correct hours details are obtained
            Do Obtain Hours Worked

Obtain Correct Employee
    Accept employee ID from VDU
    If end-of-data signal is not entered
        Do Check Employee ID
        If employee ID is not valid
            Display error message
            Request operator to repeat operation

Check Employee ID
    Read Employee History Record using EMPLOYEE_ID as a key
    If record is read successfully
        Signal valid employee ID
    else
        Signal invalid employee ID

Obtain Hours Worked
    Accept details of hours worked from VDU
    Check details against criteria for reasonableness
    If no errors are found
        Signal correct details
    else
        Display error message
        Request operator to repeat operation
```

Pay Calculation
Compute payable hours worked
Compute gross pay = payable hours × hourly rate of pay
Do Tax Formula (not specified here)
Compute net pay = gross pay − tax
Do Coinage Analysis (see Chapter 1, Example 1.8)
Add gross pay, tax, net pay and coinage analysis to accumulated totals for factory

Update History Record
Add gross pay, tax and net pay to respective year-to-date fields in Employee History Record
Rewrite Employee History Record to its file

Print Pay Details
Print a line of pay details comprising employee ID, name, gross pay, tax, net pay

Print Factory Totals
Print accumulated totals for gross pay, tax, net pay and coinage analysis.

2.8 MANAGING vs WORKING

As we progress from the top to the bottom of the hierarchical structure of a program, we should expect to see a change in the nature of the instructions contained in the modules. The higher level modules should contain mostly 'managerial' instructions, that is, decision-making and directions to subordinates. As we move down the hierarchy, gradually more and more 'working' instructions begin to creep into the modules in the nature of calculations, movement of data, and so on. Avoid introducing working instructions too early in the hierarchy of any program. Certainly the first level and possibly the second level modules within a program should consist only of **Do**, **Perform** and **If** statements, as shown, for example, in Figure 2.12.

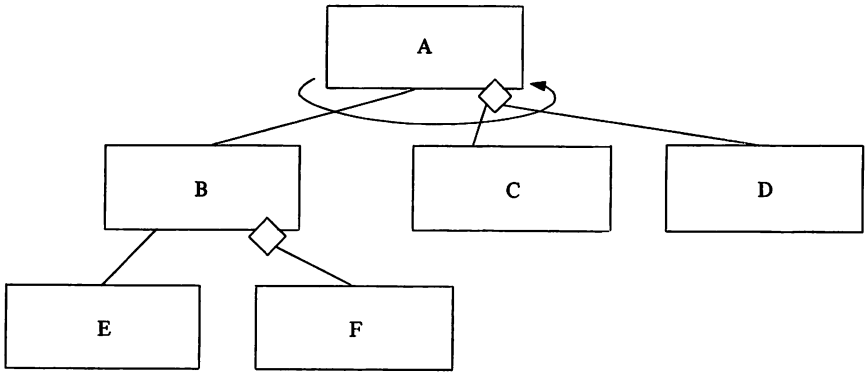


Figure 2.12

A (1st level)
    Perform until condition
        Do B
        If condition
            Do C
        else
            Do D
    Stop

B (2nd level — as for C and D)
    Do E
        ⋮ } Perhaps some minor work done
    If condition
        Do F

E (lower levels)
    ⋮ } mostly working instructions

F
    ⋮

In this context, we should also avoid placing large groups of working instructions in a decision-making framework. It is much cleaner to remove the workers and place them in a module of their own, for example:

```
If condition
   statement
      .
      .
      .
      .   (group A)
      .
      .
      .

   statement
else
   statement
      .
      .
      .
      .   (group B)
      .
      .
      .

   statement
```

POOR STRUCTURE

```
If condition
   Do A
else
   Do B

A
   statement
      .
      .   (group A)
      .

B
   statement
      .
      .   (group B)
      .
```

BETTER STRUCTURE

2.9 DECISION-MAKING STRUCTURE

A structure diagram gives a programmer an opportunity to review the decision-making structure of a program and to rearrange it, if necessary, before its implementation in code.

Two areas need to be considered in examining the control structure of any program:

(a) Aim to eliminate repetitive evaluations of any one condition, as shown, for example, in Figure 2.13.

Poor:

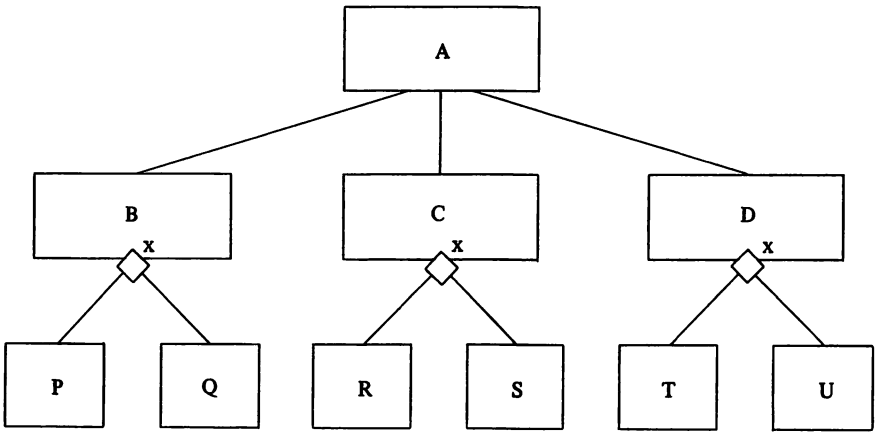


Figure 2.13

Here, Condition x is tested in three separate sections of the program. In Figure 2.14, it is tested only once.

Better:

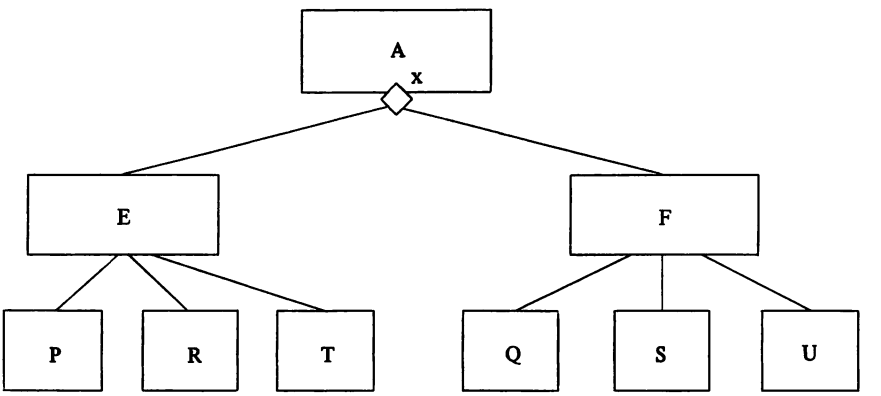


Figure 2.14

Note that the same working modules P, Q, R, S, T, and U are required in each case; all that differs is the managerial control structure. In practical terms, the difference lies between the following strategies:

Strategy 1 (*Poor*)

Calculate Pay
Do Compute Gross
Do Compute Allowance
Do Compute Bonus

Compute Gross
If permanent employee
Do Compute Gross for Permanent Staff
else
Do Compute Gross for Temporary Staff

Compute Allowance
If permanent employee
Do Compute Allowance for Permanent Staff
else
Do Compute Allowance for Temporary Staff

Compute Bonus
If permanent employee
Do Compute Bonus for Permanent Staff
else
Do Compute Bonus for Temporary Staff

Strategy 2 (*Better*)

Calculate Pay
If permanent employee
Do Compute Permanent Employee Pay
else
Do Compute Temporary Employee Pay

Compute Permanent Employee Pay
Do Compute Gross for Permanent Staff
Do Compute Allowance for Permanent Staff
Do Compute Bonus for Permanent Staff

Compute Temporary Employee Pay
Do Compute Gross for Temporary Staff
Do Compute Allowance for Temporary Staff
Do Compute Bonus for Temporary Staff

In this example the nature and method of operation of the six compute modules are unaffected by the choice of either strategy. What is achieved is a neater and more maintainable control structure because of the centralization of the testing mechanism.

(b) Ensure that all processing which depends on the result of a decision is subordinate to the module in which that decision is made; see, for example, Figure 2.15.

Poor:

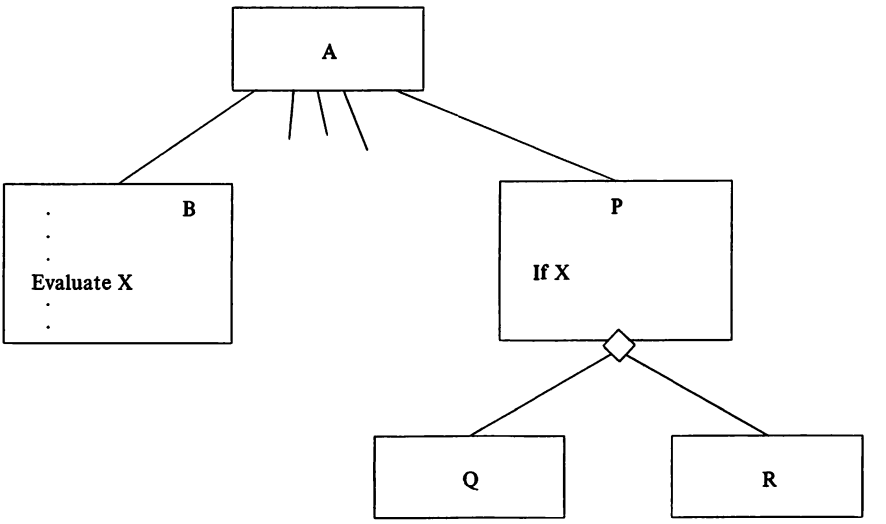


Figure 2.15

In Figure 2.15, Condition x is evaluated in module B but the result of that evaluation is later tested in module P which may be far removed in the program from the source of the decision. Hence, a maintenance programmer amending B may fail to see the implication of some change which he makes to the evaluation of Condition x and, conversely, a programmer looking at module P may fail to understand how the condition being tested was originally evaluated. In Figure 2.16, the results of the decision based on the evaluation of Condition x are made subordinate to the module (B) in which the evaluation is made and consequently can be seen related together.

Better:

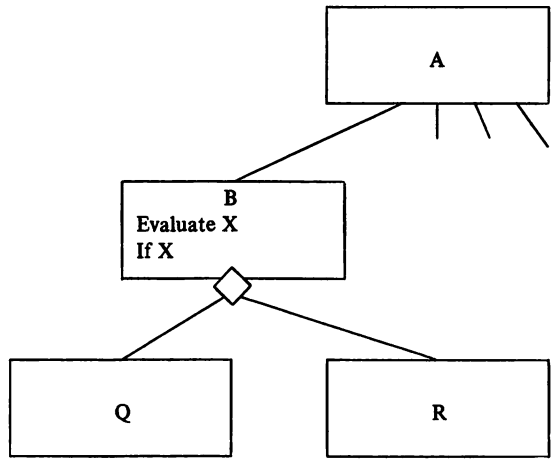


Figure 2.16

2.10 CONCLUSION

Given that one of the most important criteria in the assessment of any program is that of its module structure, it is extremely important to spend time in the program design stage to ensure that this criterion is met.

The steps involved are:

1 Decompose the problem, working from the top down, into a hierarchy of modules or routines each of which performs a single task.

2 Draw a structure diagram which represents the hierarchy from top to bottom and indicates the sequence of operations from left to right.

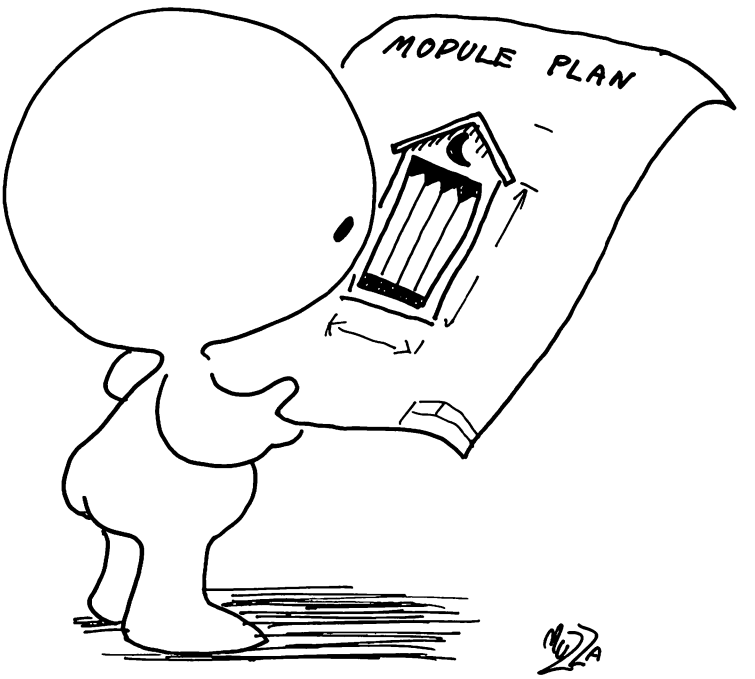
3 Indicate repetition and decision constructs within the diagram using appropriate symbols.

4 Appraise the decision-making and control structure and re-draft the hierarchy if required.

5 Translate the structure diagram into pseudocode by a direct correspondence of pseudocode control mechanisms with those depicted in the structure diagram.

Remember that the whole procedure is iterative. Do not expect to arrive at the appropriate algorithm structure on your first attempt. Be continuously prepared to recast the hierarchy or the sequence of operations as the algorithm develops and be prepared to regroup or subdivide functions where a more effective structure would result.

3 MODULE DESIGN



CHAPTER 3

3.1 MODULE DESIGN

We have examined the means for specifying the procedure performed by individual program modules and their subsequent grouping to develop a program as an entity. This chapter is devoted to a consideration of the functions carried out by modules and the means by which they may communicate with each other.

3.2 INTER-MODULE COMMUNICATION

Given that each module of a program will perform a function the results of which will affect some or all of the other modules within the program, there must be some means by which those results may be made known to the other interested modules. There are basically two methods of module-to-module communication:

(a) communication by data items, and
(b) communication by status items.

Consider the case, discussed in Chapter 1, Example 1.6, of a module designed to check the validity of a calendar date. To communicate on a two-way basis with that module, two items must be specified:

(a) the data area holding the date to be validated, and
(b) some means of indicating whether or not the date was found to be valid.

Items used thus for communicating between various parts of a program are referred to as 'parameters' or 'arguments'.

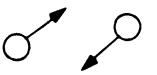
Parameters may have one of three functions:

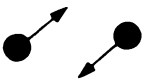
(a) they may serve to pass information to a performed, or subordinate, module which may be referred to as a 'subroutine',
(b) they may pass information back from a subroutine to its performer, or caller, or

(c) they may fulfil a two-way role and contain information which is passed to the subroutine, amended in some fashion and then passed back again to the caller.

Hence in the date validation subroutine, the date would be passed to the performed validation module and the OK/not-OK answer passed back to the calling module.

Such communication parameters may be incorporated into a structure chart by the use of two symbols:

data parameters 

status parameters 

A structure chart incorporating the data validation procedure as part of its operations is shown in Figure 3.1.

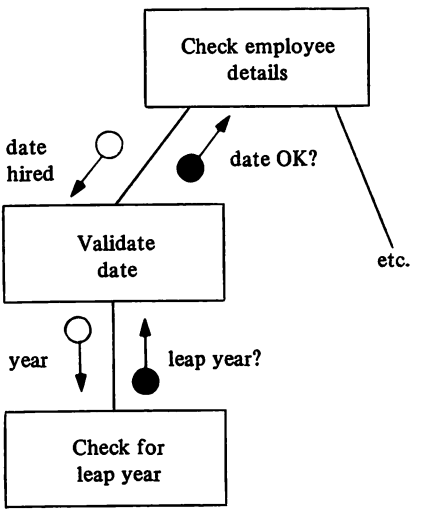


Figure 3.1

The chart indicates that the subroutine was to be used to check the validity of the date on which the employee was hired. It, in turn, calls a subroutine to which it passes the year and receives an indication of whether or not it is a leap year.

When designing algorithms and, ultimately, when writing programs, avoid using data parameters to also indicate status. In the example above, it would be possible for the subroutine to replace the date to be checked with a value of, say, zero if the date was found to be invalid. Subsequently, on return from the validation subroutine, the calling module could examine the content of the date parameter and, if a zero value was found, take this to indicate that the date was not valid.

Avoid this procedure! If a parameter is a data item by nature, let it remain a data item. Do not confuse the reader of your program by using it for another purpose entirely. In addition to the possibility of confusing the reader with, say, a date set to zero there is the more urgent danger that if the program is amended at some later time the amending programmer will be unaware that the date may have been set to a zero value and may rely on the original contents still being present in that item.

A status parameter should ideally contain only one of two possible values, True or False. Some programming languages reserve a separate class of variables, called logical or Boolean variables, especially for this purpose. Where such a feature is not included in the syntax of a language, the programmer should endeavour to set aside a number of variables expressly to be used as status parameters and to create two constants which represent respectively the logical value of True and False, for example: True may be the value of 1 or "T" or "Y" (for Yes) and False may be 0 or "F" or "N" (for No). The actual values used should be assigned to variables called TRUE and FALSE (or similar), and it is these symbolic

names which will become the subject of instructions not the specific values, that is:

If Date-OK = TRUE
  Do ... procedure ...
not
If Date-OK = 1 (or "T", etc.)
  Do ... procedure ...

With this in mind, it would have been preferable to have written Example 1.6 in Chapter 1 with the variables ERROR_FOUND and LEAP_YEAR_FOUND (which are status parameters by nature) set to values of TRUE/FALSE rather than 1 and zero as they now appear.

As an aside, it should be noted that there are two parameters returned from every performance of a module which is to read a record from a file and, in many cases, one which is to accept input from a screen. The more obvious of these is the data item sought; the other is a status variable which indicates whether or not the input operation was successful, that is, whether the record was unable to be read because of an end-of-file condition being reached or an invalid key (one for which no record exists on the file) being supplied to the read operation or the screen operator giving a null response to a prompt for data.

Hence, we should expect automatically to depict such an operation as shown in Figure 3.2.

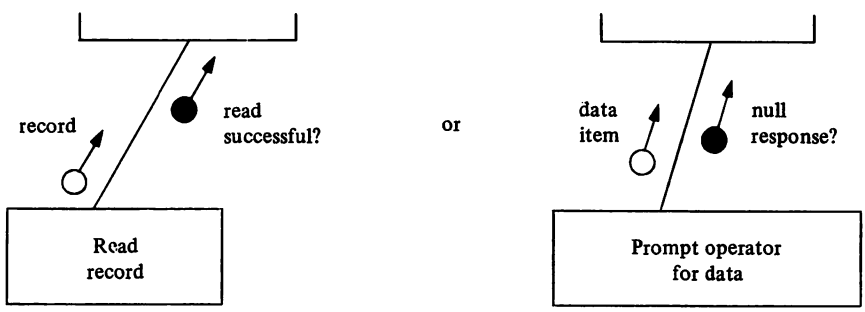


Figure 3.2

3.3 MODULE STRENGTH OR COHESION

Cohesion, or module strength, is a measure of the extent to which the statements in a particular module belong together. When asked for a

reason why a group of statements appear in a single module, there are several possible justifications which could be advanced, for example:

(a) the statements are all necessary to perform one particular task within the overall program structure;

(b) the statements encompass a variety of small unrelated tasks but, as these must be performed at the same time, for convenience they have all been included in a single module;

(c) the statements perform two or more related tasks which are always performed together and therefore have been placed in the same module.

These are all reasons for the inclusion of a number of statements in a single program module. The reasons become weaker, however, as we move from (a) to (c).

The prime reason for the existence of any module in a program should be that it performs a single logical task. The data validation routine already discussed falls into this category. It emphasizes the separation of tasks by creating a subroutine to check for the occurrence of a leap year.

An example of the cohesion referred to as type (b) is the collection of tasks which one often needs to execute at the beginning of a program's operation, for example, linking data to specific files, setting counters and totals to initial values, heading pages ready for output, and so on. Such actions are often placed together in an 'initialization routine' and have no real connection with one another other than that they all must be performed at that point of the program's execution.

Consider, as a further example, the case of a program which is written to run at the end of each month and print statements of overdue accounts for customers. Because it is run monthly, each run necessitates the ageing of customers' debts by one period, that is, current debts become 30 days outstanding, 30 days become 60 days, and so on. Because of this, it may be tempting to include the ageing procedure in the same module as the reading of the customer's record, an example of type (c) cohesion.

The problem arises here when circumstances change and the program has to be run more often than once a month. The maintenance programmer is now faced with two choices. He may remove the ageing process and place it in a separate module which is executed only in the end-of-month run, as shown in Figure 3.3.

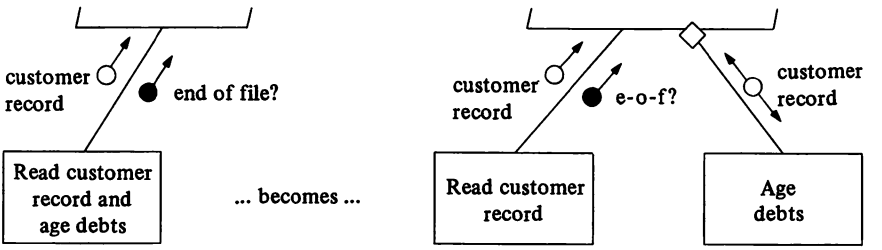


Figure 3.3

He may choose to retain the single module as before and pass to it a status item which indicates whether or not the debts are to be aged this run, as shown in Figure 3.4.

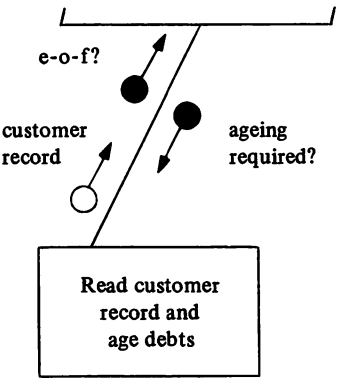


Figure 3.4

Of these two alternatives, the first (Figure 3.3) is to be preferred. Be very wary of passing status parameters downwards. This is an admission that the subordinate module performs more than one task (otherwise the status item would not be required) and that it would be better organized by being split into separate modules each with its own function.

3.4 MODULE COUPLING

Coupling is a measure of the interdependence between sections of a program or between a section of procedural code and one or more data items.

Consider the following example.

As part of its overall operations, a certain program is required to perform the following operations:

- Read records from a file;
- Validate certain data items in each record according to criteria for acceptance, one such item being a product ID number which is checked against valid product numbers held in a table which also holds the name of the product associated with each ID number;
- Print either an error message (in the case of an invalid record) or a report line containing the contents of each valid record with product ID numbers accompanied by their associated product name.

The structure diagram, in part, for the program would appear as shown in Figure 3.5.

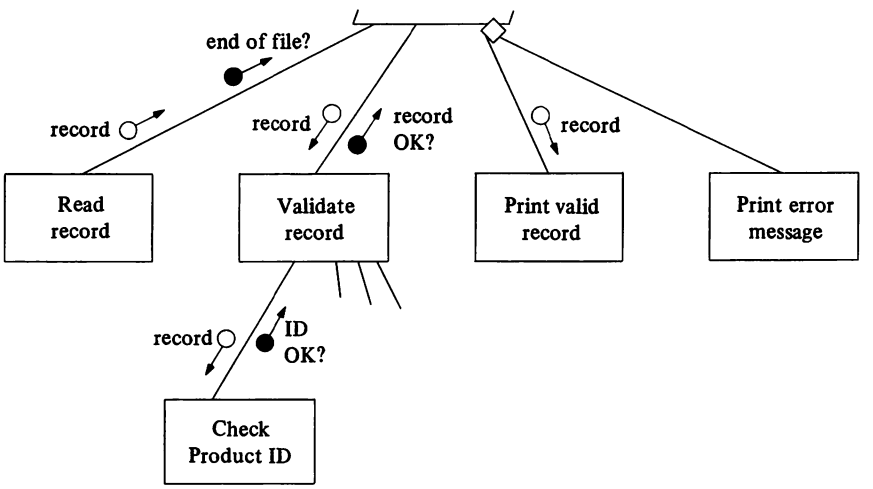


Figure 3.5

The programmer, when designing the algorithm (and later when writing the code) for the Validate Record procedure, specifies the procedure for checking the product ID number against those in the table as follows:

Check Product ID Number

TABLE_INDEX←1

ID_FOUND←False (a status item)

```
Perform until ID_FOUND = True
      or TABLE_INDEX > Size of the table
   If Product ID in input record = Product ID (TABLE_INDEX)
      ID_FOUND←True
   else
      TABLE_INDEX←TABLE_INDEX + 1
```

Later in the program, when about to specify the procedure for the "Print Valid Record" module, the programmer realizes that there is again the requirement to match the product ID number against its counterpart in the table in order to obtain the product name which also resides in the table. It is at this point that the programmer realizes that this task has already been accomplished in the Check Product ID Number module. On leaving that module, after locating the product referred to in the record being processed, the table subscript TABLE_INDEX was left holding the number of the element in the table at which a match was found. Given that nothing else in the program has since disturbed the contents of TABLE_INDEX, its value may be used to extract the name of the product associated with the ID number. Hence the programmer specifies:

```
Print Valid Record
   .
   .
   .
Place Product Name (TABLE_INDEX) in line to print
   .
   .
   .
```

These procedures, although containing no errors of logic, have placed many restrictions on the future maintainability of that program because of several instances of coupling:

(a) Although there is no hint of it in the structure chart above, the Print Valid Record module is coupled with the Check Product ID Number module via the subscript variable TABLE_INDEX. If a subsequent amendment to the program causes the value in TABLE_INDEX to change between leaving Check Product ID Number and entering Print Valid Record, then that latter module will no longer print the correct name for any given product.

One remedy for this would be to pass the position in the table at which the product ID was found as a formal parameter returned from the Check Product ID Number module. By making it a specified parameter and indicating it as such on the structure

chart, there is less chance that a subsequent maintenance programmer will fail to recognize its significance. The passing of this parameter could be achieved by leaving TABLE_INDEX unaltered until Print Valid Record had executed or, better still, by placing the position indicated by TABLE_INDEX in a separate variable and thus releasing TABLE_INDEX for other purposes if required. The structure chart would then appear as that in Figure 3.6.

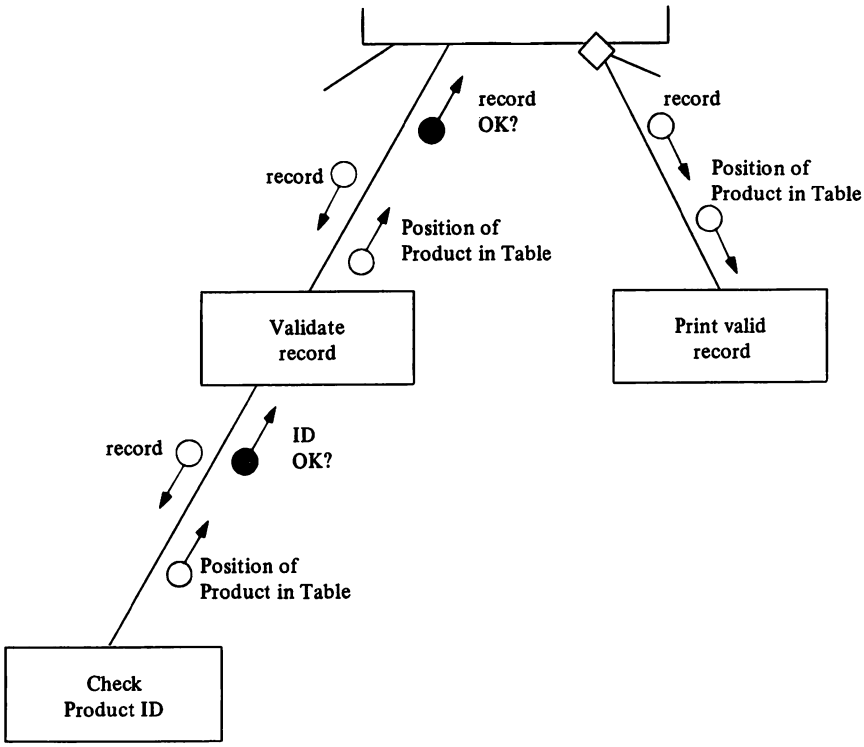


Figure 3.6

(i) The suggested solution to the problem discussed in (a) has introduced a further element of coupling! The operation of the Print Valid Record module now depends on the fact that Check Product ID Number uses a table as its means of validating the products. The range of products may expand to the point at which an internally stored table would be too large to tolerate in the program. The product validation may then be done via

an access to a disk file to attempt to retrieve a record for the product ID number to be validated, the success or otherwise of that operation indicating whether or not the product is legitimate. In such a case, the table subscript passed as a parameter is now meaningless.

A better solution would be to pass the product name itself as the formal parameter rather than merely an indirect reference to it. This would achieve the objective of removing the necessity for the Print Valid Record module having once again to search for the product in order to know its name, and it would also make the operation of that module totally independent of whatever method the validation module needed to ascertain the validity of the product ID number.

The structure chart is now as shown in Figure 3.7.

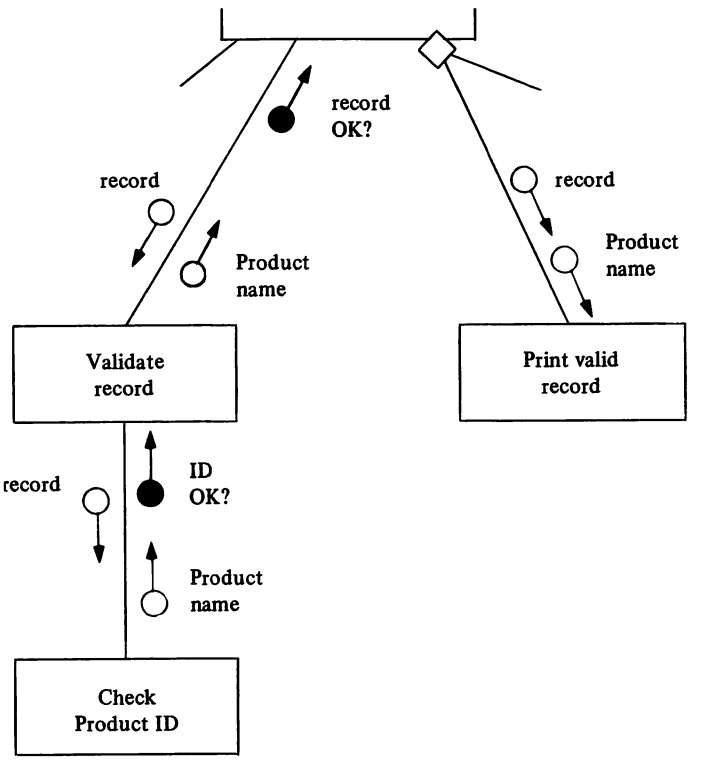


Figure 3.7

Note: The question of what the Product Name parameter contains if the validation module finds the product invalid will be considered later in this chapter.

(b) The original solution and the suggested amendments so far still suffer from a further coupling. The Check Product ID Number module is written to validate the product ID number in the record read from the file referred to in the original specification.

It is conceivable that future enhancements to the program may require a product ID number from a different source to be subjected to the same validation procedure. As it currently stands, the Check Product ID Number routine would be unable to perform the second, or subsequent, check because the statement

```
.
.
.
If Product ID in input record
    = Product ID (TABLE_INDEX)
.
.
.
```

restricts the routine to operate only on the specified product ID.

A better solution would have been to pass the product ID number to be validated as a separate independent parameter to that module, for example:

```
.
.
.
PRODUCT_TO_CHECK←Product ID in input record
Do Check Product ID Number
.
.
.
```

```
Check Product ID Number
    TABLE_INDEX←1
    ID_FOUND←False
    Perform until ID_FOUND = True
            or TABLE_INDEX > Size of the table
      If PRODUCT_TO_CHECK = Product ID (TABLE_INDEX)
        ID_FOUND←True
        PRODUCT_NAME←Product Name (TABLE_INDEX)
      else
        TABLE_INDEX←TABLE_INDEX + 1
```

The structure chart, finally, appears as that shown in Figure 3.8:

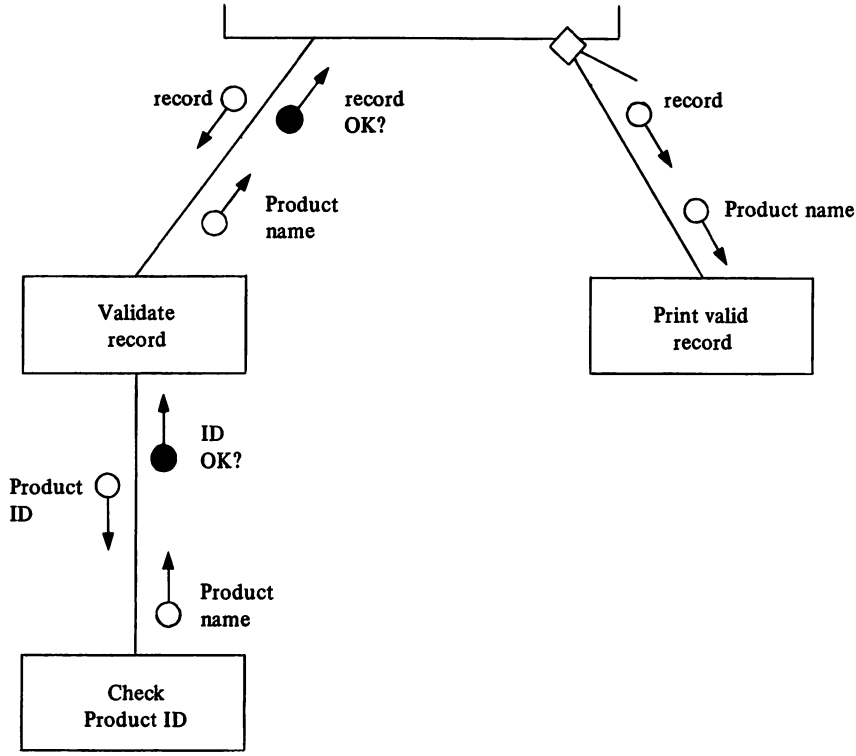


Figure 3.8

One of the aims, then, in the design of any program module is to uncouple it from as much of its surroundings as possible. This is done by:

(a) passing as much of its required data as is possible by means of independent formal parameters rather than by designing the module to operate on data which is merely 'lying around for general use', and

(b) writing the module as a self-contained unit which accepts any given parameters, operates on them without reliance on the functions of other sections of the program, and passes the results of its operation back to its caller again via a formal independent parameter.

This procedure enables the function of any program module to be specified without reference to the manner in which the function is actually effected; for example, for the validation module we have been examining, a mini-specification may be written as:

Module Name:	Check Product ID Number
Function:	To ascertain whether or not a given product ID number is a legitimate code.
Input parameter:	PRODUCT_TO_CHECK Contains product ID number to be validated
Output parameter:	ID_FOUND Contains a status of True if the product ID is valid, otherwise contains a status of False
	PRODUCT_NAME Contains the name associated with a valid product ID, otherwise ...

Notes:

(a) This specification is all one needs to make use of that module, despite no clue being given on how the module operates.

(b) A decision must be made at this point (and inserted in the specification) on what the contents of PRODUCT_NAME will be in the case of an invalid product; for example, contents may be blank, all asterisks, the words 'INVALID PRODUCT', and so on. Some definite content, however, *must* be specified.

3.5 INFORMATION HIDING

One of the major concepts of software engineering is that of 'hidden intelligence' or 'information hiding'. It implies an approach to program design exemplified by the points considered in the previous section. Essentially, it states that the important feature of any program module is *what* it does not *how* it does it. Modules should be defined in terms of the function they perform and the means by which the rest of the program communicates with them, that is, their parameters. This procedure then leaves the programmer implementing the module free to choose whatever method of operation is thought best. Indeed, in the interests of efficiency or practicality (as seen previously in the case of the product ID validation), the procedure of the algorithm may be drastically changed from time to time. As long as the performed function and the communication parameters remain the same, however, there is no need for the rest of the program to be aware of any changes in procedure.

This concept imposes two restrictions on the programmer:

(a) The task module must not depend on the operation of any other section of the program.

The module performing any task must be entirely self-contained and must not rely on any feature of the program's environment outside itself, that is, any other procedural module or data which are not declared explicitly as part of the module specification.

For example, if the product ID validation module was obliged to fill PRODUCT_NAME with blanks in the case of an invalid product, it would be wrong of it to take no action at all in such a case merely because the programmer noticed that the operation of another part of the program happened to blank-fill PRODUCT_NAME before executing the validation module. The argument that PRODUCT_NAME will therefore contain blanks in the case of an invalid product simply because the validation module has not then put anything else in that data item is destroyed as soon as that other program operation, which has no connection with the validation routine, has its mode of operation changed for some reason and no longer blank-fills PRODUCT_NAME.

(b) The task module must not have any side effects on any other section of the program.

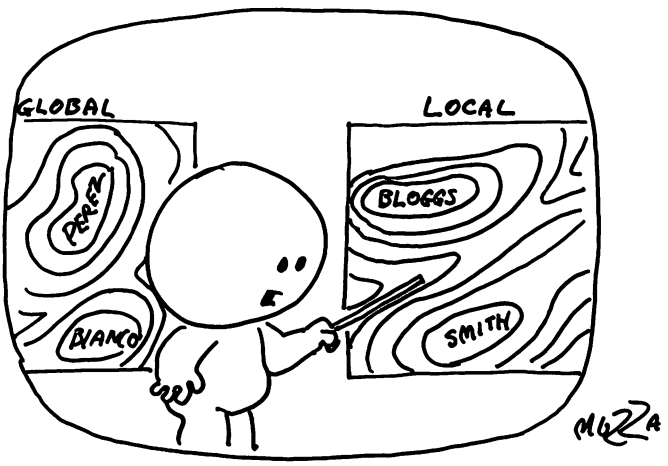
This restriction applies particularly to the use of data items other than those referred to as explicit parameters of the task module.

For example, a module may be required to perform certain calculations which necessitate the storage of temporary results during the evaluation process. It would be wrong for that module to use data items which, although they had no particular connection with the current task, were not being used for any other purpose at that point of the program's execution. In such a case, the calculation module should create temporary storage areas for its own use. The additional amount of memory utilization in such cases is usually trivial and the problems avoided may be considerable.

3.6 LOCAL AND GLOBAL DATA

To facilitate the concepts of module independence and information hiding, it is advantageous for program modules to be able to lay exclusive

claim to certain data areas in a program. This means that they may feel free to use these areas for whatever purpose is required without any concern for effects on the remainder of the program and, conversely, they need have no concern that other program modules are corrupting data which belongs solely to that task. For this reason, as much data as is feasible should be passed as parameters to task modules, operated on in the modules' data areas and returned again as parameters.



Certain data, however, will fall outside this category. Either because of its universal requirement for use by all sections of a program or because of its volume, certain data may have to be regarded as globally accessible by all sections of the procedural code. Records read from files and large arrays, or tables, of data will almost certainly fall into the category of global data.

Some programming languages have syntax facilities for the declaration of local and global data items, for example, FORTRAN, PL/1, Pascal. Others, however, have no such formal facility and rely on the discipline of the programmer to enforce such a standard, for example, BASIC, COBOL.

Whether done by syntax or self-discipline, however, a programmer should consciously seek to divide all data items in a program into two categories:

(a) global items which may be accessed and altered by any portion of the program code, and

(b) local items which are essentially the property of one task module and which may only be used either as an input and/or output parameter for that module or by the module itself as a repository of temporary calculations.

Unless otherwise stated explicitly in the specification for any module, it is presumed that the data items provided as input parameters do not have their contents corrupted during the course of the module's operation.

3.7 PARAMETER CHECKING

A perennial problem concerning the transfer of parameter data to and from processing modules is that of validation, that is, checking for correctness/reasonableness. In some instances, the nature of the module itself will detect erroneous input; for example, in a routine to check the validity of a calendar date or the existence of a code in a table, it will be the nature of the task to detect a date or code which is invalid. Even such procedures, however, may require an initial check to ensure that the data presented for validation is of a correct type; for example, is the calendar date a numeric value? What would be the effect on the algorithm if it were asked to validate a date of ABCD1984? If the assumption by the module that the date was entirely numeric would cause the program to abort because of incorrect data type, it is essential that this check be made before proceeding to examine the date any further.

If a module were to compute the square root of a given number, it would seem to be essential that the parameter was checked to ensure that it was not negative.

Apart from such cases as those mentioned above, a program module must often take on trust the parameter data passed to it. To do otherwise would result in excessive rechecking of data and consequent loss of efficiency. For example, a module required to compute the days' duration between two calendar dates may have to rely on the fact that the program has previously subjected both of those dates to a validation procedure and would only have executed the duration calculation provided that both of the dates had been found to be correct.

Despite the 'defensive programming' warning that all parts of the program should be mutually suspicious of one another and should, as far as is practicable, check their own input for validity, common sense must prevail, and the following guidelines are offered:

(a) A module must always validate its input against any condition which would cause program termination. Possible candidates here are non-numeric values in numeric data items, negative arguments to square root functions, and so on.

(b) Subject to (a), a module need not validate any input parameter which has been validated previously within the program (or by an earlier program) provided that the reliance on that earlier validation is explicitly stated as part of the specification for the module concerned.

Hence, the mini-specification for a module may appear as:

Module Name:	Calculate Duration
Function:	To calculate the days' duration, in a positive or negative direction, between two given calendar dates.
Input Parameters:	START_DATE, END_DATE The boundaries of the duration to be calculated. Both dates are presumed to be of the format DDMMYYYY, to be not before 1/1/0000 and to have been previously checked for validity. Failure of these criteria will result in unpredictable results.
Output Parameters:	DURATION Contains a signed integer representing the days' duration between the two given dates.

3.8 CONCLUSION

A program is best constructed as a collection of task-oriented modules which operate as independently of one another as possible. Each module must be defined in terms of what it does and how it communicates with any other section of the program which wants to make use of its services. The method by which it performs its function is best unknown to the remainder of the program, and it should operate as far as practicable on data which is local to itself.

Programs which are developed in this fashion are easy to check and easy to maintain.

Modules thus constructed may be placed in module libraries and may be incorporated in many different programs, this ability being the ultimate test of a module's independence from the rest of its surroundings.

4 PROGRAMMING STANDARDS

CHAPTER 4

4.1 PROGRAMMING STANDARDS

The Need for Standards

Programming standards are needed to provide a degree of predictability in programs of a common type, written in a common language or written for a common computer installation.

Standards contribute to:

(a) ease of program design,
(b) ease of program maintenance, and
(c) efficiency of computer resource utilization.

Each of these considerations will be examined in some detail.

4.2 CRITERIA FOR PROGRAM ASSESSMENT

When embarking on the writing of a program, there are many criteria facing the programmer, for example:

- Correctness of the solution.
- Cleverness of the algorithm.
- Minimum time taken to write the code.
- Maximum speed of program execution.
- Minimum use of memory space.
- Ease of comprehensibility and subsequent maintenance.

Many of these criteria are at odds with one another and the pursuit of one will necessarily be at the cost of others. Before giving attention to each of these individual criteria, the programmer must realize that, except in extremely rare situations, it is impossible to satisfy them all in any one program. Therefore, a programmer must establish a clear set of priorities which are always followed unless there is a reason for deviation in some specific set of circumstances.

Correctness of Solution

It has been so often said that 'any program which works correctly is a good one and any program which doesn't is a bad one' that most people have come to believe this piece of nonsense.

The true test of most programs is their ability to serve their user over an extended period of time, perhaps three to ten years. During this time they will typically be amended many times to reflect changes in the user's requirements. A program which works correctly as initially written but, because of the programmer's poor technique, is either unable to be modified or is enormously difficult to modify is *not* a good program.

Conversely, a program which contains a 'bug' (or error) which produces incorrect results but which is written in such a clear style as to enable the source of the error to be quickly identified and corrected and which also enables the program to be successfully modified over a period of time is *not* a bad program.

Adopting correctness of solution as the pre-eminent goal in programming is a mistake. Concentration on clarity of structure and style will produce a better program and will achieve correctness of solution as a by-product.

Cleverness of the Algorithm

There is a hackneyed principle in programming called KISS — Keep It Simple, Stupid. Unless there is some overriding need to the contrary, the best algorithm to use in the solution of any problem is one which will be apparent to another reader of the program, and indeed to its author when he returns to the program after not seeing it for some time.

If a need for, say, maximizing program speed or minimizing memory usage requires the abandonment of a simple solution in favour of one which is more efficient but somewhat difficult to follow, the programmer must include enough documentation in the program (by way of comments, etc.) to enable the solution to be understood at a later time.

Minimum Coding Time

Most programmers are familiar with the so-called 'quick and dirty' program, one which is written in a tearing hurry to serve a once-only need and which will then be discarded, thus removing the necessity for clarity of structure and style.

Those same programmers will know that the once-only program is then not discarded after use 'just in case it is needed again sometime'. It becomes part of an organization's software library and is almost certain to be modified later to meet a different set of circumstances. The maintenance programmer (who may be the original author) will rue the decision which resulted in such a monstrosity.

There really is no such thing as a one-shot-only program. Resist the temptation of quick and dirty coding. Write all programs as though they were to be maintained by programmers other than their author over a long period. The additional coding time will usually be minimal and is more than likely to be recouped in reduced testing time to ensure that the program operates correctly.

Maximum Speed of Execution

The vast majority of programs are not so time-critical in their execution that they warrant sacrificing simplicity of algorithm and clarity of structure to achieve additional speed. Normal efficiency standards, discussed below, should weed out gross wastage of execution time.

If the KISS principle is abandoned in favour of a procedure which, although obscure, saves time during execution, there should be enough documentation prepared by the programmer to enable subsequent program maintenance.

Minimum Use of Memory Space

Precisely the same argument for the maximum speed of execution applies in this case. Normal efficiency standards should prevent gross wastage of memory. If special techniques are used in exceptional circumstances, the programmer must ensure that his procedures are well documented.

Comprehensibility and Ease of Maintenance

In the absence of any conflicting requirements, this factor should be regarded as the overriding criterion in program design. Given the length of time over which a program is used, the number of modifications typically needed to enable it to meet changing requirements and the number of different programmers likely to be involved in effecting those changes (or 'patches'), it is essential that the author of the program follows clearly

defined standards aimed at producing clarity of structure and style.

This should achieve the two goals of:

(a) enabling the program initially to be written, tested and placed in production in the shortest possible time, and

(b) enabling subsequent 'generations' of programmers to read the program code, understand it readily and amend it without difficulty.

4.3 STANDARDS FOR PROGRAM DESIGN

(a) Use standard algorithms.

Without wishing to cramp the initiative of any programmer, it will be far better if individuals or groups in an organization adopt standard approaches to common situations.

There are standard algorithms for table searching, file updating, sorting, and so on, which are available by reference to published works. Use of these common techniques will free maintenance programmers from having to be familiar with many other methods of tackling the same procedure.

(b) Use standard structure patterns.

Adhere to the precepts of module design discussed in Chapter 3, and consistently construct programs in the same pattern. Use a mainline, or driving, module to control the overall execution of a program. Use invoked subroutines to perform the individual tasks constituting the program's operations. Where programming languages have a variety of possible module entities, use only one as a standard; for example, in COBOL use Sections not Paragraphs as the module entity.

Never invoke a program segment as a subroutine module in some instances and use it as a section of code to be sequentially 'fallen through' in others.

The term 'subroutine' here is used to describe the concept of a module which is invoked from a distant point in the program (typically by an instruction such as CALL, GOSUB, PERFORM, etc.), executes and then returns program control to the instruction following its invocation. The pseudocode operation **Do** has been used in this book to cover this concept.

(c) Use standard control mechanisms.

When, as is often the case, a program module is to be repeated many possible times depending on the occurrence of a nominated

condition, adopt a consistent procedure to handle the status items which control the repetition of the procedure; a recommended style is, for example:

```
Set condition-x to be false
Perform until condition-x is true
    Statements
    .
    .
    .
```

Unless the controlling condition is explicitly set to its false status immediately before embarking on the loop, there is a danger that an earlier performance of the loop may have left the condition set to its TRUE status and hence any following 'perform until' will ignore the loop as the controlling condition is already true.

(d) Use the subroutine invocation, or procedure call, as the normal method of sequence control, not the branch instruction.

As already emphasized in earlier chapters, indiscriminate use of branch instructions (typically a 'GO TO' instruction) causes poor program structure. The programmer should seek out and use whatever syntax constructs his current language provides to enable program control via:

Perform until	— loops
If/else	— conditional tests
Do	— subroutine invocation

which are the pseudocode operations used to construct the program algorithm before its coding in the target language.

Branch instructions of the GO TO type should only be used in extraordinary situations or where needed to implement one of these constructs owing to the absence of a suitable programming language facility.

4.4 STANDARDS FOR PROGRAM MAINTENANCE

Defining Data Items

(a) Where the syntax of the language permits:
Use meaningful names, for example:

```
GROSS_PAY
STOCK_ON_HAND
TOTAL_MALES_UNDER_18_YEARS
```

Otherwise:
Provide a data dictionary as comments, for example:

```
REM G: GROSS PAY
REM S: STOCK ON HAND
REM T1: TOTAL MALES UNDER 18 YEARS
```

Take pains to ensure that the data names are *really* meaningful. The name SALES_TAX, for example, could mean the amount of the tax, the percentage tax to be levied, a factor to be multiplied by the cost of an item of goods, and so on. Make the name quite explicit; for example:

```
SALES_TAX_VALUE
SALES_TAX_PERCENTAGE
SALES_TAX_FACTOR
etc.
```

Unless the language used restricts data names to only a few characters, do not use abbreviations, that is:

```
Use GROSS_WEEKLY_PAY
not GRWKPAY
Use PAGE_NUMBER
not PGENO
```

(b) Define related data items together in a group and provide a comment if this would be helpful, for example:

```
COMMENT: PAY CALCULATION ITEMS
  HOURS_WORKED
  HOURLY_RATE
  GROSS_PAY
  TAX_AMOUNT
  NET_PAY
```

For clarity, separate each such group from its neighbours by a few blank lines.

(c) Never rely on data items having a specific initial content other than that which your program explicitly assigns to them.

Some implementations of some languages initialize data items to, say, zero or blanks. This should never be relied on. If, for example, a counter or a total must start with an initial value of zero, ensure that a program instruction puts that value in the data item. Never presume that such an item automatically starts at zero.

(d) Preserve the distinction between data items and status indicators.

As already mentioned, some languages have a special class of 'logical' or 'Boolean' variables to be used to represent status items. Those languages which do not have this facility must use ordinary data items to perform the task of status indicators. Use two such items, for example, TRUE (value, say, 1) and FALSE (value, say, 0) to set and test the contents of these indicators and do not use data items to perform both the task of holding data at some times and at others indicating the status of a condition.

(e) Preserve the distinction between local and global data items.

To facilitate module independence, inter-module coupling may be reduced by providing each program module with its own parameter data items to communicate with the other segments of the program. These items are 'local' in the sense that they may only be used for the purposes of their owner modules and for two-way communication with those modules.

Data which is freely available for any portion of the program to access, for example, a record read from a file, a table of reference material, and so on, may be regarded as 'global'. Many languages have no means of formally differentiating between these two types of data and it becomes a matter for the programmer to implement by means of a discipline in the code.

Procedural Code (Instructions)

(a) Write one instruction per line.

(b) Separate program modules or related groups of instructions by a few blank lines to enhance comprehensibility.

(c) Indent related instructions in loops and conditional statements where the syntax of the language permits, for example:

```
FOR N = 1 TO 10
  A = B + C(N)
  P = Q*R(N)
  X = Y/Z(N)
NEXT N

IF A = B
  MOVE X TO Y
  ADD P TO Q
ELSE
  MOVE D TO E
  ADD J TO K.

BEGIN
  A := B;
  P := Q*R;
  X := Y - Z
END
```

(d) Do not use constants (or literal values) in the procedural code.

Consider the following program statements (the programming language is immaterial):

```
COMPUTE GROSS_PAY = HOURS_WORKED*14.75

IF EMPLOYEE_TYPE = 6
  PERFORM SPECIAL_CALCULATION.

IF DATA_OK = FALSE
  DISPLAY "LAST RECORD READ WAS INVALID".
```

In the first example, we may draw the inference that the factor 14.75 is most likely the hourly rate of pay of the employee (i.e. $14.75) and is hence used here in calculating his gross wage. When the hourly rates change, as they are bound to do, the maintenance programmer must spend a considerable amount of

time in locating the places (because there may be several pay calculations to cover several types of employees) where these pay rates have been used and amend each of the values. If he misses one, that class of employees will not receive their pay rise!

In the second example, two problems arise. The first is that it is not at all clear what a 'type 6' employee is. The second problem is that if whatever type of employee this is testing for has his category changed from 6 to some other value, the maintenance programmer is faced with the same search-and-amend task as in the first example.

In the third example, it is possible that the error message may need to be shortened, to say, "INVALID RECORD" at some future time. Again, the search-and-amend necessity.

The solution to each of these problems lies in defining the constant as a data item with a mnemonic name and using that name in the procedural code, for example:

```
HOURLY_RATE                ... VALUE 14.75
CASUAL_EMPLOYEE_CODE       ... VALUE 6
ERROR_MESSAGE              ... VALUE "LAST
                       RECORD READ WAS INVALID"
```

and then

```
COMPUTE GROSS_PAY
    =HOURS_WORKED*HOURLY_RATE
IF EMPLOYEE_TYPE
    =CASUAL_EMPLOYEE_CODE
        PERFORM SPECIAL_CALCULATION
IF DATA_OK=FALSE
   DISPLAY ERROR_MESSAGE
```

If any of the literal values change, irrespective of the number of times which they are used in the procedural code, there is only one amendment needed to the program, that is, the altering of the VALUE assigned to the symbolic name.

(e) Construct the program from a number of small, independent, functionally oriented modules.

This tends to be a reinforcement of the precepts expounded in this and earlier chapters. Each program module should conform to the following criteria:

(i) it should have a meaningful name (e.g. COMPUTE_TAX)

where the syntax of the language permits, otherwise it should have a comment statement to indicate its function;

(ii) unless its purpose is plainly apparent from its instructions, it should contain a few comment lines explaining its purpose within the program;

(iii) where appropriate it should have local data items assigned to it for its own use and for communication with modules which make use of its function;

(iv) it should carry out a single function; and

(v) it should consist of relatively few instructions; say a maximum of 50 lines of code.

4.5 STANDARDS FOR EFFICIENCY

(a) Where the syntax of a language permits several types of data items to be defined (e.g. binary, packed, decimal, character), ensure that the most suitable data format is used when programming mathematical calculations. This will normally imply the use of binary variables. This point is particularly applicable to languages such as COBOL and PL/1 where computations may use character variables at a consequent cost to program efficiency. Languages such as BASIC, FORTRAN and Pascal will permit numeric variables to be held only in binary format.

(b) Avoid repeated operations on subscripted variables; for example, rather than coding:

```
If Item (N)>zero
    Add Item (N) to Total
    Multiply Result by Item (N)
    Subtract Item (N) from Answer
```

it would be far more efficient to write:

```
Temporary Item←Item (N)
If Temporary Item>0
    Add Temporary Item to Total
    Multiply Result by Temporary Item
    Subtract Temporary Item from Answer
```

Evaluation of subscripts is an onerous task for the program at run time. In the second example above, only one evaluation is needed

compared with four in the first example to achieve the same result.

(c) Use efficient testing procedures. Two points are worth noting when using testing procedures:

(i) When testing for a range of possible values, do not repeat unnecessary tests, for example:

Do not write:

```
If Item = value − 1
    statement(s)
If Item = value − 2
    statement(s)
If Item = value − 3
    statement(s)
      .
      .
      .
```

This will cause the item being examined to be subjected to *every* test.

Instead, write:

```
If Item = value − 1
    statement(s)
else If Item = value − 2
    statement(s)
else If Item = value − 3
    statement(s)
      .
      .
      .
```

The use of **else** as a connector between all of these tests will ensure that as soon as one value has been successfully recognized, no further tests will be carried out.

(ii) When testing for a range of possible values, test for them in their order of probability of occurrence, that is, from the most likely down to the least likely. Hence, if a program was required to check the type of each employee before a pay calculation and the distribution of employees was:

Type 1 10%
Type 2 16%
Type 3 54%
Type 4 20%

the most efficient testing order would be:

```
If employee type = 3
   statement(s)
else If employee type = 4
   statement(s)
else If employee type = 2
   statement(s)
else If employee type = 1
   statement(s)
```

(d) No loop should contain statements which do not vary as the loop progresses.

For example, we may need to place the result of a calculation in each item of an array. The coding could take the form:

```
N←1
Perform until N>50
   ITEM (N)←A*B/C+D−E
   N←N+1
```

In this example, the expression A*B/C+D−E is calculated 50 times, presumably each time giving an identical result, in order to place the result in each of the 50 occurrences of the item. A much more efficient solution would be:

```
X←A*B/C+D−E
N←1
Perform until N>50
   ITEM (N)←X
   N←N+1
```

In this example, the arithmetic expression needs to be evaluated only once.

(e) Owing to the practice of 'paging' or 'program segmentation' whereby a program is read into memory to be executed in a series of 'pages' or 'segments' or 'overlays', it is often advantageous to place program modules which are related in their function physically adjacent to one another in the coded program. If module A calls for the performance of modules B, C and D as part of its execution, it will be advantageous if A, B, C and D are all in the program segment currently in memory. This is only achievable if those modules follow one another in the program code.

> For this same reason, that is to minimize the number of disk file accesses needed to read segments of a program into memory as it is executed, it is preferable to group often-used modules together at the top of the program code and to relegate once-only or exception routines to the bottom of the code.

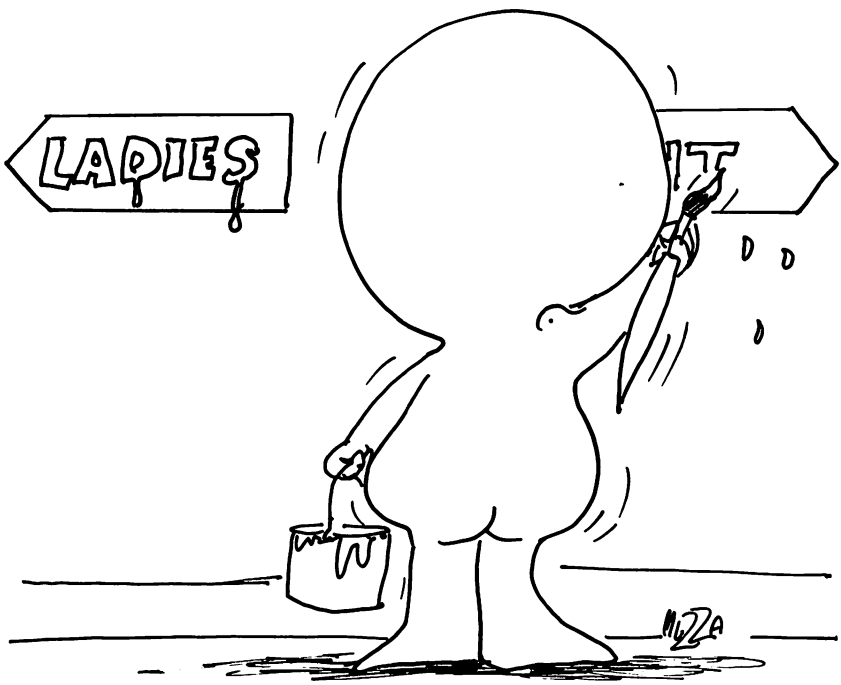
Further considerations related to program efficiency are usually dependent upon the specific characteristics of one particular language or model of computer. The programmer should be aware, however, of the impact on efficiency which the use or avoidance of certain procedures causes. Determination of these factors will normally require diligent perusal of the language or computer manuals.

4.6 CONCLUSION

Standards are often seen by programmers as a limitation on their ingenuity or inventiveness. Following a set of standards, however, is the only means of producing programs of consistent design structure and clarity.

A programmer should aim to develop a set of personal standards and an installation should adopt a realistic set of standards which are sufficiently concise to be understood and remembered by the programmers without descending to trivial levels which, by annoying programmers, result in the rejection not only of them but also of the more important standards.

5 PROGRAM DOCUMENTATION



CHAPTER 5

5.1 PROGRAM DOCUMENTATION

The provision of documentation is one of the most neglected areas of the programming discipline and is also one of the greatest points of separation between the amateur and professional programmer. The amateur programmer usually writes programs which only he needs to understand, operate and maintain. The professional programmer is writing programs for others to use and maintain over a long time. Because the provision of the necessary documentation to support a program is not seen as a challenge or intellectual exercise (and because it is almost invariably left until after the program has been developed and is then seen as delaying the new program), it is seldom done or done properly. It should be one of the fundamental principles of programmer management that no program is allowed to be used without having been provided with sufficient documentation.

5.2 WHO NEEDS DOCUMENTATION?

There are three groups of people who need to refer to program documentation and their requirements are all somewhat different.

Programmers

Once having been written, programs are used in their operating environment for a period which may span ten or more years. During this time they will be subject to many alterations or enhancements to reflect the changing requirements of their users. This task is referred to as 'program maintenance' and is carried out by 'maintenance programmers'. It is not uncommon to find that most organizations spend much more than 50 per cent of their programming effort in this area. This means that the average programmer is spending well over half of his time working with programs which were most likely written by other programmers who may no longer be around to be consulted on the details of their code. Unless the programs were initially well written and adequately documented, maintenance programming is an extremely unattractive pastime!

Operators

The amateur programmer runs his own programs; the professional does not. In a situation in which programs are run by an organization's computer centre on data which is collected, batched and run in periodic cycles, there will be a group of computer operators whose task is to manage the physical environment of the computer equipment. It is essential that these operators are aware of the task carried out by any program and any likely interaction between that program and the operator at run-time (i.e. during the program's execution).

Users

The user is the person or section of an organization for whom a program is run and who relies on its output for the continuance of their functions. Like the programmer, the user may never see his programs actually run, this task being managed by the operators. The user must understand what input he is required to provide for the program, the nature of the processing carried out by the program and the output which he will receive. He needs to be aware of any restrictions on the nature and content of the input and any errors which may be reported on the output. He does not need to know the precise means by which the data is processed except to understand any limitations which may affect subsequent amendments to the program's task.

In an interactive operating environment, the roles of user and operator may be combined. A clerk accepting orders over a telephone and entering their details immediately into a computer system via a VDU needs the documentation required both to initiate and manage the program's operation and to understand the input-processing-output cycle.

5.3 DOCUMENTATION FOR PROGRAMMERS

The documentation required to enable a programmer to maintain a program over its life span may be divided into two categories.

Internal Documentation

This covers the aspects of programs which are embodied in the syntax of

the programming language. The important points, already covered in the previous chapter, are:

- Meaningful names used to describe data items and procedures.
- Comments relating to the function of the program as a whole and of the modules comprising the program.
- Clarity of style and format: one instruction per line, indentation of related groups of instructions, blank lines separating modules.
- Use of symbolic names instead of constants, or literals, in the procedural code.

External Documentation

This category covers the supporting documentation which should be maintained in a manual or folder accompanying any program. It is essential that as changes are made to a program, its external documentation is updated at the same time. Out-of-date documentation can be misleading to a maintenance programmer and result in lots of time being wasted. There has been emphasis recently on the inclusion of as much as possible of any program's documentation in the source program itself.

External documentation should include:

- A current listing of the source program, that is, the program as written

by the programmer, including any memory maps, cross-references, and so on that are able to be obtained from the compilation process.

- The program specification, that is, the document defining the purpose and mode of operation of the program.
- A structure diagram depicting the hierarchical organization of the modules currently comprising the program.
- An explanation of any formulae or complex calculations in the program.
- A specification of the data being processed, that is, data accepted from or displayed on a screen, items in reports, external files processed, including the format of record structures and fields within those records, all data being described in terms of its size, format, type and structure.
- Where applicable, the format of screens used to interact with users and of printed reports.
- The test data used to validate the operation of the program.
- Any special directions of importance to programmers subsequently amending or enhancing the program, for example, restrictions on the sizes of tables, and so on.

5.4 DOCUMENTATION FOR OPERATORS

Persons operating programs do not need to know precisely how the programs work. They need to know the points at which their actions and the program interrelate, that is, their interface with the program.

Such documentation is usually contained in an operating instructions document and should cover:

- The command(s) necessary to load the program into memory from secondary storage (e.g. disk or tape) and to start its operation.
- The names of all external files accessed by the program.
- The format of all messages which the program may display to the operator during its execution, describing the text of the message, the situation causing the message to be displayed and any action to be taken by the operator as a result of the message.
- Any options in the program's operation which require operator action to trigger at run-time (e.g. a special end-of-month run, etc.).

- A brief description of the program's function so that the operator can obtain a feeling for whether or not the program is behaving correctly.
- Any technical details relating to the equipment being used (e.g. a minimum amount of memory space required for execution, a minimum number of peripherals or file storage capacity).

5.5 DOCUMENTATION FOR USERS

As in the case of the operator, a user is concerned more with how he interacts with the program and what the program does for him than with the technicalities of how the program goes about its tasks.

All programs or collections of programs comprising a composite system should have a User's Manual. This document follows much the same path as that of the Operating Instructions except that the user will not normally be as concerned with equipment technicalities as will the operator.

The User's Manual should cover:

- A detailed description of the function performed by the program.
- The means by which the user supplies data to the program to be processed, covering the format and content of the data together with any restrictions on values included, and so on.
- A detailed description, preferably with examples, of any output produced by the program for the user.
- Details of any error messages or exception reports which the program may produce, explaining precisely their impact on the user and any subsequent action required on his part.
- Details of any options able to be exercised by the user, including the implications of each option and the means of selecting each option.
- Any restrictions on subsequent amendments or enhancements to the program, to enable the user to obtain a realistic appraisal of the prospective usage of the program.

As with any technical report, the User's Manual should be clearly presented, simply explained and indexed to facilitate its use by its target audience which will usually consist of non-technical staff who may have only a rudimentary understanding of the nature of a computer system.

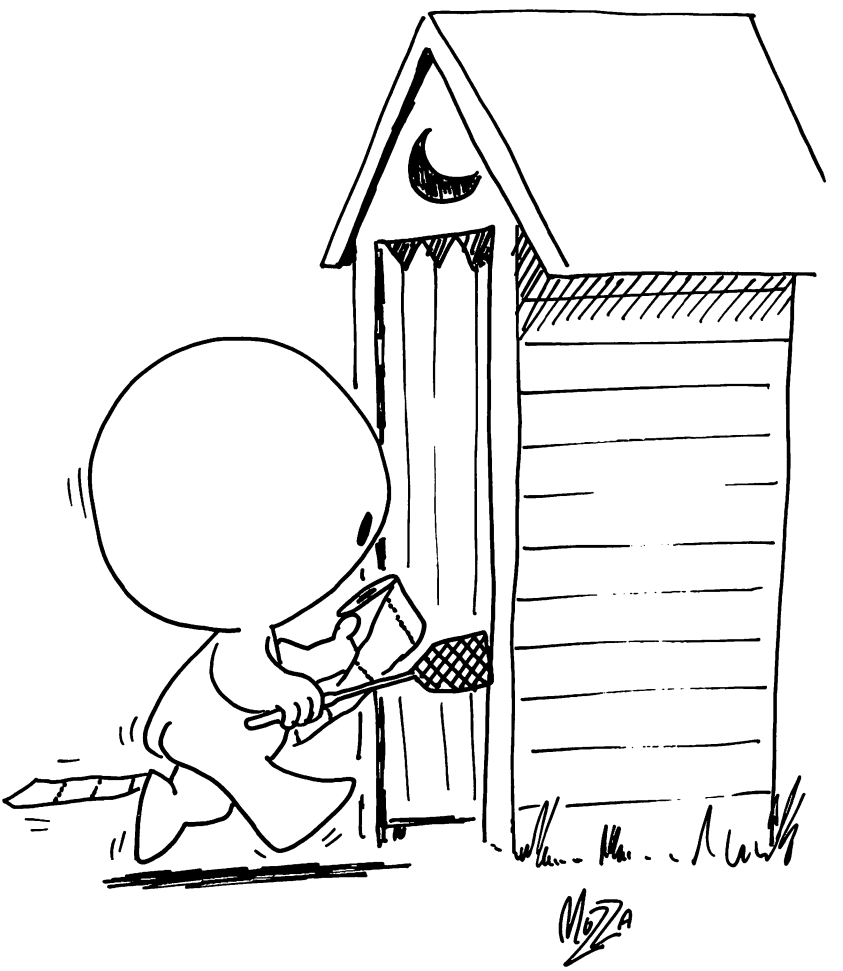
5.6 DOCUMENTATION FOR INTERACTIVE SYSTEMS

As stated earlier, a large proportion of contemporary data processing systems require on-line interaction between program and user via terminals. The documentation required in such systems is a combination of that for operators and users. Care must be taken that the technical aspects of the system are kept separate from the operational aspects to allow the user to follow his routine procedure without becoming embedded in a morass of detail.

5.7 CONCLUSION

The lack of sufficient documentation at all levels plagues most computer installations. The production of documentation must be seen as a task as necessary and important as the writing of program code. It is a function of professional programming management to ensure that no system is allowed to become operational without having a sufficient volume of appropriate documentation, and to ensure that the documentation is continually updated as the system itself changes to meet the altered requirements of its users.

6 PROGRAM TESTING/DEBUGGING



CHAPTER 6

6.1 PROGRAM TESTING/DEBUGGING

Although many authors attempt to draw a distinction between *testing* and *debugging* a program, I believe that in practical terms they amount to the same thing. Consequently, I will use the term *testing* to cover this procedure.

> *Testing is the process of executing a program with the deliberate intent of finding errors.*

This definition is important because it specifies a state of mind on the part of the person performing the test. It is akin to the engineering practice of testing a mechanism or structure not only to verify that it can perform its designed task under ideal conditions but also to determine the point (if any) at which it fails.

6.2 TESTING IN ITS CORRECT PERSPECTIVE

Diligent and exhaustive testing is no substitute for careful design. The time taken to test a program to satisfaction will be in proportion (with an inverse ratio) to the time taken in designing the program before its coding. Days spent in design will save weeks in testing.

The elegance of design of any program will directly affect both the ease of testing the program and the incidence of errors detected. I will not refer to errors as 'bugs' because gracing them with a euphemism makes it easier for us as programmers to ignore the fact that they are *errors*. They are faults in the execution of a program which have been induced by the programmer through poor design or incorrect logic.

Most errors are the result of incorrect control functions in programs rather than incorrect computation or data manipulation. The latter types of error are usually the easiest to detect and correct; the former often go undetected and remain in programs after they have been approved to run in a production (or 'live') environment. Control functions are those which regulate the sequence in which tasks are performed, the selection of appropriate processing modules for a given situation, the number of times a loop is executed, and so on.

6.3 WHEN TO TEST?

Program testing is a continuous procedure. The first test of a program should be in advance of any coding. Once the algorithm has been expressed in pseudocode, it should be subjected to a desk check. This is a procedure which should be applied to each module of the algorithm and involves selecting sample values of test data for the inputs to the module and processing them precisely as directed by the algorithm to determine whether or not they produce the desired output. This presupposes that the correct result was initially calculated independently of the written algorithm.

Such desk checking is ideally done by a person other than the author of the algorithm. This could be another programmer, or if the pseudocode has been well written, the user of the program. Presumably the user is in the best position to know whether the results produced by an algorithm are what he would have required.

Having convinced himself that the algorithm is correct, the programmer proceeds to code it in a programming language. The next stage of testing occurs when the coding has been freed from syntax errors and is in a format which allows it to be executed. It may then be tested module by module or as a complete entity.

6.4 DESK CHECKING

The procedure of desk checking is illustrated by this example:

(a) The problem definition:
A factory pays its employees each week on the basis of a data record prepared for each employee. This record contains:

Employee Name, Hours Worked, Hourly Pay Rate

It is the factory policy to pay time-and-a-half for hours worked beyond 40 in any week. Taxation is to be calculated at 20 per cent of any wage greater than $200 for a week and nil otherwise. The program is to print each employee's name, gross pay, tax and net pay.

(b) The pseudocode written by the programmer:

```
Calculate Wages
    Read employee record
    Perform until end-of-file
        Place employee's details in NAME, HOURS, RATE
        If HOURS<40
            PAYABLE_HOURS←HOURS
        else
            PAYABLE_HOURS←40+(HOURS−40)*3/2
        GROSS_PAY←HOURS*RATE
        If GROSS_PAY is not<200
            TAX←GROSS_PAY*20/100
        NET_PAY←GROSS_PAY−TAX
        Print NAME, GROSS_PAY, TAX, NET_PAY
        Read employee record
    Stop
```

(c) Test data:
This table shows the data to be used to test the program and the expected results in each case:

Test Data			*Expected Results*		
Name	*Hours*	*Rate*	*Gross*	*Tax*	*Net*
ADAMS	35	5.00	175.00	0.00	175.00
BAKER	40	5.00	200.00	0.00	200.00
CAIRNS	44	5.00	230.00	46.00	184.00
DONALD	20	6.00	120.00	0.00	120.00

(d) The check table:
The procedure used to check the operation of the program is to allocate an area on a working pad for each data name used by the

program and to follow the program instructions, doing precisely what they indicate, *not* what you 'think' should be done in each case. It is important to start each data area with an initial value which is unknown. Unless a program initializes an area by a specific operation, its contents are likely to be unpredictable. The following table illustrates the testing of the program above with the test data shown:

Test	*Name*	*Hours*	*Rate*	*Payable Hours*	*Gross*	*Tax*	*Net Pay*
Start	?	?	?	?	?	?	?
1	ADAMS	35	5.00	35	175.00	?	(175.00–?)
2	BAKER	40	5.00	40	200.00	40.00	160.00
3	CAIRNS	44	5.00	46	220.00	44.00	176.00
4	DONALD	20	6.00	20	120.00	44.00	76.00

The program has, so far, not produced a single correct result! In working through its operation, the programmer should have discovered the following faults:

(i) Because the first employee had no tax calculated, the initial contents of TAX is unpredictable and hence the first value of NET_PAY is unpredictable in such a case.

(ii) In the case of overtime (hours exceeding 40), the PAYABLE_HOURS are being correctly calculated, although unnecessarily in the case of exactly 40 hours, but the program is using HOURS not PAYABLE_HOURS in its calculation of GROSS_PAY.

(iii) If the GROSS_PAY is exactly $200.00 it should not be taxed; however, the program only exempts incomes less than $200.00.

(iv) If an income is less than $200.00 there is no tax calculation performed. This will leave undisturbed the existing contents of TAX which will then be imposed on the employee with the non-taxable income.

Having discovered these faults in the algorithm logic, the programmer should correct the erroneous instructions and re-test the program with the same test data. This procedure would be repeated until the program executes successfully on all test data.

It is important that the programmer does not proceed to the coding phase until the pseudocode algorithm has been verified.

Exercise

Rewrite the algorithm above and prove that it works by application of the test-data using a desk-check table.

DATA DRIVEN vs LOGIC DRIVEN TESTING

Data driven testing refers to the principle of regarding the program, or any segment of it, as a 'black box' and ensuring that it gives a correct output for all input data. This approach regards the algorithm as relatively unimportant provided that it executes correctly for the range of data that it is required to process.

The advantages of data driven testing are:

- It requires empirical testing of an algorithm with real data rather than the forming of an opinion about the correctness of an algorithm by examining its construction.
- It can be done by a user of the program as an alternative to a programmer.
- It provides documentary evidence, by way of printed or displayed output, that an algorithm works correctly.

Its disadvantages are:

- The programmer may omit to supply test data which would have uncovered a flaw in the algorithm.
- 'Correct' results may sometimes be the consequence of compensating errors within an algorithm, errors which may cease to compensate one another as a result of a subsequent change in the operating environment.

Logic driven testing relies on an examination of the algorithm of the program, or any segment of it, with the aim of detecting any flaw in its procedure.

The advantages of logic driven testing are:

- A thorough walkthrough the logic using desk-check procedures should uncover any weakness in the processing procedure; these weaknesses are often detected as a by-product of processing data which is itself handled correctly but indicates to the programmer doing the testing that

a somewhat different choice of data would not have been handled correctly.

- If the testing is done by a programmer other than the author, suggestions may be made concerning the improvement of the structure, style or efficiency of the algorithm.

Its disadvantages are:

- Test data may be chosen to fit the algorithm and either deliberately or subconsciously avoid causing an error.
- It requires a programmer to effect the test; a user cannot be relied upon to have sufficient knowledge of the programming language.
- A programmer, whether he is the original author or another, may bring to the testing procedure the same preconceptions about the operation of the coded instructions as were brought to the coding operation; for example, the programmer who wrote the instruction:

```
IF SEX_CODE IS NOT = "M"
OR SEX_CODE IS NOT = "F"
   PERFORM ERROR_ROUTINE
```

will be just as convinced of the correctness of this test during the checking of the program as he was during its coding and will be at a loss to explain later on why the program rejects *all* SEX_CODEs (including M and F!).

6.6 SELECTION OF TEST DATA

The design of test data for any program is an extremely important task. Test data should be designed from the program specification, not from the program algorithm or the code itself. Although a programmer may construct test data to satisfy himself that the program executes correctly, the primary responsibility for the formulation of test data and the expected results lies with the user of the program, that is, the person or organization requesting the program to be written.

Remember, the purpose of testing a program is to verify that:

- The program does what it is supposed to do.
- The program does *not* do anything which it is *not* supposed to do.

Consider the problem of testing a program to read three positive integers and print them in ascending sequence. Using A, B and C to

represent any three different integers where A is the smallest and C the largest, the test data should include the following patterns:

A, B, C
A, C, B
B, A, C
B, C, A
C, A, B
C, B, A
A, A, B
A, B, A
B, A, A
A, A, A

In addition to these patterns, the test cases should include:

Cases where one, two and all numbers are zero;
Cases where some or all of the data is invalid, for example, negative values, non-integer numbers, non-numeric items (e.g. 4B3);
Cases where the number of data items is too few or too many.

Approximately twenty test cases would have to be submitted to even a trivial program to ensure that:

(a) it handles correct data correctly;
(b) it does not treat correct data incorrectly;
(c) it does not treat incorrect data as though it were correct.

It will also be apparent that the testing of programs consisting of hundreds or thousands of instructions to perform a complex function is an extremely creative and intellectually challenging task.

Test data should exercise every path of logic within a program and ideally it should exercise every combination of paths. In a program containing nine decision points, there would be 2^9 possible logic paths taking 512 separate combinations of test data. If the program were designed as three modules each containing three decisions, however, the testing could be reduced to $2^3 + 2^3 + 2^3 = 24$ combinations of test data to test each of the paths in each of the modules, plus a few additional cases to test the communication between the modules.

This is a powerful argument for designing programs as a collection of modules rather than as single monolithic structures.

It is well to remember that it is variety rather than volume which is the requirement of test data. Unless a program is specifically being tested for

its ability to handle a certain number of transactions in a given time or its ability to accumulate totals from a series of identical transaction types, there is no merit in providing multiple occurrences of identical data items. In the above example, if the program could handle the pattern A, B, C for one set of data, it could handle that pattern for any number of such sets and further A, B, C test patterns would prove nothing at all. In selecting test data, one must provide a sufficient variety of test data to exercise all paths of a program's logic and all combinations of occurrences which the program must be able to handle.

In a production environment, a set of test data should be created, preserved and maintained for each program. Any time a modification is made to a program, however minor the patch, the program must be re-tested on the entire test data set to ensure that the correct results are still produced.

6.7 TESTING PROCEDURE

The correct execution of a program implies:

- The correct execution of each component module.
- The correct communication of data between modules.
- The correct sequence of execution of the modules.

The testing of a program is then reduced to verifying these three conditions.

This verification may be done by a variety of methods all of which are variations on a theme often referred to as tracing. Tracing must be applied to the data processed by a program and the control, or execution, path of the program. It will involve communicating to the programmer the point at which the program is currently executing and the values contained in selected variables at that time. The crucial points at which to insert traces in a program are usually the entry and exit to each module.

The information required by the programmer may be printed or displayed on a terminal and may be produced by an instruction (a PRINT, TYPE, DISPLAY, etc.) inserted by the programmer for that purpose or by means of a software tool such as an interactive debugging aid which allows the programmer to specify one or more statements at which he wishes the program to halt for interrogation of the contents of data areas.

In a true-to-life example of a payroll calculation such as that quoted in the earlier example in this chapter, the control structure may be similar to:

Calculate Wages
    Do Read Employee
    Perform until end-of-file
        Do Compute Payable Hours
        Do Compute Gross Pay
        Do Compute Tax
        Do Compute Net Pay
        Do Print Pay Details
        Do Read Employee
    Stop

It should be possible to place in each of these modules tracing procedures which indicate the flow of program control and the intermediate results progressively calculated by the program instructions. Maintaining the errors which existed in the given example, the algorithms for each module (after inserting typical tracing statements) would be:

Read Employee
    Print "READ EMPLOYEE"
    Read employee record
    Place employee's details in NAME, HOURS, RATE
    Print "NAME = ", NAME
    Print "HOURS = ", HOURS
    Print "RATE = ", RATE

Compute Payable Hours
    Print "COMPUTE PAYABLE HOURS"
    If Hours <40
        PAYABLE_HOURS←HOURS
    else
        PAYABLE_HOURS←40+(HOURS −40)*3/2
    Print "PAYABLE HOURS = ", PAYABLE_HOURS

Compute Gross Pay
    Print "COMPUTE GROSS"
    GROSS_PAY←HOURS*RATE
    Print "GROSS PAY = ", GROSS_PAY

Compute Tax
    Print "COMPUTE TAX"
    If GROSS_PAY is not <200
        TAX←GROSS_PAY*20/100
    Print "TAX = ", TAX

Compute Net Pay

```
Print "COMPUTE NET"
NET_PAY←GROSS_PAY − TAX
Print "NET_PAY = " NET_PAY
```

Print Pay Details

```
Print NAME, GROSS_PAY, TAX, NET_PAY
```

(Given that the purpose of this module is to print the contents of all associated variables, it is hardly necessary to insert additional tracing instructions.)

The tracing output from the processing of, say, the third and fourth test records as shown below should, when compared to the expected results already calculated, point out the existence of several errors in the program:

READ EMPLOYEE	(3rd Record)
NAME = CAIRNS	
HOURS = 44	
RATE = 5.00	
COMPUTE PAYABLE HOURS	
PAYABLE HOURS = 46	(correct)
COMPUTE GROSS	
GROSS PAY = 220	(46 hours @ $5.00 = $230)
COMPUTE TAX	
TAX = 44.00	(consistent with a gross pay of $220)
COMPUTE NET	
NET PAY = 176.00	(consistent with gross pay and tax)
READ EMPLOYEE	(4th Record)
NAME = DONALD	
HOURS = 20	
RATE = 6.00	
COMPUTE PAYABLE HOURS	
PAYABLE HOURS = 20	(correct)
COMPUTE GROSS	
GROSS PAY = 120.00	(correct)
COMPUTE TAX	
TAX = 44.00	(tax should be zero)
COMPUTE NET	
NET PAY = 76.00	(consistent with gross pay and tax)

From the trace output the programmer can ascertain that some modules work correctly each time (e.g. COMPUTE PAYABLE HOURS), some work incorrectly sometimes (e.g. COMPUTE GROSS PAY and COMPUTE TAX), and others give incorrect results but results which are consistent with the data which they have had to operate on (e.g. COMPUTE NET PAY and, presumably, the printing of the pay details).

Without this display of intermediate results, the programmer would merely see that the final pay results produced were incorrect for the data input but would not know where the incorrect processing was occurring.

6.8 PROGRESSIVE TESTING (TOP DOWN vs BOTTOM UP)

For a large program, it may not be necessary for the programmer to wait until the entire program has been coded to begin testing. Modules which have not been coded may be replaced by 'stubs' which enable the call to the module to be executed but do not perform all or any of the module's functions.

In the payroll program above, for example, it is likely that the coding of the taxation calculation would be delayed until all other sections of the program are operating correctly. (The actual method of calculating tax is obviously not as trivial as that in the example!) In such a case, the Compute Tax module could be coded to always return to a value of, say, $10 and the rest of the program could be tested independently of that routine.

The procedure of developing the higher level control modules and testing their operation in advance of the lower level 'nuts-and-bolts' routines is referred to as top-down development. Such a procedure allows the programmer to insert progressively more coding, testing each new segment as it is introduced and enabling any errors discovered to be localized and corrected with minimum disruption to the remainder of the program which had previously operated correctly.

In practice, top-down development is often mixed with a certain amount of bottom-up implementation. In some systems, the operation of a low-level routine within certain restraints, for example, a time limit or a memory usage limit, is critical to the functioning of the program as a whole. In such a case the low-level module(s) may be coded in advance of the main program structure and independently tested to ensure that they operate within their design criteria. This requires the critical modules to be surrounded by a 'test harness' of sufficient code to provide them with typical operating input and to evaluate their output.

The taxation calculation routine in our example could be developed in such a manner by executing it in conjunction with a simple-minded module which provided it with an artificial gross pay that started at a certain value and was incremented after each calculation until a prescribed limit was reached.

6.9 TESTING STRATEGY

There are several guidelines worth following when testing programs:

(a) For each set of test data used, ensure that the expected results are pre-calculated.

(b) Do not swamp a program with such a deluge of test data that the results will be too voluminous to check in detail. Often small errors can go undetected in a mass of output.

(c) Thoroughly inspect the results from each test run. Do not assume that something which was correct in the previous run will automatically remain correct. Often the fixing of one error will upset another output which was previously valid.

(d) Test data must cover the invalid and unexpected as well as the valid and expected. Remember that testing is a procedure of trying to find errors in a program.

(e) Ensure that test data is designed by somebody other than the author of the program which it is testing.

(f) In the initial stages of program testing, test one feature at a time. Attempting to test everything in a single run will not initially help to determine which parts of a program work correctly and which do not.

(g) Watch for side-effects errors, that is, ensure that a program segment not only performs its own function correctly but that it does not interfere with another module by, for example, corrupting data which will not be used until a later stage of the program.

6.10 PROGRAM WALKTHROUGHS

Much attention has been given in recent years to the pros and cons of having one or several other programmers examine in detail the code produced by any programmer. The advantages of such 'walkthroughs' are:

- The reviewing panel has no preconceptions about what the code should do and hence may detect errors which fell in a 'blind spot' of the author of the program.
- The reviewing panel may be able to make suggestions to improve the structure, style or efficiency of the code.
- The reviewing panelists may learn something from the code reviewed.

On the other hand, criticism must be constructive. Despite the urging towards egoless programming, most programmers still have difficulty in divorcing themselves from their product and in accepting that criticism of their code is not directed against them.

6.11 COMMON SOURCES OF ERROR IN PROGRAMS

Mistakes made by programmers tend to be of the same nature independent of the programming language used. The following list is a selection of points to check when debugging programs in any language:

(a) Make sure that all constants and variables are explicitly assigned initial values and not assumed to have been initialized by the compiler.

(b) In languages allowing for implicit definition of variable names (e.g. BASIC and FORTRAN) check for misspelling of data names, e.g. MISTAKE←MISTEAK + 1

(c) Check for variables shared between program modules — one module leaving a temporary value in a data area which is then corrupted by another module before being retrieved by the original routine.

(d) Check that status variables (flags/switches) are cleared after their purpose has been served.

(e) Make sure that subscripts do not direct the program beyond the bounds of an array. Many compilers do not generate object code which checks the validity of subscripts at run time. Some languages allow array indexes to have meaningful zero values; others have a minimum index value of 1.

(f) Ensure that the control of loops is correctly implemented and that they correctly handle first, intermediate and last iterations. Check also for a zero-iteration loop, that is, a condition requiring no executions of the loop.

(g) Check that tests are handled correctly and that all test cases are handled explicitly; for example, a test for a variable to contain a value from 1 to 3 may test explicitly for 1 and 2 and erroneously assume that anything else must be 3.

(h) Check for compound negative conditions to be expressed correctly, for example, a test to ensure that SEX contains only either "M" or "F" and expressed as:

```
IF SEX NOT = "M" OR NOT = "F"
THEN execute error routine.
```

will reject every value of SEX including "M" and "F". The test should have been expressed as:

```
IF SEX NOT = "M" AND NOT = "F"
THEN execute error routine.
```

6.12 SYSTEM TESTING

In a working environment, a data processing application may involve anything between five to a hundred programs all of which must interact to provide a service to the user. Testing in such an environment involves not only testing each individual program but also testing the flow of data through the system as a whole. Essentially this is only a larger version of the procedure used in testing a single program because we are still testing a number of modules (in this case each is a program), the sequence in which they operate and the communication of data between them.

The main problems arising in system testing are:

(a) Misunderstandings of the functions of programs.
Program B presumes that program A has performed certain operations on data passed between them, when program A has not performed them at all or has performed something else or something in addition. This can only be guarded against by ensuring that the program specifications are spelled out in detail before the coding starts and that programs adopt a healthy mutual suspicion of one another during execution. This latter point implies that, at least during the testing phase, each program performs as much validation as is practical on any data which it inherits from other programs. This principle is a more global application of that referred to in Chapter 3 when considering parameter checking between program modules.

(b) Inconsistent descriptions of data.
Data will be passed from program to program by being transferred as parameters specified as part of a calling procedure or by being written to a file on an external peripheral by one program and read by others. In both of these mechanisms it is possible for one program to contain a data description which differs from the description contained in others. This difference may be in the size or type of data items concerned. This is best overcome by having a single definition of any major data item held once only within the system. This description may then be included in the source code of any program using that data. The inclusion may be effected by means of a copy facility which may be part of the syntax of the programming language used or may be done via a text editor. It also has the obvious benefit that any alteration to the format of the data items needs to be done in one place only, although it will usually then involve a re-compilation of all programs using that data.

(c) Availability of test data.
A significant problem in the testing of multi-program systems is that programs are not necessarily written in the sequence in which they will execute during the operation of the finished system. Hence a programmer wishing to test a program well down the execution sequence is often tempted to wait until all of the preliminary programs have been tested in order to obtain test data which has been passed through the system. Such a strategy is usually untenable in a large system. In effect, all parts of it need to be tested in parallel. This necessitates the creation of test files in advance of the programs which will ultimately produce them in the natural operation of the system. If this cannot be done easily by means of, say, a text editor, the system designer should commission the development of a software tool which operates as a test file loader. Such programs are easily written and the time spent on their development is repaid many times over. Unless the creation of adequate test data is made as painless as possible, programmers and users will tend to supply meagre test material which, because it does not exercise programs sufficiently, may allow errors to go undetected into the production system.

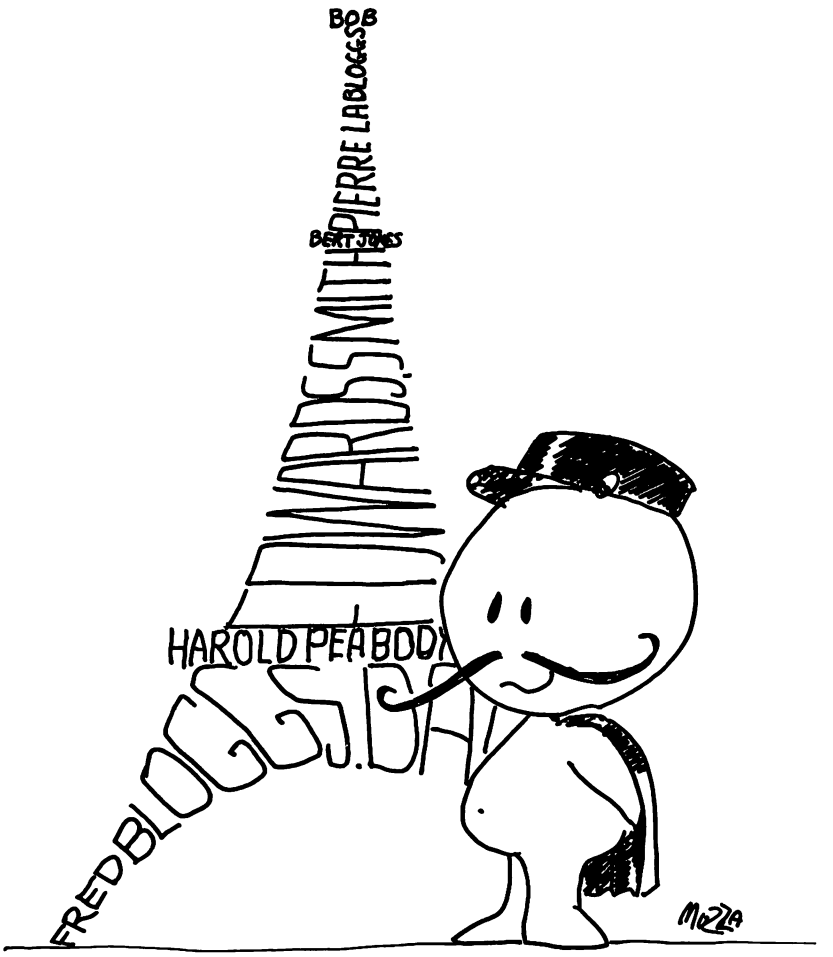
(d) Multiple versions of programs
As programs are compiled, tested, modified, re-compiled, re-tested, and so on, there is an ever-present danger that several versions of any one program may exist in the program library. It is essential that care is taken in performing a librarian function to

manage this procedure. This function may be carried out by the programmers themselves, by a person appointed as a librarian or by a software-automated process. However it is done, it must be done with as much attention to detail and documentation as the development of the system itself, otherwise the running of a previous version of a program which has subsequently been modified to correct certain errors will result in those errors once again being introduced into the system.

6.13 CONCLUSION

The importance of methodical and exhaustive testing of programs cannot be over-emphasized. The time taken and pain suffered during this phase of program development will be in inverse proportion to the care with which the algorithm was designed and the code written, that is, there should be more emphasis placed on 'antibugging' than 'debugging'. Attention to elegance in program structure and style will prevent errors from being introduced into the program at coding time, and mastery of these techniques must be regarded as more important than spending large amounts of time in searching for errors in programs as a result of poor initial programming.

7 DATA STRUCTURES



CHAPTER 7

7.1 DATA STRUCTURES

As Niklaus Wirth, one of the co-designers of the language Pascal, stated in the title of his recent book, *Programs = Algorithms + Data Structures.* The study of programming is of necessity as much concerned with the understanding of the nature and structure of data as it is with the structure and logic of algorithms. Programs are written to process data, and the programmer must have a clear understanding of this, his raw material.

These are the main principles relating to data as they affect the writing of programs.

7.2 ELEMENTARY ITEMS

Elementary, or single, data items may be considered under a number of headings.

Type

Although some contemporary languages allow the programmer to invent his own data types and define their related operations, there are a number of traditional data types found in most languages:

(a) Integer
Integers are numeric data items which are positive or negative whole numbers, for example, 1, 47, −193. Some programming languages place restrictions on the magnitude of integers which may be used in instructions. These restrictions are usually dependent on the size of the memory location of the computer on which the language may run.

Integers may be held within the computer's memory as a pure binary number or in a coded format such as ASCII, EBCDIC, BCD or packed decimal. Although the method of internal storage may affect the efficiency of a program, it will not have any bearing on the results of the use of such data items.

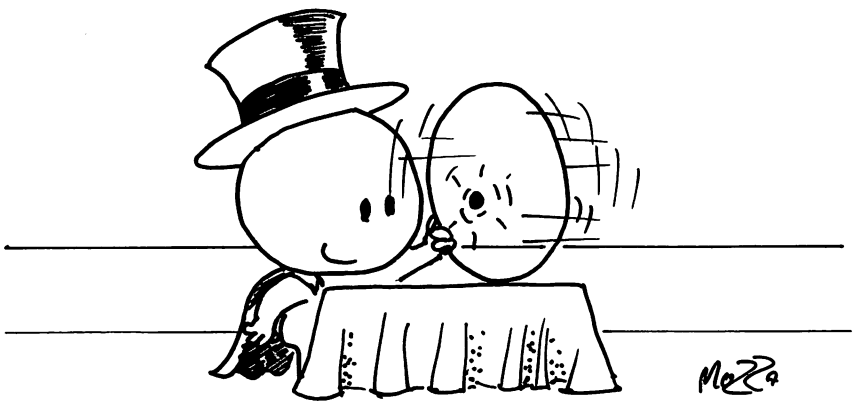
(b) Fixed point
Fixed point data items are numbers which have an embedded

decimal point but are essentially held within the computer in the same representation as for integers. For example, in translating an instruction involving the multiplication of an item by a fixed point value of 1.5, a compiler would typically generate the code necessary to multiply by an integer value of 15 and divide by the scaling factor of 10. Because whole decimal numbers can be represented with greater accuracy than fractions by a computer which works internally in binary, fixed point representation will give more accurate results to computations than will floating point representation.

(c) Floating point

Floating point data items are numbers which are held as binary fractions by a computer. The numbers are reduced to a normalized form consisting of a mantissa and an exponent, for example:

	Number	*Mantissa*	*Exponent*
12.3	$(=0.123\times10^{2})$	.123	2
123000	$(=0.123\times10^{6})$	.123	6
.000123	$(=0.123\times10^{-3})$	.123	−3



Floating point representation of data is used to overcome the restrictions placed on the magnitude of numbers by the size of a computer's memory locations; however, they do not give the same precision of arithmetic because of the inability of the binary numbering system to represent a large proportion of decimal fractions as exact binary values (just as 1/3 cannot be represented exactly as a decimal fraction).

(d) Character
Character data, sometimes referred to as 'string' data, may consist of any digits, letters or symbols which the internal coding system (e.g. ASCII, EBCDIC) of the computer is capable of representing. Many programming languages require character data to be enclosed by quotation marks when used in instructions, for example: PRINT "HAPPY NEW YEAR"

(e) Boolean
Boolean data items are used as status indicators (flags, switches) and may contain only one of two possible values: True or False. It is immaterial to the programmer what the internal representations of True and False are. The only allowable operations on Boolean items, sometimes called logical items, are setting them to True or False and testing them for those values.

Usage

There are two basic methods of using data items in a program:

(a) Constants
A constant is a specific value or character string used explicitly in an operation, for example:

```
Multiply ... by 47.5
Add 1 to ....
If .... = 13
Print "PLEASE INPUT TODAY'S DATE"
```

(b) Variables
A variable is a symbolic name assigned to a data item by the programmer. The actual location in the computer's memory used to hold that item is immaterial and its contents vary from time to time as a result of the program's operation, for example:

```
add .... to TOTAL
If RECORD_TYPE = ....
Print MESSAGE
```

7.3 DATA AGGREGATES

Data aggregates, or groupings, may be classified as hierarchical or tabular, the usual categories being:

(a) Files

A file is a collection of logical entities, or records, relating to one aspect of a computer system; for example, customers, products, motor vehicles, invoices, and so on. Records may be accessed by serially reading through the file, starting at the first record, or by directly locating a required record, depending on the storage medium used.

(b) Records

A record is a collection of elementary items, or fields, comprising one logical entity, that is, one customer, one product, one motor vehicle, and so on. The elementary items may be of differing types and sizes and can be accessed through a unique name assigned by the programmer. In addition, some programming languages allow the assignment of a symbolic name to the record as a whole.

(c) Arrays

An array, or table, is a collection of elementary items each of which is the same type, has the same size and is accessed by the same name. Items in a table are distinguished from one another by the use of a subscript or index indicating the position of the element in the array, for example:

ITEM (6), ELEMENT (23)

The subscript may be a variable and may then be used to access any item within the valid bounds of the array; for example:

ITEM (COUNTER), ELEMENT (INDEX)

(d) Strings

A string is essentially a stream of characters containing no inherent internal structure. A reader, for example, may look at the character string ITISWETONTHEMAT and discern several discrete words. The separation is from the reader's recognition of character patterns forming certain words rather than from any indication within the character string itself.

For a program to effect the same separation it would need to have a dictionary of words with which it matched the characters in the string. Such a program may come to the conclusion that the words contained in the string are IT IS WE TON THEM AT.

String processing relies on rules or conventions being able to be applied by a program in order to pack and unpack the components of a string. In the above example it would be a useful convention to separate words by space characters and hence allow the program to make a more likely diagnosis that the words contained are IT IS WET ON THE MAT.

7.4 ABSTRACT DATA STRUCTURES

An abstract data structure is one which is identified by its behaviour rather than by its physical structure. Examples of abstract data structures are:

Queues

A queue manages the storage of data so that the first item committed to the structure for storage is the first item retrieved, that is, first-in first-out.

Stacks

A stack stores data in such a fashion that the most recently added data item is the first item retrieved, that is, last-in first-out.

Linked Lists

A list stores data items so that the position of any element is only determined by means of a pointer from the logically preceding element. To process the entire list, one must access the first logical element, by a head-of-list pointer; thereafter each element leads on to the next.

Trees

A tree is a particular instance of a list and, by means of two or more pointers in each element, maintains a sequence or hierarchy among its constituents.

All of these structures will be dealt with more fully in a later chapter.

Multi-dimensioned Arrays

A multi-dimensioned array is one which consists logically of two or more dimensions, for example, rows and columns or rows, columns and pages, and so on. Items must then be accessed using a subscript for each dimension of the array; for example, ITEM (3, 4) may represent an element in the third row and fourth column of the array.

7.5 DATA STRUCTURES AS MODELS

One of the functions performed by data structures in a computer system is the internal representation, or modelling, of external entities in the world around us.

Programs are required to process, and therefore data structures are required to represent, such situations as:

- Seating plans for a theatre.
- A railway or road network.
- A queue of tasks awaiting processing.
- Text comprising sentences, paragraphs and sections in a report (for 'word processing' applications).
- Plots on a radar screen for radar operators' training simulation exercises.
- Game playing situations: board games such as chess; tunnel-and-cave games such as 'Adventure'; and seek-and-catch games such as 'Space Invaders'.

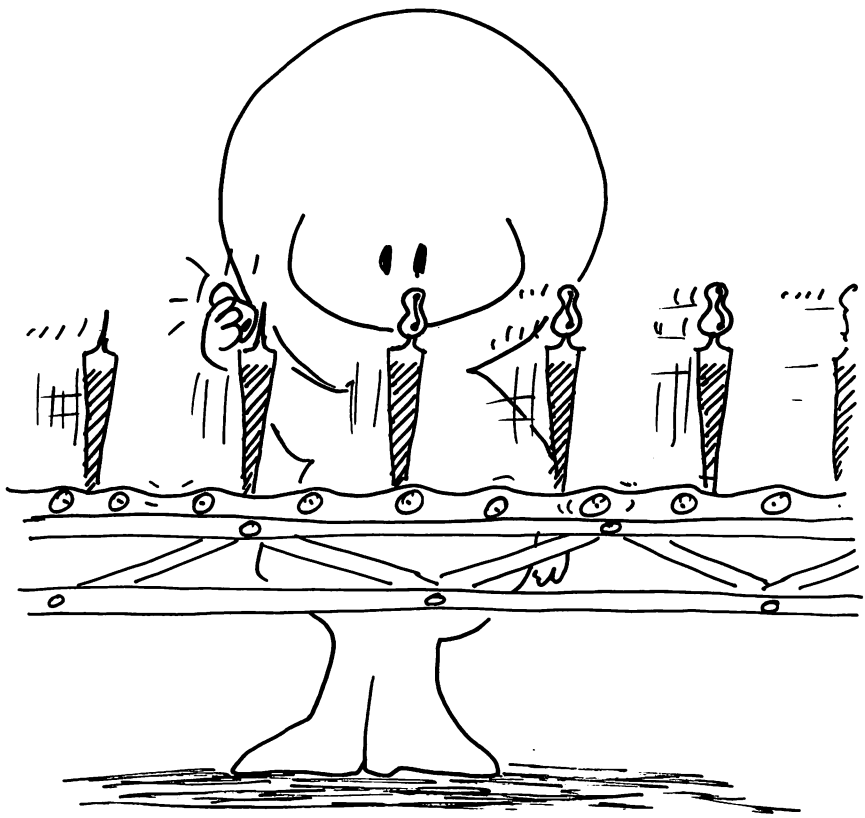
In all these situations a program must contain not only a suitable algorithm to perform the required data processing but also an appropriate data structure which most easily allows the program to execute those operations which reflect the manipulation of the real world entity.

This is a further argument in favour of the need for programmers to be familiar with the handling of a wide variety of data structures.

7.6 CONCLUSION

The two most important factors in the design of programs are algorithms and data structures. The preceding portion of this book has dealt with the construction of algorithms. It is essential that a programmer understands the nature and structure of the data which his program must process. It is also essential that the programmer is able to construct and manipulate data structures which will enable his program to function efficiently.

8 SERIAL FILE PROCESSING



CHAPTER 8

8.1 SERIAL FILE PROCESSING

A serial file is one in which records may only be accessed in the order in which they are physically held on the storage medium. Examples of such files are those held on punched cards, punched paper tape and magnetic tape and those produced on a printer. Each of these media allows only serial access to data. Disk files may be organized to permit serial access only, although the nature of this storage medium permits direct (random) access to its records.

When the records on a serial file are arranged in a particular sequence, that is, in ascending or descending order of a chosen key field, the file is usually referred to as a 'sequential' file. It is important to note, however, that the terms 'serial' and 'sequential' are not necessarily synonymous.

Three areas associated with processing serial files will be examined in this chapter:

(a) processing single input files,

(b) updating master files, and

(c) producing printed reports with control breaks.

Much of this approach to file processing is somewhat dated at present as it belongs to batch-processed systems which concentrate on a validate-sort-update-report cycle. Although very few such systems are being currently developed, there are sufficient examples of them remaining in service to warrant some consideration of their associated techniques.

8.2 PROCESSING INPUT FILES

The first task associated with the processing of an input file will normally involve the reading and processing of a file header record. Such a record may have an identifier or period indicator which may have to be validated and found to be correct as a prerequisite to proceeding with the following records.

The processing of the body of the file should follow the 'priming read' logic introduced in Chapter 1, Example 1.4. This involves the reading of the first record before entering the main processing loop and thereafter reading each successive record as the last operation within the loop.

A serial file may typically end with a file trailer record which contains one or more record counts and/or balancing totals which serve as a check on the accuracy of preceding file processing, for example, the total number of all customers' records on a file together with an accumulated total of their outstanding balances. These values in the file trailer record would be compared to similar totals accumulated independently by the program reading the file. Any discrepancies would be reported for follow-up action. The skeleton structure of such a program would be as shown in Figure 8.1.

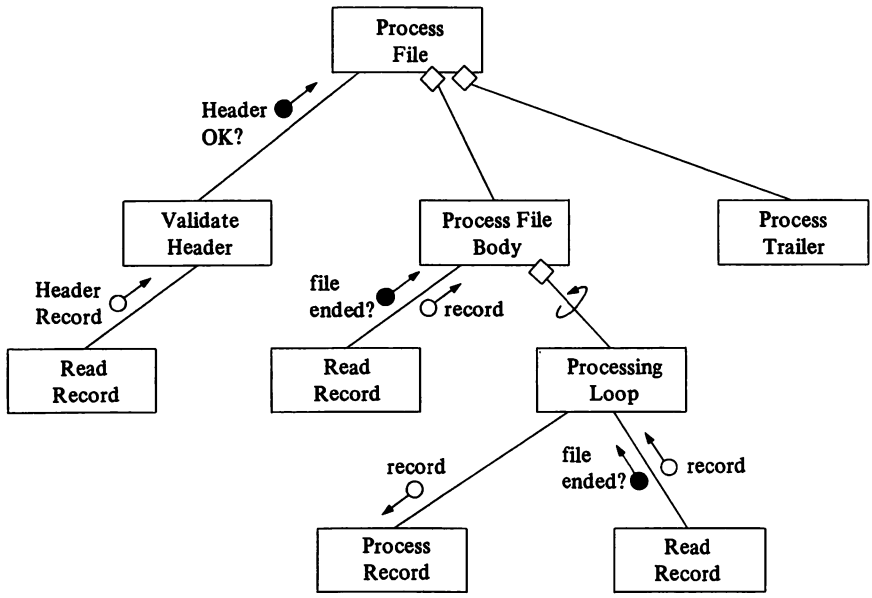


Figure 8.1

The resulting algorithm would be (in general terms):

Process File
Do Validate File Header
If Header_OK = True
Do Process File Body
Do Process File Trailer
Stop

```
Process File Body
    Do Read Record
    Perform until Record_type = File_trailer
        Do Processing Loop

Processing Loop
    Do Process Record
    Do Read Record
```

To avoid the multiplicity of separate modules which this approach creates, a more concise algorithm may be expressed by amalgamating the control logic into a single module, for example:

```
Process File
    Do Validate Header
    If Header_OK = True
        Do Read Record
        Perform until Record_type = File_trailer
            Do Process Record
            Do Read Record
        Do Process File Trailer
    Stop
```

In general, the structure of the algorithm processing a serial file should correspond with the structure of the file itself. The structure of the file for the example above is shown in Figure 8.2.

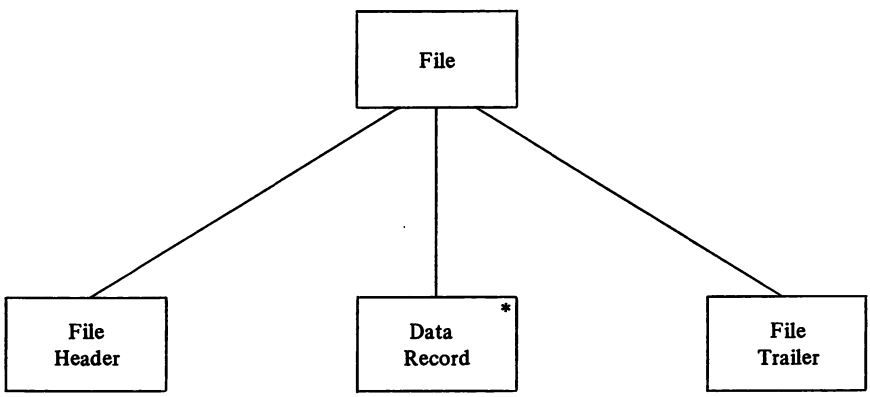


Figure 8.2

The asterisk (*) is used here to indicate the repetition of this element in the structure. A diamond (◇) will be used later to indicate a choice between several possible alternatives or the possible existence of an optional record. The structure of the algorithm was essentially as shown in Figure 8.3.

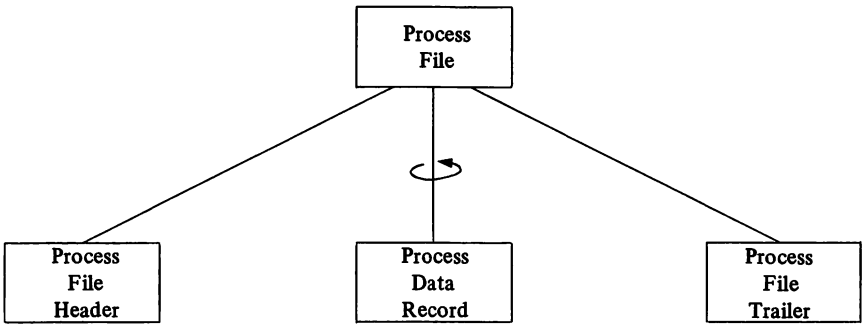


Figure 8.3

If this approach is adhered to, there should be one module of the program which controls all of the processing for an associated record. If that record is repeated in the file, its processing module would be built into a loop in the program.

A more complex case arises when the file contains records in groups, each group possibly consisting of a variable number of varying types of records. Consider the following example.

> A file contains records relating to customers of a certain organization together with any invoices outstanding for those customers and possibly special delivery instructions for goods sent to them and special credit arrangements for discount or extended payment terms. The set of records for any one customer will comprise:
>
> Name and Address Record — always present
> Invoice Record — as many (could be zero) present as there are outstanding invoices for this customer
> Aged Balance Record — always present
> Delivery Instructions Record — optional
> Special Credit Arrangements Record — optional
>
> In addition, the overall file itself contains a File Header and File Trailer record.

The structure of the file may be represented as in Figure 8.4.

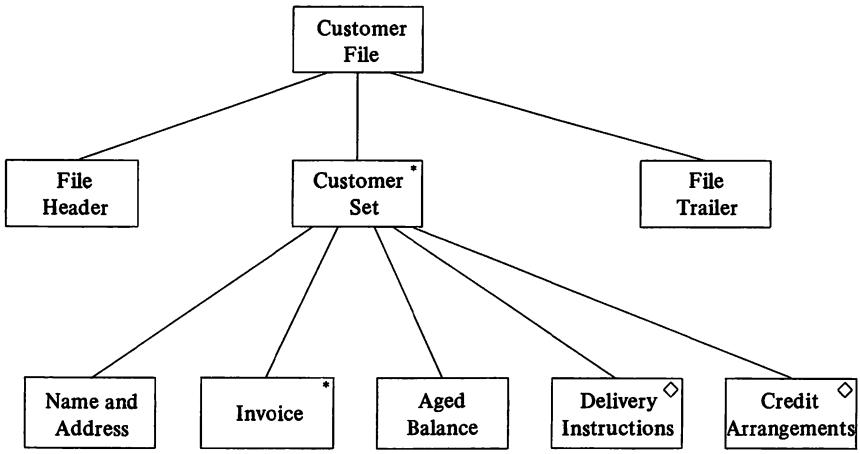


Figure 8.4

Based on the structure of the data, the structure of the algorithm should essentially be as in Figure 8.5.

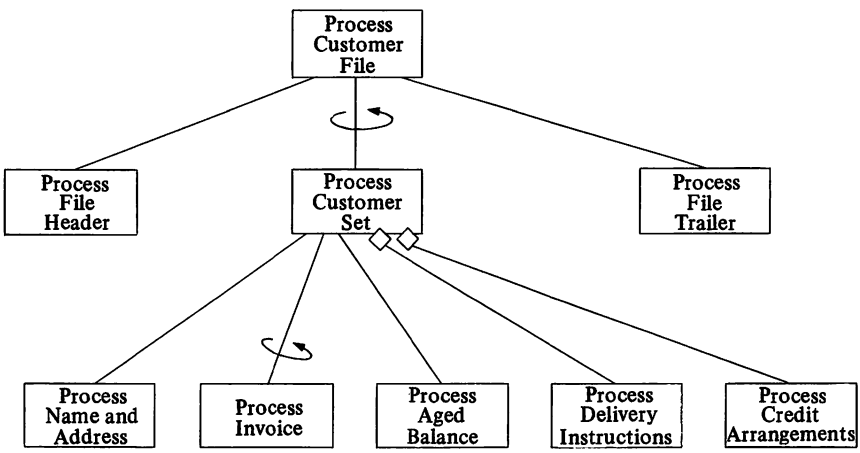


Figure 8.5

When combining this process with the principle of a priming read, it must be remembered that there are nested serial processing operations here. There is the serial processing of the customer sets within the file as a

whole and, within each set, the serial processing of a number of possible invoice records and of a number of possible optional records. All of these operations will need a priming read. A skeleton algorithm is set out below which presumes that the file structure is valid, that is, that no essential records are missing and no records are encountered other than those whose types are known. In reality, a programmer would need to check for breaches of both of those assumptions.

```
Process Customer File
    Do Validate File Header
    If Header_OK = true
        Do Read Record (Priming read for Customers)
        Perform until Record_type = File_Trailer
            Do Process Name_and_Address Record
            Do Read Record (Priming read for Invoices)
            Perform until Record_type = Aged_Balance
                Do Process Invoice Record
                Do Read Record
            Do Process Aged_Balance_Record
            Do Read Record (Priming read for optional records)
            Perform until Record_type = Name_and_Address
                                        or File_Trailer
                If Record_type = Delivery_Instructions
                    Do Process Delivery Instructions Record
                else
                    Do Process Special Credit Arrangements Record
                Do Read Record
        Do Process File Trailer Record
    Stop
```

8.3 UPDATING SERIAL MASTER FILES

The most common form of updating carried out on serial files is the technique known as father/son updating. Transactions are applied against a master file and a new generation of the file is produced. Customarily, at least three generations of the master file, together with the corresponding transaction files, are retained as a security measure in case of data loss or corruption.

The underlying principles of this procedure are:

- The master file records are in sequence (usually ascending) on a key field (or fields).

- The transaction file records have been sorted into the same key sequence as the master file (the primary sort key).
- There may be zero, one or several transactions for any master record.
- Transactions fall into three broad categories: amendments to existing master records, deletions of existing master records, and insertions of new master records.
- In the case of several transactions for any one master record, they are sequenced on a secondary key which is usually the transaction type or the date on which the transaction occurred.
- Transactions are applied against the master records on the basis of attempting to match their respective primary sequence keys (e.g. customer number, product code, car registration number, etc.).
- When each file ends, its record key has placed in it a 'padded' value equal to the largest possible content (e.g. 999999) which the field could hold, and a match between master and transaction records of this key value signifies the end of the procedure.
- It is customary for the transaction file to contain a file generation number in its File Header Record, this number being required to match that in the File Header Record of the master file to ensure that the transactions are updating the correct edition (generation) of the master file.

A summary of the actions required is set out in Table 8.1.

Table 8.1

Transaction Type	*Matching Master Record Found*	*No Matching Master Record Found*
Amendment	Action A	Action B
Deletion	Action C	Action B
Insertion	Action D	Action E

Actions:

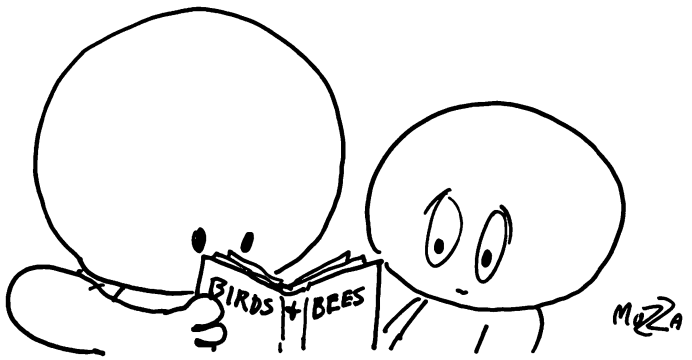
A: Amend the master record with the data contained in the transaction.

B: Reject the transaction as an error because it purports to relate to a non-existent master record.

C: Flag the master record as deleted and do not write it to the updated file.

D: Reject the transaction as an error because it is attempting to insert a duplicated master record.

E: Create a new record ready to be written to the updated master file.



An algorithm for a standard father/son update:

```
File Update
    Do Update Initialization
    If OK_TO_PROCEED = true
        Do Read Master
        Perform until END_OF_JOB = true
            Do Update Procedure
        Do Update Finalization
    Stop

Update Initialization
    Do Read Master          }
    Do Read Transaction     }   Read respective
                                File Header Records.
    If file generations match
        OK_TO_PROCEED ← true
        Write file header record to new master file
        END_OF_JOB ← false
    else
        OK_TO_PROCEED ← false
```

Read Master

Read a record from the master file into an input 'buffer' area
If End_of_file has been reached
    Place the 'padded' key value in the key fields of both input and output record buffer areas
else
    Copy the record read into the record buffer area for the output master file
    (*Note:* This will not destroy the copy of the record in the input buffer.)

Read Transaction

Read a record from the transaction file
If End_of_file has been reached
    Place the 'padded' key value in the key field of the transaction record (buffer) area

Update Procedure

Do Read Transaction
Perform until Key field of the master record in the *output* area ≥ key field of transaction record
    Do Master Key Low
If Key field of master record in the *output* area > key field of transaction record
    Do Master Key High
else (i.e. keys are equal)
    If transaction (or master) key = 'padded' value
        END_OF_JOB ← true
    else
        Do Master Key Match

Master Key Match

If transaction type = Deletion (Action C)
    DELETE_RECORD ← true
else
    If transaction type = Insertion
        Report transaction as an error (Action D)
    else (i.e. transaction type = amendment)
        Update master record in the *output* record area with the data from the transaction record (Action A)

Master Key High

If transaction type is not = Insertion
Report transaction as an error (Action B)
else
CREATE_RECORD ← true
Create a master record in the *output* record area from the data in the transaction.
(Note: This will obliterate the copy of the latest master record read from the input file; however, another copy of that record is still retained in the *input* master record area.)

Master Key Low

If DELETE_RECORD = true
DELETE_RECORD ← false
else
Write the record in the *output* master record area to the new master file
If CREATE_RECORD = true
CREATE_RECORD ← false
Copy (again) the record in the *input* master record area to the *output* master record area
else
Do Read Master

Update Finalization

(*Note:* At this point the File Trailer Records for both the master and transaction files should be in their respective buffer areas.)
Carry out any reconciliation procedures appropriate for the data within the trailer records.
Write a trailer record to the new master file.

8.4 PRINTED REPORTS WITH CONTROL BREAKS

Reports are often printed requiring certain totalling procedures to be carried out each time a 'control' field changes value. A sales report, for example, may relate to many salesmen working in many districts in each of several states. Totals of goods sold, and so on, may be required for each salesman in each district in each state together with a grand total for the report as a whole, for example:

State: VICTORIA			
Districts: GIPPSLAND			
BROWN	2000		
JONES	4000		
SMITH	3000	9000	
WIMMERA			
HUNT	4000		
JONES	6000	10000	19000
State: QUEENSLAND			
Districts: GOLD COAST			
ROSE	5000		
WHITE	7000	12000	
CAPRICORNIA			
GREEN	2000		
ROSE	4000	6000	18000
			.
			.
			.
GRAND TOTAL			200000

The production of such a report would presume that the sales records were sorted on a three-part key of State Code (Highest) — District Code — Salesman Code (Lowest). Whenever there is a change of value between the sort keys of successive input records (i.e. a 'control break') a total must be printed for the entity whose value has changed and totals for all entities subordinate to the changed key. In the above example, the printing requirements are:

Change of:	*Totals to print:*
Salesman Code	Salesman
District Code	Salesman District
State Code	Salesman District State

An algorithm for a general case involving a three-level control structure is given below. The reader should be able to extrapolate from that to a case involving more (or fewer) control breaks. This is a further instance of serial

(sequential) file processing and hence the priming read principle applies again.

Where the algorithm refers to a change of one of the components of the key, it means the detection of a different value in that component of the key of the latest record read from that component of the Control Key set up from some preceding record.

```
Print Report
Clear totals to zero at all control levels.
Read Record
Place record key (i.e. all 3 components) in Control Key
Perform until End-of-file is reached
    If highest key component has changed value
        Do Lowest Level Change
        Do Middle Level Change
        Do Highest Level Change
    else
        If middle key component has changed value
            Do Lowest Level Change
            Do Middle Level Change
        else
            If lowest key component has changed value
            Do Lowest Level Change
    Accumulate lowest level total(s) for
        current record
    Read Record
Do Lowest Level Change  }
Do Middle Level Change  }   These totals are required because
Do Highest Level Change }   the end of file is reached.
Print Grand Total
Stop

Lowest Level Change
    Print total(s) for lowest level entity
    Add lowest level total(s) to middle level total(s)
    Clear lowest level total(s) to zero
    Place lowest level key value (in the latest record read) in lowest level
        of the Control Key
```

Middle Level Change

Print total(s) for middle level entity
Add middle level total(s) to highest level total(s)
Clear middle level total(s) to zero
Place middle level key value (in the latest record read) in middle level of theControl Key

Highest Level Changes

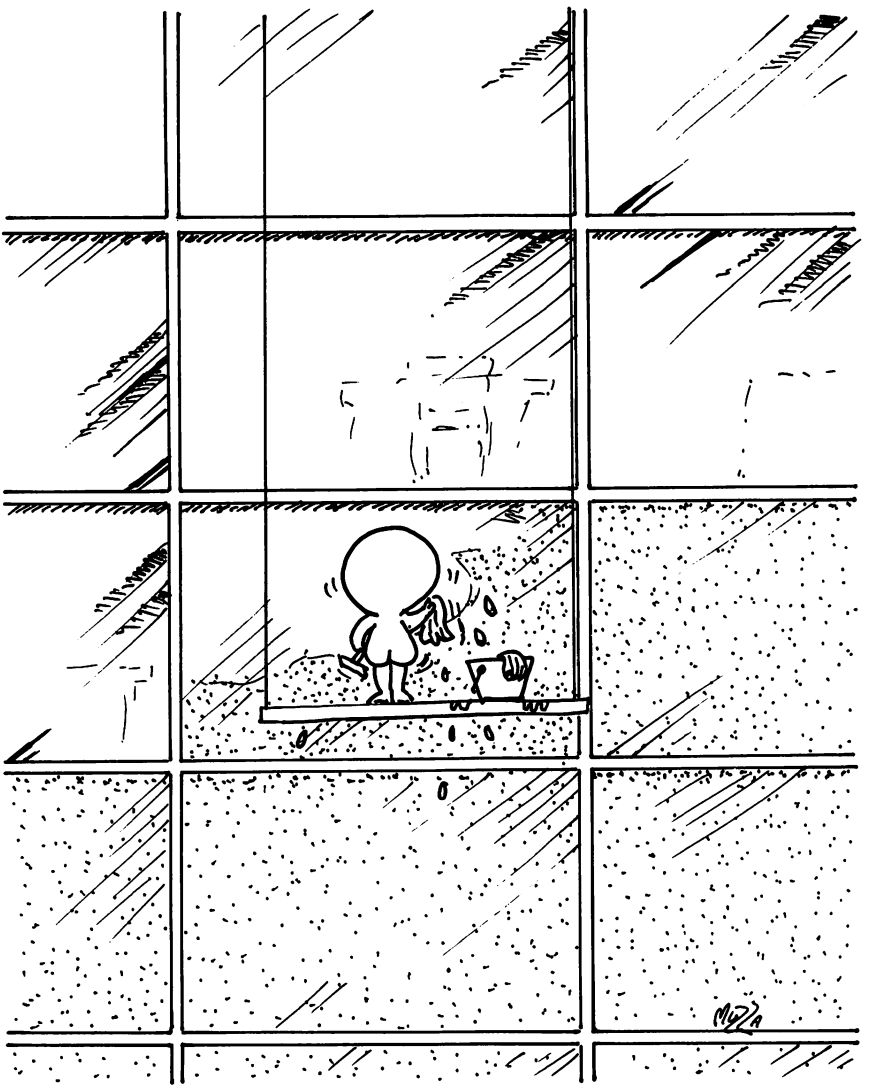
Print total(s) for highest level entity
Add highest level total(s) to grand total
Clear highest level total(s) to zero
Place highest level key value (in the latest record read) in middle level of the Control Key

8.5 CONCLUSION

Serial file processing has historically constituted the majority of commercial data processing. Despite its disappearance in favour of on-line processing, it is worthwhile being familiar with the principles involved and having access to some of the most common algorithms employed.

Many of the principles involved are also carried across into other areas of programming, particularly the priming read concept, which is applicable to any series of similar operations.

9 ARRAY PROCESSING



CHAPTER 9

9.1 ARRAY PROCESSING

An array, or table, is a group of elements all of the same type and size and all having the same name. Any individual element is accessed by nominating its position within the array, for example, ITEM (10) or ITEM (POINTER).

The types of arrays which may be constructed and some consideration of the uses to which they might be put will be dealt with in this chapter, and algorithms will be provided for the operations most commonly performed on them.

9.2 UNORDERED LINEAR ARRAYS

A linear array, sometimes referred to as a linear list (as opposed to a linked list, see Chapter 10), is the simplest table structure able to be created and manipulated by a program. It consists of a contiguous series of data items, each able to hold one or more pieces of data. An unordered linear array is one in which the data items are held in no particular sequence. It is usually assumed that as many items as are currently stored in the array start at the first element and continue without gaps for as many elements of the table as are required.

Insertions/Deletions

In an unordered linear table, additions to the table are put in after the last element occupied. But how is that vacant element found? One method would be to examine the contents of each element in the table, starting with the first element, until one was found which contained no data, this being signalled by its contents containing some predetermined value which indicates its vacancy. A better method would be to maintain a pointer the contents of which would always indicate the position of the first vacant element in the table. The concept of such a pointer is of importance in handling many data structures and will be referred to as a 'free pointer'. The free pointer must be incremented or decremented each time an item is added to or deleted from the table and is the prime means of determining when the table is full.

Deletions in such a table would normally be followed by some procedure to fill the gap in the table. This could be done by 'shifting up' the following elements but if a free pointer were maintained, this could be used to remove the last element currently held in the table and insert it in the gap caused by the deletion. This procedure is preferable as the shifting up operation would involve an average of $n/2$ movements of data, where n is the number of elements in the table.

Access to Data in the Table

In an unordered linear table, the only means of locating an item of data in the table is by a linear search. This involves the examination of each element of the table starting with the first, and proceeding until either:

(a) the required item is found, or
(b) the data in the table is exhausted, or
(c) the physical end of the table is reached.

Hence, to locate an item which is in the table, the average number of elements which must be examined is $n/2$ (where n is the number of occupied elements in the table) and to detect the absence of an item from such a table will involve n examinations.

A typical algorithm to perform such a linear search would be:

```
Linear Search
    INDEX←1
    LAST_ITEM←Furthest element currently occupied
    SEARCH_DONE←False

    Perform until SEARCH_DONE = True

        If item sought = element (INDEX)
            ITEM_FOUND←True
            SEARCH_DONE←True
        else
            If INDEX < LAST_ITEM
                INDEX←INDEX + 1
            else
                ITEM_FOUND←False
                SEARCH_DONE←True
```

Notes

(a) The status variable ITEM_FOUND will indicate the success or failure of the search on completion of the execution of the algorithm.

(b) An alternative approach would have been to set ITEM_FOUND to, say, False before starting the loop and then to alter its value to True only if the item sought was found. I consider that to be a weaker solution than explicitly setting the variable to True or False within the loop when one of the terminating conditions is encountered.

(c) A further alternative would have been to control the loop with two variables rather than one as shown, for example:

```
.
.
.
Perform until ITEM_FOUND = True
              or INDEX > LAST_ITEM
```

Again, I consider it preferable to control loops with a single variable to avoid the complications likely to arise in evaluating compound conditional expressions (see Figure 9.1).

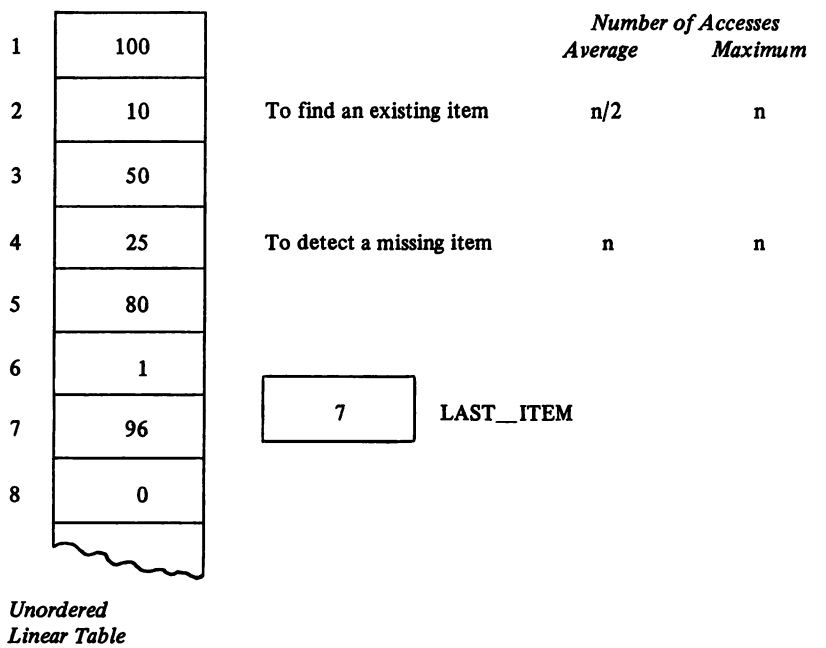


Figure 9.1 Linear Search

9.3 ORDERED LINEAR ARRAYS

An ordered, or sequenced, linear array is one in which data items are held in a specific ascending or descending sequence.

(a) Insertions/deletions
In an ordered linear table it is essential that data items be maintained in sequence. Additions of new items will involve the shifting along of current items to make room for the insertion in the appropriate element. Deletions will similarly involve a shifting operation to fill the gap.

(b) Access to data in the table
Data in the table may be sought by a linear search as described above, but a much more efficient method is by means of a binary search.

A binary search is a technique whereby the area under consideration is continually halved until the item sought is found or its absence is detected. Unlike a linear search, it takes no longer in a binary search to detect the absence of an item than to detect its presence.

An algorithm for a binary search on an array in which data was held in ascending sequence would be:

Binary Search

```
LOW←1
HIGH←Position of the furthest occupied element in the
array
SEARCH_DONE←False

Perform until SEARCH_DONE = True
   INDEX←½*(HIGH + LOW)
   If item sought = element (INDEX)
      ITEM_FOUND←True
      SEARCH_DONE←True
   else
      If item sought < element (INDEX)
         HIGH←INDEX − 1
      else
         LOW←INDEX + 1
   If LOW > HIGH
      ITEM_FOUND←False
      SEARCH_DONE←True
```

Using a binary search, the maximum number of elements which must be examined to detect the presence or absence of any item is given by $1 + \log_2 n$ where n is the number of elements in the table.

It should be noted that although a binary search is possible for an ordered table, a linear search will be faster if the table holds less than (approximately) 100 entries. The reason for this lies in the amount of computational overhead involved in a binary search compared with a simple subscript increment to accomplish a linear search.

9.4 DIRECTLY ACCESSED ARRAYS

The principle underlying the operation of a directly accessed table is that the location of any item within the table is indicated by the item itself. A simple example would be the case of 100 descriptions associated with 100 items having a code number ranging from 1 to 100. Hence, item 36 would be found in element 36 in the table, item 92 in element 92, and so on (see Figure 9.2).

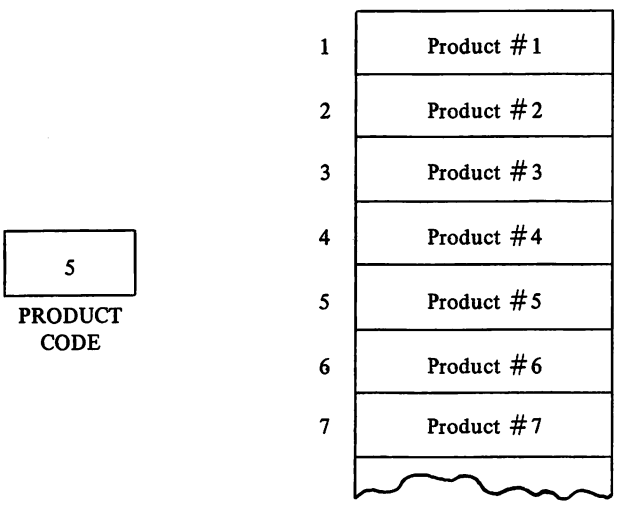


Figure 9.2 Directly Accessed Table

9.5 HASH-ADDRESSED ARRAYS

The advantage of a directly accessed array was that no searching algorithm was needed to locate data held in the array. The disadvantage, however,

was that the data stored needed to be such as to provide an exact correspondence between some attribute of the data (e.g. a product code) and the position in which it was stored. Such a correspondence does not often occur in practice.

It would be more common to find, for example, that an organization's range of 600 products had 5-digit identification codes spread more or less randomly between 1 and 99999 and perhaps even containing non-numeric characters. If it is still desirable to pursue speed of retrieval of data by a direct access technique rather than a searching technique, a process referred to as 'hashing' may be carried out on the key (i.e. the unique identifier) of the data item. Hashing is a procedure which computes a table address from a given key. Because the procedure is a numerical operation, the key must be expressed as a numeric quantity. This implies that any non-numeric characters within the key are reduced to numeric values by either ascribing an arbitrary value to each character (e.g. A = 1, B = 2 .... Z = 26, etc.) or by operating on the binary coded internal representation of the character as held in ASCII, EBCDIC, and so on.

The numeric key is then subjected to a procedure which computes from it a number within the range of 1 to the size of the array. This procedure is referred to as a 'hashing function'. A commonly used hashing function is one which uses a division technique and computes the table address from the remainder, for example:

1 Divide the key of the item to be stored or retrieved by the largest prime number which is not greater than the number of elements in the table.

2 Ignoring the quotient, add 1 to the remainder of the division to give the table address.

For example:

Size of table: 100 elements
Largest prime number $\leqslant$ 100: 97
Key of item to be processed: 30679

$$\frac{30679}{97} = \text{Quotient } 316$$

Remainder 27

Array address = 27 + 1
= 28

i.e. data item 30679 would be stored in the 28th element of the array.

The advantage of such a technique is that it is much faster than a searching operation. The obvious disadvantage is that there are likely to be a number of keys for which the hashing function produces identical array addresses. These are referred to as synonyms or collisions and must be handled by storing the collided items in elements other than those indicated by the hashing function.

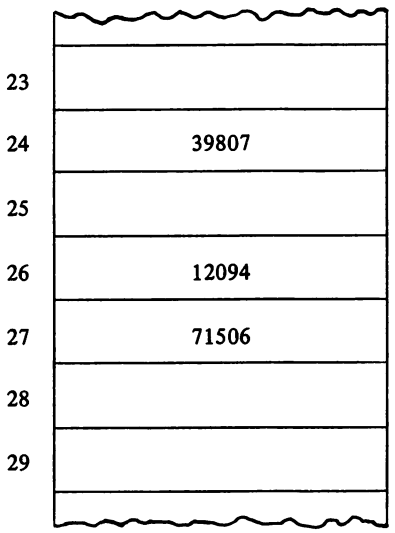


Figure 9.3

Consider a table which currently contains the items as seen in Figure 9.3. Three further items are to be placed in the table, all of them generating the same element address via the hashing function:

Key of item to be stored:	14796	20493	37108
COMPUTED ADDRESS:	23	23	23

Three typical approaches to the solution of storing the collided items are:

(a) Store the first item in its 'home' address and store the collisions in the next vacant table element located via a linear search. They would be subsequently retrieved by the same procedure (see Figure 9.4).

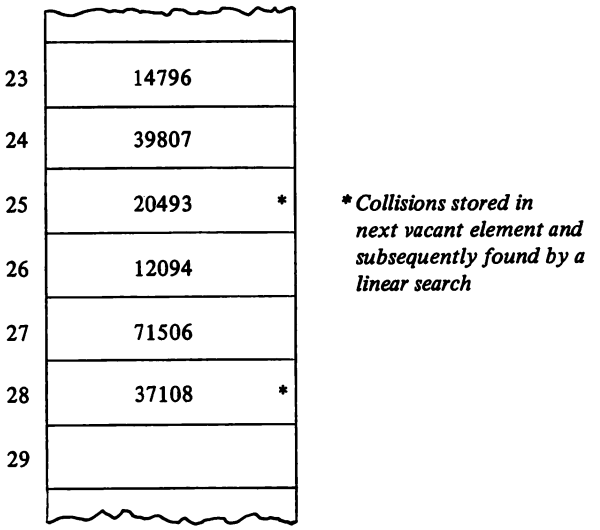


Figure 9.4

(b) Store the items as indicated in (a) but maintain a series of pointers to the collided items (see Figure 9.5). This method shortens the retrieval process by eliminating the necessity for the linear search as needed for method (a).

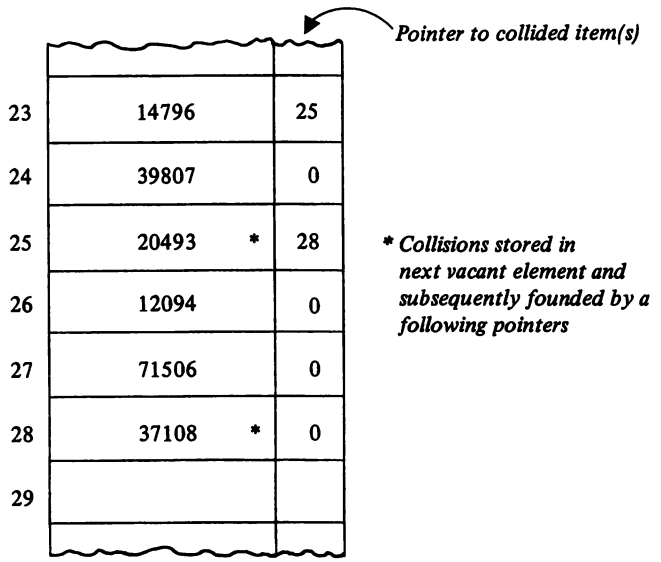


Figure 9.5

(c) Store the first item in its 'home' address and store collisions in a separate overflow table (see Figure 9.6). This method eliminates the problem which occurs with both of the previous methods, that is, that each collision stored in the main table occupies an element which another data item would be likely to compute as a 'home' address and find it already occupied.

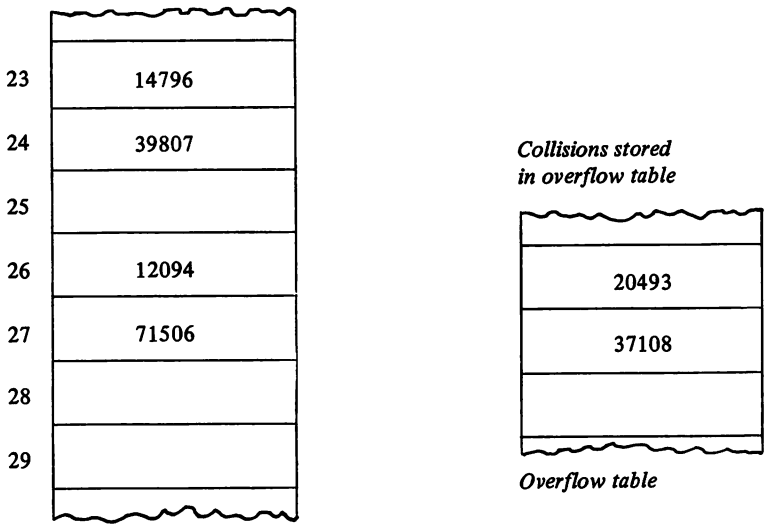


Figure 9.6

9.6 INDEXED LINEAR TABLES

A compromise between a single linear table and a directly addressable table may be reached by the use of a directly addressable index to an unordered linear table. The following example should illustrate the technique.

Consider a case where a table of product descriptions, each of 40 characters, must be maintained by a program which must translate a three-digit numeric product code into its appropriate alphanumeric description. The three-digit code allows for 1,000 possible values, but it is found that only 200 of these values are actually used and are randomly spread between zero and 999.

Three alternatives may be considered:

(a) A linear table of 200 elements each of 43 characters (the three-digit product code + the 40-character description).

Storage requirements: $200 \times 43 = 8600$ characters
Access time:

(i) If table is unordered, access is slow owing to the linear searching operation, but table insertions and deletions are simple;

(ii) If table is ordered, access may be via a binary search which will take up to 10 table accesses + related computation overhead for each search, and table insertions and deletions are lengthy procedures.

(b) A directly addressable table may be constructed using either 1,000 elements of 40 characters (= 40,000 characters of storage!) and enabling each description to be retrieved by using its code as a subscript, or by using only 200 elements of 43 characters and employing a hash function for computing the address of each description together with a collision-handling routine (both of which are time-consuming).

(c) Two tables may be constructed:

(i) An index table of 1,000 elements each of three digits (= 3,000 characters of storage). For each product code used between 0 and 999, the corresponding element in this index table would contain a pointer to its description held in the second table, that is, the subscript of the element in the second table which held the description relating to that code.

(ii) An unordered linear table of 200 elements each of 40 characters (= 8,000 characters of storage), each element holding one description.

The access to any description would therefore involve only two table accesses and no computation or searching procedure, the operation being:

(i) use the product code to directly access the index table and extract the subscript of the related description;

(ii) use that subscript to directly access the second table and retrieve the appropriate description.

It can be seen that this third alternative provides an access time almost as fast as a single directly addressable table without paying a large penalty in storage space beyond that required by an unordered linear table and without the complexities of maintaining an ordered linear table (see Figure 9.7).

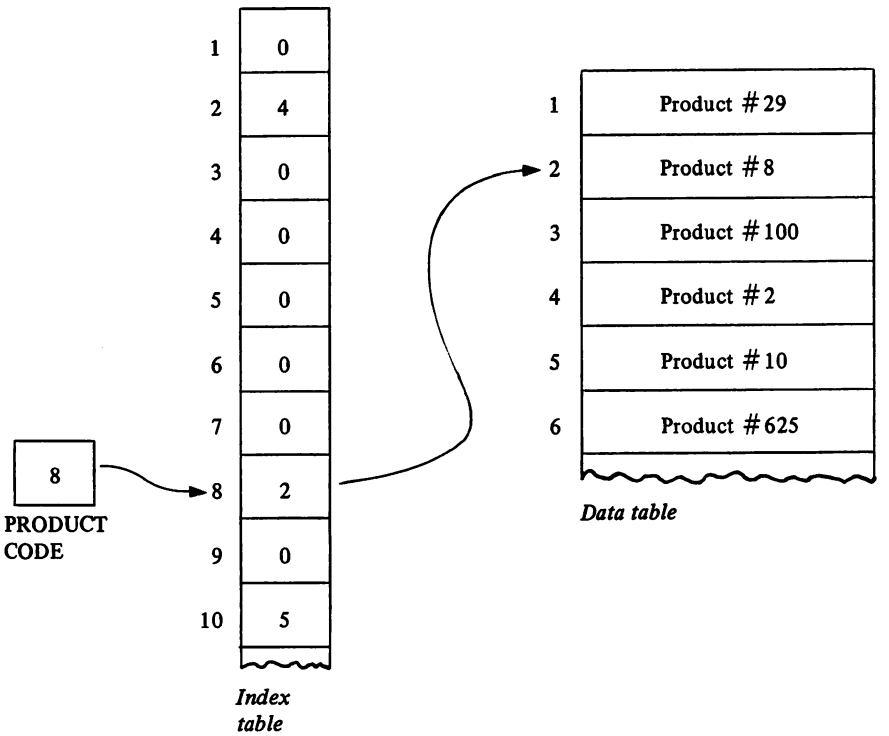


Figure 9.7 Indexed Array

9.7 MULTI-DIMENSIONAL ARRAYS

Most programming languages give the programmer the facility to construct arrays of more than one dimension. Thus, two-dimensional arrays may be thought of as having rows and columns. Three-dimensional arrays may be thought of as having pages, rows and columns, and so on. Remember that the pages, rows, columns, and so on, are only a notional concept to aid the programmer. The array held in memory is still essentially a one-dimensional array. The physical arrangement of the array elements is referred to as 'mapping' and is effected in different ways for different languages, as, for example, in Figure 9.8.

When accessing elements in a multi-dimensional array, a separate subscript is required for each declared dimension. The subscripts must be nominated in the hierarchy required by the conventions of the programming language used.

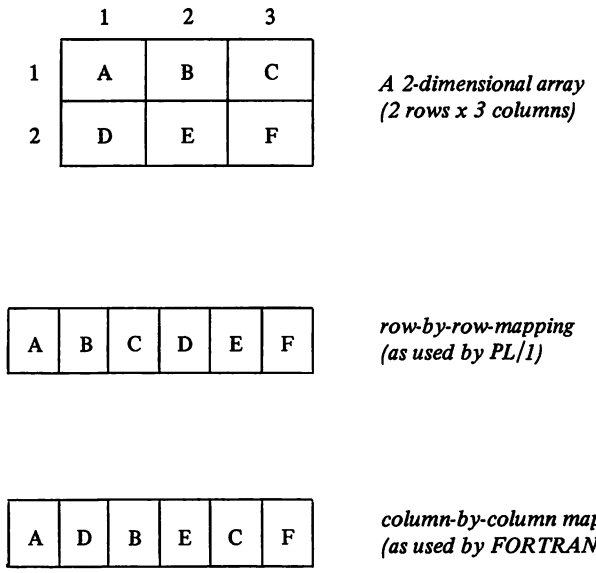


Figure 9.8

Hence, in a language which uses column-by-column mapping, the array in Figure 9.8 array would be declared as:

ARRAY (3, 2)
i.e. 3 columns each of 2 rows.

The element containing C would be accessed as element (3, 1) and that containing D would be element (1, 2).

Whatever the mapping convention, the quoted order of the subscripts when accessing an element in an array must be the same as the order used in the declaration, or creation, of the array.

9.8 PSEUDO-ORDERED AND SELF-ORGANIZING ARRAYS

From the previous discussion it can be seen that there is a conflict which must be resolved between speed of access to data and ease of maintenance of the data held in an array. Unordered arrays provide for simple maintenance procedures (i.e. additions and deletions) but do not provide speedy access, being limited to linear searching. Ordered arrays provide rapid access via binary search but make the maintenance of the sequence of the items a burden.

There is a further technique which provides a compromise in this area. A linear search is easy to implement but slow to execute on large arrays because the items sought may be located randomly throughout the array. An alternative approach is to sequence the data in order of frequency of access rather than according to the primary characteristic of the data itself (i.e. numeric or alphabetic sequence). This implies that the most often sought data items are held towards the start of the array and those least often accessed towards the end. A linear search will now enable the most 'popular' data to be found rapidly by a procedure which does not require the computational overhead of a binary search.

A problem still remains, however, with the maintenance of the array. New items inserted in the table would seem to require a prediction about their future frequency of access in order to locate them appropriately. Similarly, deleted items would seem to still require the 'shifting along' procedure dictated by an ordered array. Another apparent problem is the changing access patterns to the data in the array. As items become more or less frequently accessed, they will need to move towards the front or back of the array, respectively.

These problems of maintenance can be overcome in a variety of ways, two of which are outlined below:

(a) Dynamic reorganization:
When any item of data within the array is accessed, it is exchanged with the item immediately before it in the table. With this procedure, frequently accessed data will automatically shift towards the top of the array, displacing less frequently accessed data.

At any given instant, the positions of the data items in the array will be an accurate reflection of their current 'popularity'. With this operation, new items can be added to the array at the lowest position and will rise automatically to whatever level is appropriate to their access pattern. Deleted items may be flagged and removed when they eventually reach the bottom of the array.

The searching performance which this method of organization enables is only achieved at the cost of some additional housekeeping procedures (the swapping of adjacent items of data) for *every* access to the array. This method is used by some operating systems in managing memory space allocation to programs.

(b) Periodic reorganization:
This procedure is essentially similar to dynamic reorganization in its aim to keep the most frequently accessed data at the head of the

array. It differs from the former method in that the re-sequencing is done at prescribed intervals rather than every time data is accessed. Associated with each data item in the array is a counter which is incremented each time the data item is accessed. At the end of a period, which may be a month, a day or an hour, depending on the urgency of the task involved, the array is re-sequenced on the contents of the counters and the counters all returned to a value of zero.

This procedure provides an array of data which is never as up to date in access frequency sequence as that provided by dynamic reorganization but does not require the housekeeping overhead of that method.

9.9 SORTING

The evolution of computer sorting techniques parallels the evolution of computers. Sorting of data was possibly the first common point of contact between the original scientific uses of computers and their increasing involvement in commerce.

The considerable body of theory underlying the principles of sorting and the more esoteric procedures involved are left to the pursuit of the more interested student and this section will deal with sorting problems which may be encountered in common program applications and which may be worthwhile solving within a program rather than by having recourse to a manufacturer's standard sort utility.

9.10 BUBBLE SORT

The 'bubble sort' is the simplest method of sorting the elements in an array. It relies on a series of compare-and-exchange operations performed on successive passes through an array until the items are in the required sequence.

On each pass through the array, each pair of adjacent elements is compared and their contents exchanged if they are not already in the desired sequence. This is repeated for as many passes through the array as are required to reach a situation where a complete pass may be made without any exchanges being necessary. On each successive pass, data items need be compared no further than the point at which the last exchange occurred in the earlier pass. Figure 9.9 should illustrate the procedure.

Element Number:	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*
Initial Contents	14	6	23	18	7	47	2	83	16	38
After 1st pass	6	14	18	7	23	2	47	16	38	83
After 2nd pass	6	14	7	18	2	23	16	38	47	83
After 3rd pass	6	7	14	2	18	16	23	38	47	83
After 4th pass	6	7	2	14	16	18	23	38	47	83
After 5th pass	6	2	7	14	16	18	23	38	47	83
Final pass	2	6	7	14	16	18	23	38	47	83

Figure 9.9

An algorithm for this procedure is as follows:

Bubble Sort

```
TARGET←Number of items in the array to be sorted
SORT_DONE←False
Perform until SORT_DONE = True
    SWAP←False
    LEFT←1
    RIGHT←2
    Perform until RIGHT > TARGET
        If item (RIGHT) < item (LEFT)
            Swap item (RIGHT)⇌item (LEFT)
            SWAP←True
            LAST_SWAP←LEFT
        LEFT←LEFT + 1
        RIGHT←RIGHT + 1
    If SWAP = False
        SORT_DONE←True
    else
        TARGET←LAST_SWAP
```

The number of compare-and-swap operations required to sort the contents of an array sequence obviously depends on the initial sequence of the items before sorting.

The number of comparisons required to sort items into sequence via a bubble sort is in the order of

$\frac{1}{2}N^2$ where N is the number of items to sort.

This means that as N gets larger, the sorting procedure becomes exponentially less efficient. Because of the simplicity of its algorithm, however, a bubble sort remains the easiest method of sorting small volumes of data.

9.11 SHELL SORT

This sorting algorithm is named after its proposer, Donald Shell, and is a variation on the compare-and-swap principle of the bubble sort. Essentially it uses the same principle except that the span for comparing items starts at half the size of the array to be sorted and is halved each time a pass is made through the array without the need to exchange any items. Finally, the span is reduced to a value of 1 and hence the final passes of the sort are identical with a bubble sort.

An algorithm for this procedure is as follows:

```
Shell sort
    ARRAY_SIZE←Number of items in the array to be sorted
    SPAN←ARRAY_SIZE/2
    Perform until SPAN = 0
        COUNTER←1
        TARGET←ARRAY_SIZE − SPAN
        Perform until COUNTER > TARGET
            LEFT←COUNTER
            PASS_COMPLETED←False
            Perform until PASS_COMPLETED = True
                RIGHT←LEFT + SPAN
                If item (LEFT) < item (RIGHT)
                    PASS_COMPLETED = True
                else
                    Swap item (RIGHT)⇌item (LEFT)
                    If LEFT > SPAN
                        LEFT←LEFT − SPAN
                    else
                        PASS_COMPLETED←True
            COUNTER←COUNTER + 1
        SPAN←SPAN/2
```

Note: When calculating each value of SPAN, the division by 2 must produce an integer result, that is, 51/2 = 25, 1/2 = 0, etc.

9.12 CONCLUSION

The construction, maintenance and manipulation of arrays is an essential skill for every programmer. It is one of the responsibilities of a programmer to be able to construct an appropriate storage structure for the data which he has to process. A familiarity with a variety of types of arrays and the common processes associated with them should be regarded as a fundamental programming tool.

10 ABSTRACT DATA STRUCTURES

CHAPTER 10

10.1 ABSTRACT DATA STRUCTURES

Data Abstraction

With the proliferation of more complex data structures there has been an increased use of data abstraction. The concept of data abstraction is that of defining a data structure and a series of possible operations on that structure and making the implementation of the process 'transparent' to the programmer. For example, we may define a stack as a series of similar elements which operate on a last-in-first-out principle whereby we may access only the most recently added item. We then define a 'push' as an operation which adds an element to the stack and a 'pop' as one which retrieves an element. It is then immaterial to the programmer whether the stack is maintained by a pointer moving along a linear array or by a list structure. Data base management packages are probably the most highly developed examples of data abstraction because a program may request the 'next' or 'prior' items in a set or the 'owner' of that set and be freed from the complexities (and made oblivious of the overhead!) of the processing required to retrieve the requested item.

The concept of data abstraction interlocks with that of structured programming which also deals with levels of abstraction. For example, we write a module of code which returns an item sought in a table. To the calling routine, the method of performing the search is immaterial. The search routine may then address itself more particularly to the data structure of the table and may be successively refined for efficiency without affecting the remainder of the program.

In looking at abstract data structures, we will define the abstraction which typifies the data structure concerned, present a method of physically representing the structure and provide algorithms for common operations on the structures.

10.2 QUEUES

A queue is a data structure capable of holding a series of items to be processed on a first-in-first-out basis. It needs to be accessible at both its head (for removal of items) and its tail (for addition of items).

The operations usually required for queues are:

- *Initialize:* set the initial conditions before the start of the queue's operation.
- *Insert:* add a new item to the queue (provided that the queue is not full).
- *Remove:* delete an item from the queue (provided that the queue is not empty).
- *Count:* count the number of items currently in the queue.
- *Space:* count the number of vacancies currently in the queue.

The queues with which we are familiar in the real world usually operate by moving each of their items one place forward whenever one is removed from the head of the queue. When representing queues inside a computer, rather than adjusting the position of items in a queue by physically moving them from element to element, a better approach is to maintain two pointers, one indicating the head of the queue and the other its tail. The pointers must therefore be capable of operating on the table on a wrap-around basis. A counter may be used to indicate the number of items currently in the table (see Figure 10.1).

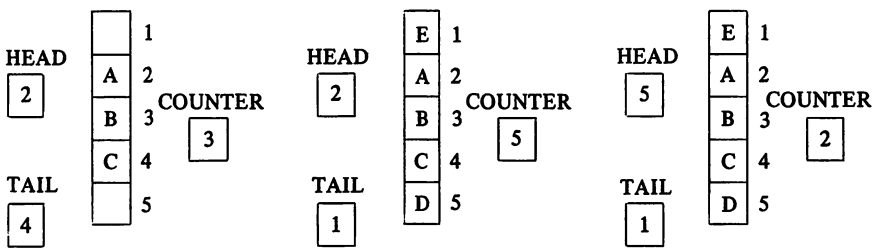
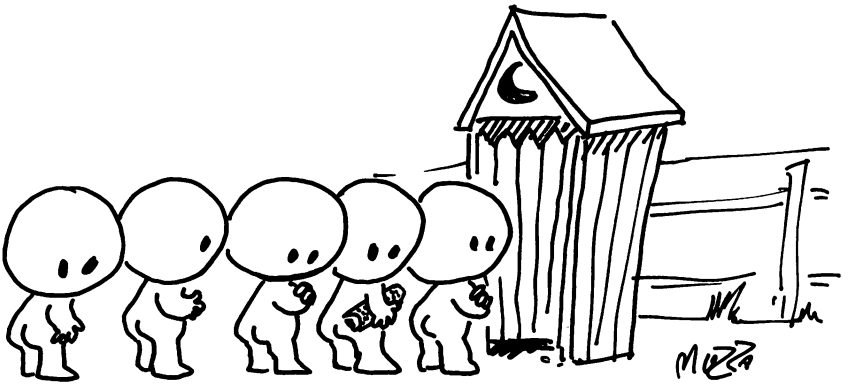


Figure 10.1

Note that it should not be necessary to physically delete items which have been logically removed. They will be over-written with new items added later as the queue operates.

The easiest method of physically representing a queue in a computer is with a linear array (although a linked list may also be used).



Algorithms:

A *Initialize*

Parameters: none
Occupancy Counter ←0
Head Pointer ←1
Tail Pointer ←Size of Queue (i.e. the number of elements in the array)

B *Insert*

Parameters: (a) New Item, a data item containing the element to be added to the queue
(b) Operation Successful, a status variable indicating the success or otherwise of the operation

If Occupancy Counter = Size of Queue
  Operation Successful←False
else
  Occupancy Counter←Occupancy Counter + 1
  Tail Pointer←Tail Pointer + 1
  If Tail Pointer > Size of Queue
    Tail Pointer←1
  Queue Item (Tail Pointer)←New Item
  Operation Successful←True

C *Remove*

Parameters: (a) Item Removed, a data item which will receive the item retrieved from the queue
(b) Operation Successful, a status variable indicating the success or otherwise of the operation

```
If Occupancy Counter = 0
    Operation Successful←False
else
    Item Removed←Queue Item (Head Pointer)
    Occupancy Counter←Occupancy Counter − 1
    Head Pointer←Head Pointer + 1
    If Head Pointer > Size of Queue
        Head Pointer←1
    Operation Successful←True
```

D *Count*

Parameter: Counter, a data item which will receive the count of the number of queue elements currently occupied

```
Counter←Occupancy Counter
```

E *Space*

Parameter: Counter, a data item which will receive the count of the number of queue elements currently vacant

```
Counter←Queue Size − Occupancy Counter
```

10.3 STACKS

A stack is a data structure capable of holding a series of items to be processed on a last-in-first-out basis. It needs to be accessible only at its top. New items are added at the top of the stack and items are retrieved also from the top.

The operations usually required for a stack are:

- *Initialize:* set the initial conditions prior to the operation of the stack.
- *Push:* add a new item to the top of the stack (provided that the stack is not full).
- *Pop:* delete an item from the top of the stack (provided that the stack is not empty).
- *Count:* count the number of items currently in the stack.
- *Space:* count the number of vacancies currently in the stack.

Figure 10.2 illustrates the use of a stack.

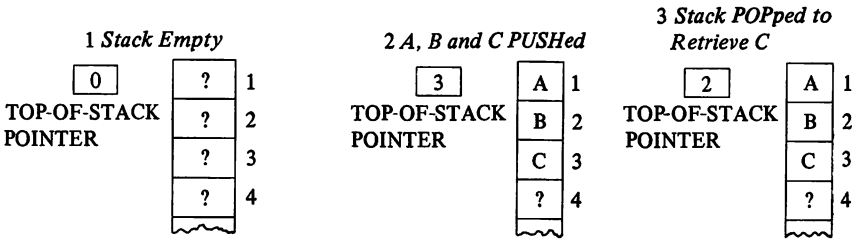


Figure 10.2

Note that retrieving items by POPping the stack does not physically remove them from the table; they will remain to be over-written by the next item to be PUSHed.

Once again, the easiest method of physically representing a stack in a computer is by means of a linear array (although, as for a queue, a linked list may be used).

Algorithms:

A *Initialize*

Parameters: none

Top-of-Stack Pointer←0

B *Push*

Parameters: (a) New Item, a data item containing the element to be added to the stack

(b) Operation Successful, a status variable indicating the success or otherwise of the operation

If Top-of-Stack Pointer = Size of Stack

Operation Successful←False

else

Top-of-Stack Pointer←Top-of-Stack Pointer + 1

Stack Item (Top-of-Stack Pointer)←New Item

Operation Successful←True

Note: Size of Stack is the number of elements in the array.

C *Pop*

Parameters: (a) Item Removed, a data item which will receive the element popped from the stack

(b) Operation Successful, a status variable indicating the success or otherwise of the operation

```
If Top-of-Stack Pointer = 0
    Operation Successful←False
else
    Item Removed←Stack Item (Top-of-Stack Pointer)
    Top-of-Stack Pointer←Top-of-Stack Pointer − 1
    Operation Successful←True
```

D *Count*

Parameters: Counter, a data item which will receive the count of the number of items currently in the stack

Counter←Top-of-Stack Pointer

E *Space*

Parameter: Counter, a data item which will receive the count of the number of stack items currently vacant

Counter←Size of Stack − Top-of-Stack Pointer

10.4 LINKED LISTS

A linked list is a data structure composed of a series of elements sometimes referred to as atoms and containing data together with a pointer to the next atom. A list will be associated with a list header which will contain a pointer to the first atom in the list. The last atom in a list may contain a null pointer to indicate the end of the table or may point back to the head of the list, in which case a circular structure is created which is often referred to as a chain or ring.

The advantage of such a structure is that elements may be inserted and deleted by manipulation of the pointers rather than by physical movement of the data.

A list of vacant atoms is maintained, called the free list, and additions to other lists are made by removing an atom (usually the first atom) from the free list and inserting it in or appending it to the other list. Similarly, atoms deleted from any list may be added back (usually at the front) to the free list either immediately upon release from their earlier list or by being flagged and picked up later by a 'garbage collection' operation. An example of these operations is shown in Figures 10.3 and 10.4.

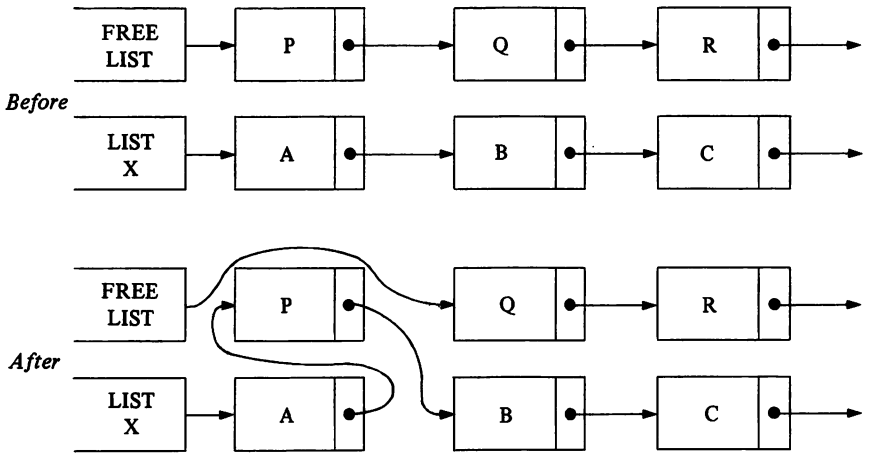


Figure 10.3 Inserting Elements in a List

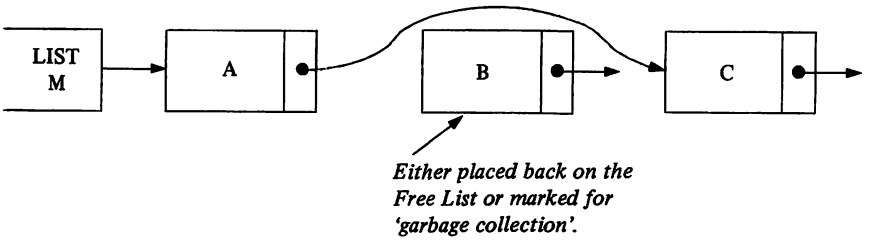


Figure 10.4 Deleting Elements from a List

A list structure may be used to maintain a series of items in sequence without physically re-arranging them as items are added to and deleted from the set. Figure 10.5 should illustrate the procedure.

A linked list may be physically represented in a computer by either a linear array of items each of which is capable of holding a data item and a pointer, or by parallel arrays of data items and pointers such that the pointer in the nth element of the pointer array relates to the data item in the nth element of the data array.

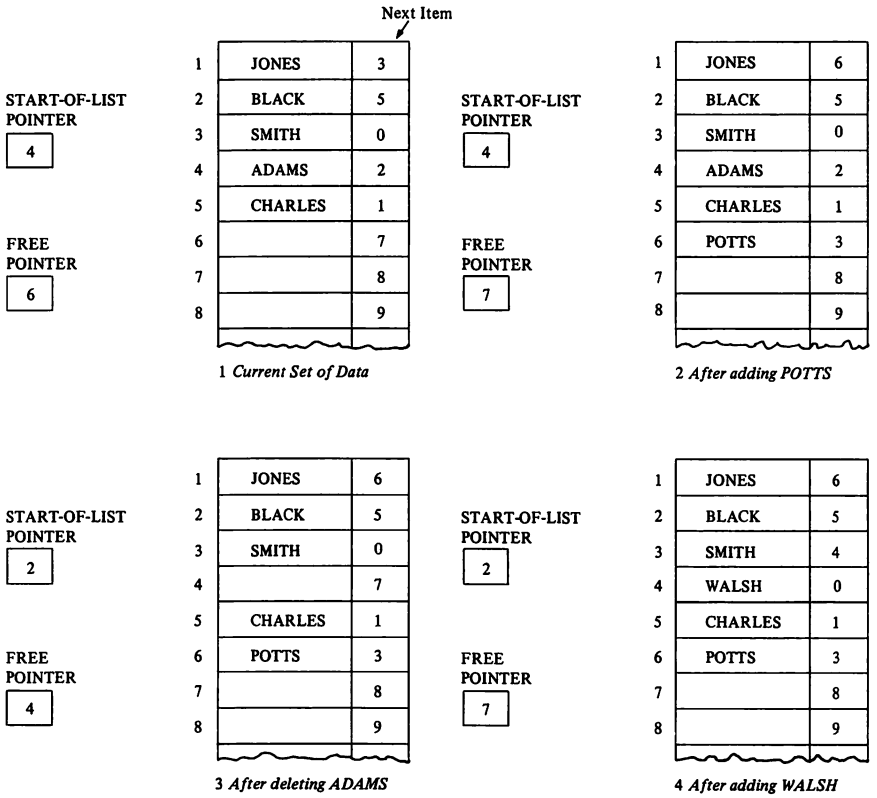
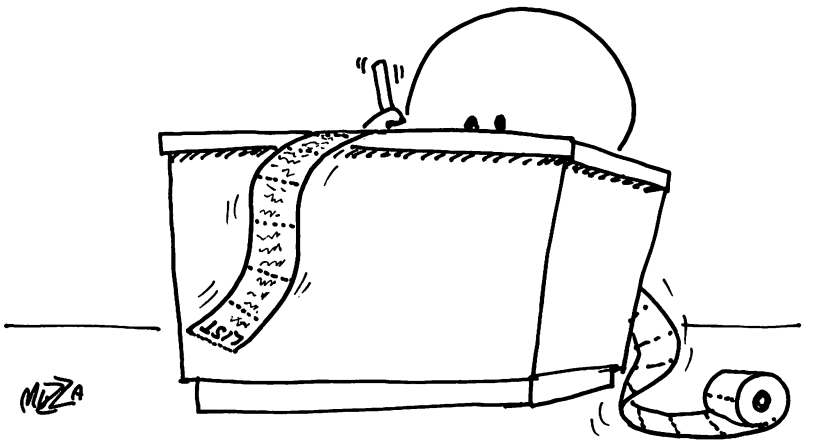


Figure 10.5

In a situation in which items are continually being appended to an existing list, it is advantageous to maintain not only a Start-of-List Pointer but also an End-of-List Pointer. This latter pointer enables the position of what is currently the last element in the structure to be determined without a search through all of the list elements.

Typical operations required on linked lists are:

- Initialize Free List: before any list operations take place, the array must be initially created as free space.
- Insert Item in Sequence: a data item is to be added to a list (which may have to be created for the purpose) in a logical sequence which maintains the list in ascending order.



Algorithms:

A *Initialize Free List*

Parameters: none
Free Pointer←1
Index←1
Perform until Index > Array Size
Pointer (Index)←Index + 1
Index←Index + 1
Pointer (Array Size)←Null

Note: This presumes that each list element contains a Data Item and a Pointer and that a 'null' value indicates the end of the list. In practice, this may be zero or a negative value.

B *Insert Item in Sequence*

Parameters: (a) New Item, a data item containing the element to be added to the list (if there is room)
(b) Operation Successful, a status variable indicating the success or otherwise of the operation

Do: Insertion
If Operation Successful = True
If Insert Pointer = 1
Start-of-List Pointer←1
Pointer (1)←Null
(This provides for the case where a new list has been created)
else
Do Link

Insertion

If Free Pointer = Null (i.e. no room left)
    Operation Successful←False
else
    Insert Pointer←Free Pointer
    Free Pointer←Pointer (Insert Pointer)
    Data Item (Insert Pointer)←New Item
    Operation Successful←True

Link

If Data Item (Insert Pointer) < Data Item (Start-of-List Pointer)
    Pointer (Insert Pointer)←Start-of-List Pointer
    Start-of-List Pointer←Insert Pointer
else
    Compare Pointer←Start-of-List Pointer
    Next Pointer←Pointer (Compare Pointer)
    Perform until Next Pointer = Null
        or Data Item (Insert Pointer) < Data Item (Next Pointer)
        Compare Pointer←Next Pointer
        Next Pointer←Pointer (Compare Pointer)
    Pointer (Insert Pointer)←Next Pointer
    Pointer (Compare Pointer)←Insert Pointer

More complex list structures may be created in which:

(a) each element contains both a pointer to the following element and one to the previous element. Such a list may be processed in either direction, as in Figure 10.6.

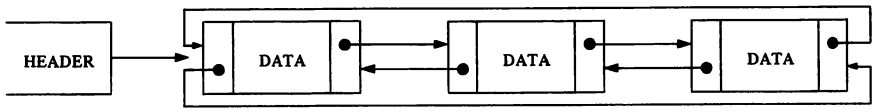


Figure 10.6 A Bi-directional List (Circular)

(b) each element contains three pointers, a next and a prior pointer and in addition a pointer to the list header. In such a list, if any element is encountered at random, immediate access is available to the start of the list without having to traverse each intervening element, as in Figure 10.7.

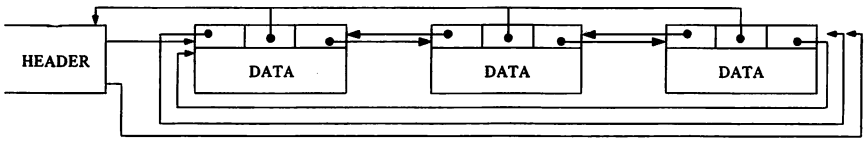


Figure 10.7 Pointers Maintained to Prior, Next, Header

10.5 TREES

A tree structure is one in which the elements maintain a sequence or hierarchy by a logical rather than a physical relationship. Each element in a tree will consist of:

(a) one or more items of data, and
(b) two or more pointers.

In general, such a structure is referred to as an *n*-ary tree where *n* refers to the number of pointers in each element. Elements in a tree are usually referred to as nodes, the first being the root node; where the number of pointers in each node is restricted to two, the tree is referred to as a binary tree.

Because the concept of a tree is an abstraction, relationships such as 'less than', 'greater than' or 'subordinate to' are maintained by pointers rather than by the physical placement of items relative to one another.

A Free Pointer will exist to indicate the address of the next vacant node (if any) available to receive data.

The accessing of records in a tree structure is referred to as the traversal of a tree and may take one of three forms:

End Order: Left branches – Right branches – Root
Post Order: Left branches – Root – Right branches
Pre-Order: Root – Left branches – Right branches

10.6 BINARY TREES

In a binary tree, the two pointers in each node may be used to maintain a sequence in the table by pointing to items which are respectively less than and greater than the node in which they are located. See, for example, Figure 10.8.

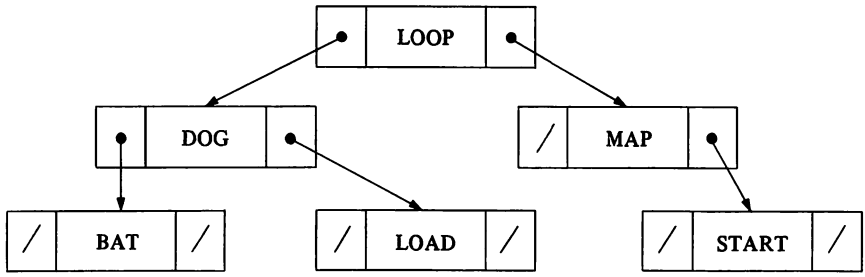


Figure 10.8

The advantages of such a table structure are:

(a) additions and deletions may be handled by adjustment of pointers and hence the logical sequence of the items is maintained without physical movement of the data; and

(b) depending on how well 'balanced' the tree is, the access time to any item approximates that of a binary search.

The physical representation of a tree structure may be implemented, as for a linked list, by means of a linear array in which each element contains a data item and two or more pointers or a group of parallel arrays for the data and pointer items. Physically, the table in the above diagram would appear as shown in Figure 10.9.

FREE POINTER: 7

		Left Pointer	*Right Pointer*
1	LOOP	2	4
2	DOG	3	5
3	BAT	/	/
4	MAP	/	6
5	LOAD	/	/
6	START	/	/
7			

Figure 10.9

The handling of duplicated items in such a table is effected by either:

(a) rejecting them, if this is appropriate in the circumstances, or
(b) storing them on either the less-than or greater-than path, the choice being immaterial provided that it is consistent.

A post-order traversal of a binary tree will retrieve the elements in their logical sequence, that is, the least first and greatest last.

Unless the tree structure was a 'threaded' tree (see section 10.7) a stack would be needed to enable the algorithm to 'remember' the order in which it must retrace its steps up the tree.

A typical such algorithm would be:

A *Post-Order Traversal of a Binary Tree*

```
Initialize stack
Pointer←position of root of tree (usually = 1)
Traversal Done←False
Perform until Traversal Done = True
  Perform until Pointer = Null
    Push Pointer on stack
    Pointer←Left Pointer (Pointer)
  If stack is empty
    Traversal Done←True
  else
    Pop stack into Pointer
    Process Data Item (Pointer), i.e. print it, etc.
    Pointer←Right Pointer (Pointer)
```

10.7 THREADED TREES

A threaded binary tree is one in which the pointers allow both downwards and upwards movement through the tree and hence remove the necessity for a stack to be used in its traversal. Downwards (or forwards) pointers will be positive values and upwards (or backwards) pointers will be negative ones.

In the following algorithms, two assumptions are implicit:

(a) the linked list of vacant elements is maintained via the right-hand pointer, and
(b) duplicated data items are placed to the 'left' of their equals.

Typical algorithms would be:

B *Insert Item in right-threaded Tree*

Parameters: (a) New Item, a data item containing the element to be inserted in the tree (if room permits)
(b) Operation Successful, a status variable indicating the success or otherwise of the operation

```
If Free Pointer = Null (i.e. no more room)
   Operation Successful←False

else
   Operation Successful←True
   Insertion Done←False
   Perform until Insertion Done = True
      New Pointer←Free Pointer
      Free Pointer←Right Pointer (New Pointer)
      Data Item (New Pointer)←New Item
      Right Pointer (New Pointer)←0
      Left Pointer (New Pointer)←0
      If New Pointer = 1
         Insertion Done←True

      else
         Compare Pointer←1
         Perform until Insertion Done = True
            Do Link


Link
   If Data Item (New Pointer) > Data Item (Compare Pointer)
      If Right Pointer (Compare Pointer) > 0
         Compare Pointer←Right Pointer (Compare Pointer)
      else
         Right Pointer (New Pointer)←Right Pointer (Compare
         Pointer)
         Right Pointer (Compare Pointer)←New Pointer
         Insertion Done←True
   else
      If Left Pointer (Compare Pointer) > 0
         Compare Pointer←Left Pointer (Compare Pointer)
      else
         Left Pointer (Compare Pointer)←New Pointer
         Right Pointer (New Pointer)←(0 − Compare Pointer)
         Insertion Done←True
```

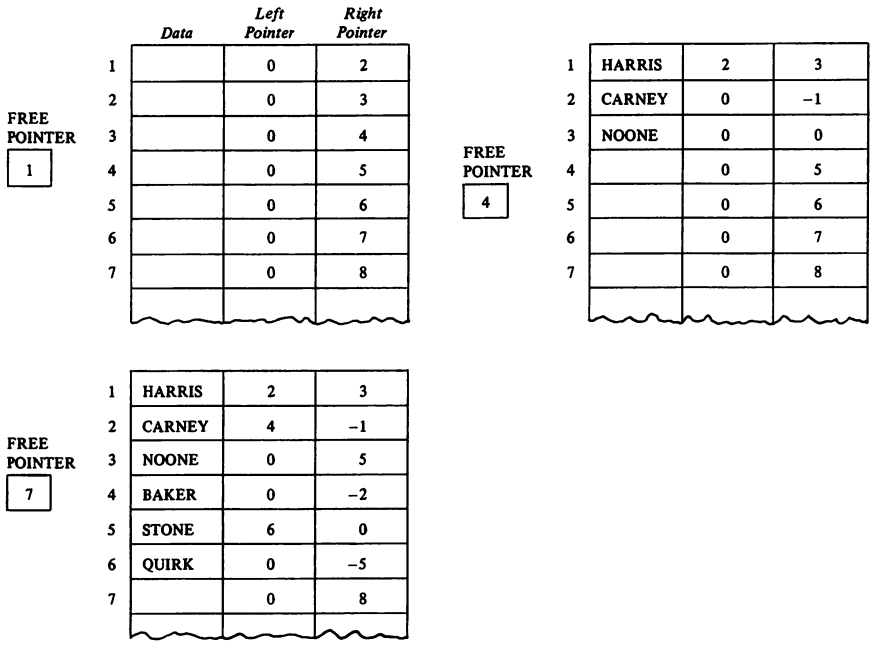


Figure 10.10 The Construction of a Right-Threaded Binary Tree

```
C  Retrieve Items in sequence from a right-threaded Tree
   Pointer←address of root node (usually = 1)
   Retrieval Done←False
   Perform until Retrieval Done = True
      Perform until Left Pointer (Pointer) = 0
         Pointer←Left Pointer (Pointer)
      Step Done←False
      Perform until Step Done = True
         Process Data Item (Pointer)
         If Right Pointer (Pointer) > 0
            Pointer←Right Pointer (Pointer)
            Step Done←True
         else
            If Right Pointer (Pointer) < 0
               Pointer←0 − Right Pointer (Pointer)
            else
               Retrieval Done←True
```

10.8 N-ARY TREES

Trees may have more than two pointers in each node and may be used to represent a hierarchical table. The number of pointers in each node may be fixed or variable. A table holding items of expenditure is illustrated in Figure 10.11.

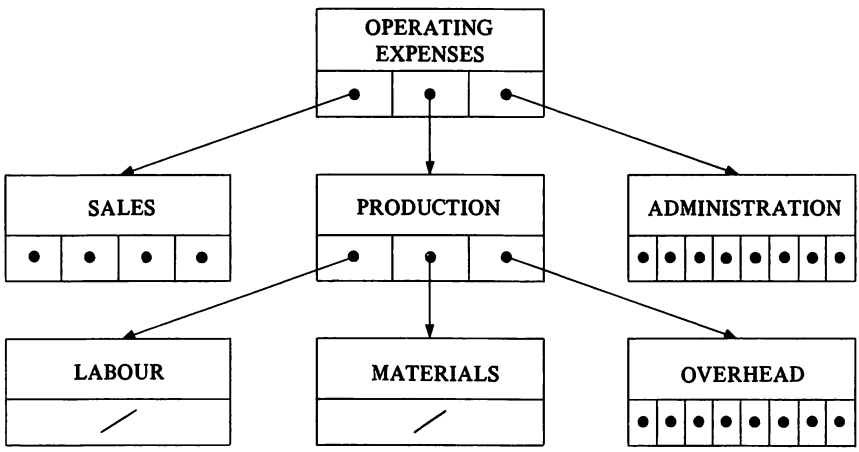


Figure 10.11

If a tree in which each node was able to hold several pointers was used to maintain a hierarchical data structure such as a chart of accounts or bill of materials (parts explosion) a pre-order traversal would enable its contents to be processed in their hierarchical groupings.

For example, such a traversal of the expenditure tree illustrated in Figure 10.11 would yield an output similar to:

Operating Expenses:
    Sales Expenses:
        Salesmen's Salaries
        Commission Paid
        Delivery Costs
    Production Expenses:
        Labour Cost
        Material Cost

Manufacturing Overhead:
Factory Rent
Light and Power
Depreciation on Machinery
Administration Expenses:
Office Salaries
Printing and Stationery
Depreciation on Fittings

An algorithm to effect this process would appear as:

D *Pre-order Traversal of an* n-*ary Tree*

```
Initialize Stack
Pointer←address of root node (usually = 1)
Traversal Done←False
Perform until Traversal Done = True
    If Pointer = Null
        Traversal Done←True
    else
        Push pointers in Element (Pointer) on the stack (from right to
            left)
        Process Data Item (Pointer)
        Pop stack into Pointer
```

It should be noted that any tree may be reduced to a binary format. The example in Figure 10.12 should illustrate the procedure.

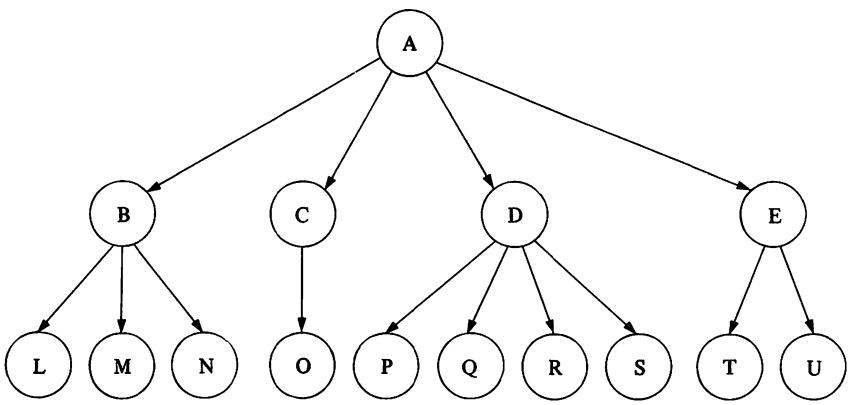


Figure 10.12 An N-ary Tree

These relationships consist of either:

1 "SONS" — B, C, D, E are sons of A
— P, Q, R, S are sons of D

OR

2 "BROTHERS" — B, C, D, E, are brothers
— L, M, N are brothers

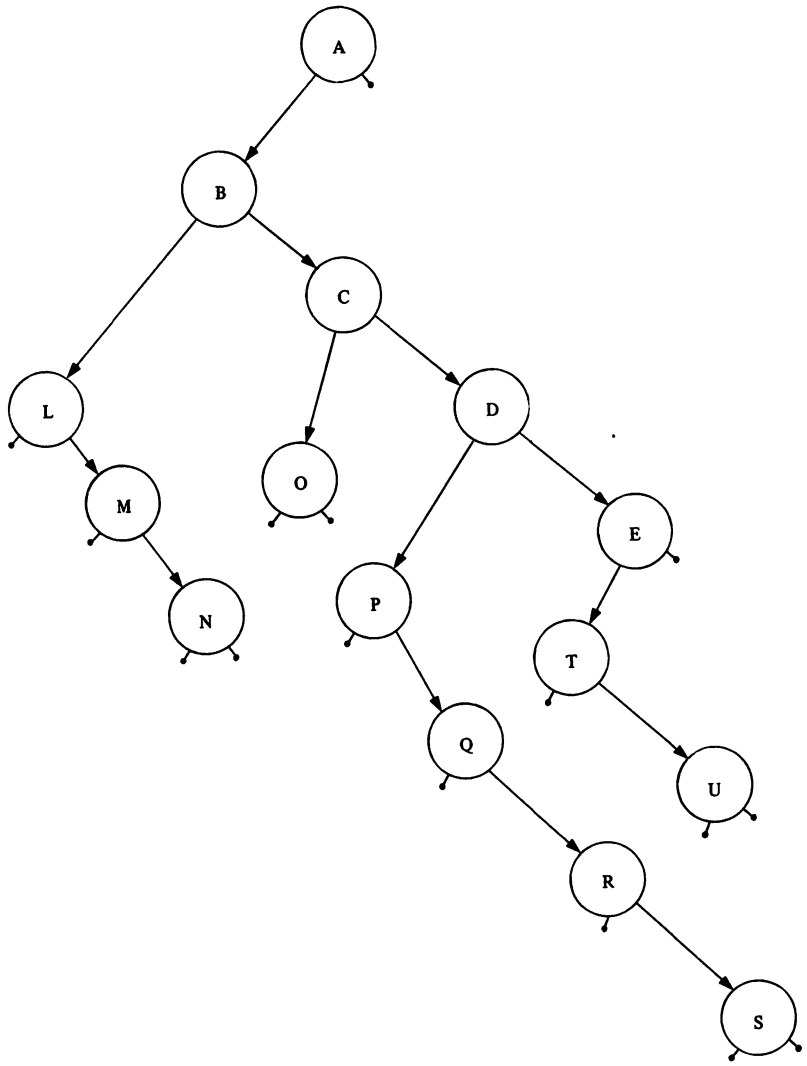


Figure 10.13 Re-structured Tree

Therefore, the tree may be re-structured with "SONS" to the left and "BROTHERS" to the right, that is, a binary format (see Figure 10.13).

10.9 TREE SORTING

A tree structure may be used as a sorting device. The methods of sorting discussed in Chapter 7 required the physical rearrangement of data items. An alternative is to place the data in a binary tree and then retrieve them in sequence. This procedure has the advantage that items may be added to or deleted from the 'sorted' data without requiring any re-sorting.

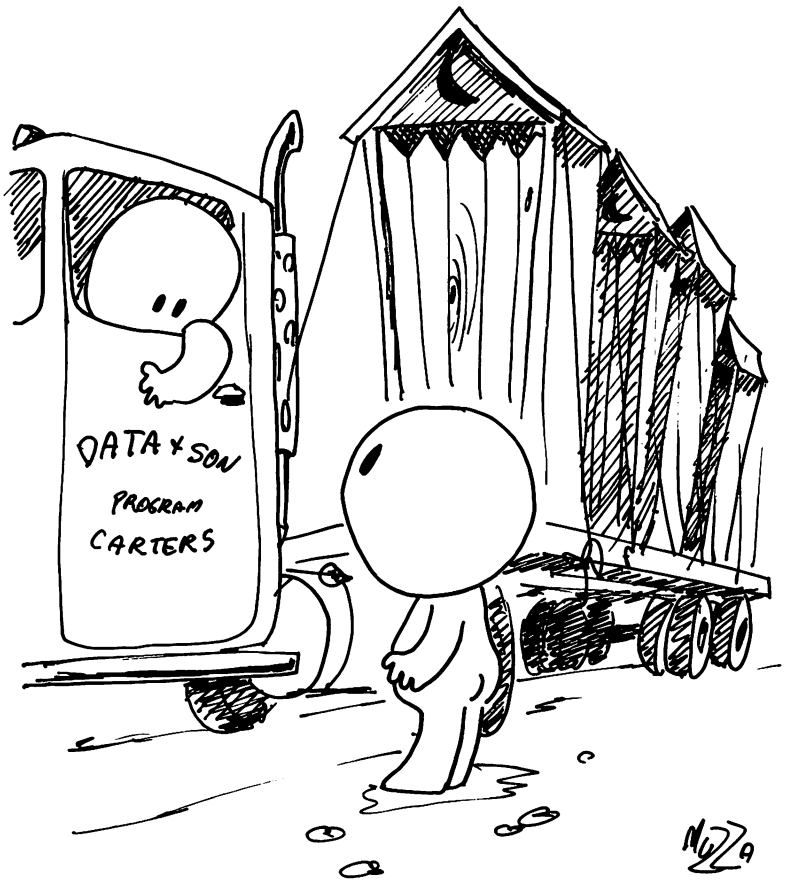
10.10 CONCLUSION

These definitions and algorithms illustrate two points:

(a) The abstract data structures are really linear arrays. The qualities which define them as stacks, lists, trees, and so on, lie in their behaviour rather than in their physical construction.

(b) A variety of data structures exists; it should be the aim of the programmer to be familiar with their construction and manipulation and to be able to use them where appropriate.

11 DATA-DRIVEN PROGRAMS



CHAPTER 11

11.1 DATA-DRIVEN PROGRAMS

Data-driven programming techniques are derived from the premises that:

(a) programs are constantly in need of amendment to make them cope with a changing environment;

(b) in general, it is easier to change data held in tables than it is to change program procedural code; and therefore

(c) programs which operate by following rules embodied in tables will be more amenable to modification than those which have their rules solely expressed in the instructions.

In this chapter we will look at some of the techniques for constructing programs, or parts of programs, by

(a) abstracting their governing rules and embodying these rules in tabular form, and

(b) driving the code by means of the data thus constructed.

11.2 DECISION TABLES

Decision tables are an often-used aid in systems analysis and design and specify rules of logic required to solve a problem. They can also be of assistance in specifying the operation of a program.

The general format of a decision table is shown in Figure 11.1 together with an example of such a table used to specify the rules relating to a simple pay roll.

	Rules			
Conditions				
Actions				

Figure 11.1 General Format of Decision Table

If all possible rules resulting from n conditions are specified, the table must provide for 2^n rules although some (as in the case of rules 5 and 6) may not be possible (see Figure 11.2).

	1	2	3	4	5	6	7	8
Hours worked > 40	Y	Y	Y	Y	N	N	N	N
Hours worked > 60	Y	Y	N	N	Y	Y	N	N
Junior employee	Y	N	Y	N	Y	N	Y	N
Reject as error	X	X			X	X		
Replace Hours by: 40 + 2 (Hours – 40)			X	X				
Replace Rate by: Rate x 0.875			X				X	
Compute Pay = Rate x Hours			X	X			X	X

Figure 11.2 A Table Showing All Rules

Rules 5 and 6 are logically impossible because they purport to relate to a situation in which the employee *has not* worked beyond 40 hours but *has* worked beyond 60 hours. Thus, rules 5 and 6 are eliminated as impossible:

	5	6	
Hours worked > 40	N	N	
Hours worked > 60	Y	Y	impossible!
Junior employee	Y	N	

It is possible to take rules which lead to the same action or group of actions and combine them in a single rule. Note that rules 1 and 2 lead to the common action 'Reject as error' and differ only in the result of the test for a Junior Employee. Clearly, the result of this test is immaterial because the action is taken whether or not the employee is a junior. Such an unnecessary test may be replaced by a 'dash' entry indicating that the test need not be made in evaluating that rule. Rules 1 and 2 are combined as follows:

	1	2		1
Hours worked > 40	Y	Y		Y
Hours worked > 60	Y	Y	becomes:	Y
Junior Employee	Y	N		–
Reject as Error	X	X		X

A simplified version of the table could be constructed by removing impossible conditions and introducing dash entries (see Figure 11.3).

	1	2	3	4	5
Hours worked > 40	Y	Y	Y	N	N
Hours worked > 60	Y	N	N	N	N
Junior Employee	–	Y	N	Y	N
Reject as error	X				
Replace Hours by:					
40 + 2 (Hours – 40)		X	X		
Replace Rate by: Rate x .0875		X		X	
Compute Pay = Rate x Hours		X	X	X	X

Figure 11.3 A Table with Dash Entries

Although the simplification of decision tables as shown in Figure 11.3 is of value in a documentation/analysis context, the tables are generally of more use as a programming aid if they are left in their fully expanded version. This has the additional benefit of ensuring that every possible combination of conditions has been considered and catered for.

If, in the case of the table incorporating all rules, each *Y* was replaced by a 1 and each *N* by a zero, the result would be as shown in Figure 11.4. Note that the actions have been simplified for the purpose of the example.

	1	2	3	4	5	6	7	8
Hours worked > 40	1	1	1	1	0	0	0	0
Hours worked > 60	1	1	0	0	1	1	0	0
Junior Employee	1	0	1	0	1	0	1	0
Action 1	X	X			X	X		
Action 2			X	X				
Action 3			X				X	
Action 4			X	X			X	X

Figure 11.4

By ascribing a value of 4 to a 1 in the first row (i.e. a 'Yes' answer to the question Hours Worked > 40?), a value of 2 to a 1 in the second row (i.e. 'Yes' to Hours Worked > 60?) and a value of 1 to 1 in the third row (i.e. 'Yes' to Employee is a junior?) the resulting table represents for each rule a number in the range 7 to zero.

Each number in the range can be obtained by satisfying only one pattern of results, that is, a 3 can only be obtained from a pattern of *N, Y, Y* as the result of the three tests, and a 6 can only result from answers of *Y, Y, N.*

The code to evaluate the three tests and drive the program logic accordingly can be implemented by a Case statement:

```
RESULT←zero
If Hours Worked > 40
   Add 4 to RESULT
If Hours Worked > 60
   Add 2 to RESULT
If Employee is a junior
   Add 1 to RESULT
Case of RESULT:
   7, 6, 3 or 2: Do Action 1
              5: Do Action 2
                 Do Action 3
                 Do Action 4
              4: Do Action 2
                 Do Action 4
              1: Do Action 3
                 Do Action 4
              0: Do Action 4
```

This technique is often referred to as 'rule matching'. It provides a neater solution in many cases to its alternative, the branching tree implemented by a series of nested **If**s, for example:

```
If Hours Worked > 40
   If Hours Worked > 60
      If Employee is a junior
         Do Action 1
      else
         Do Action 1
   else
      If Employee is a junior
         Do Action 2
```

```
        Do Action 3
        Do Action 4
    else
        .
        .
        .
        etc.
```

11.3 STATE TRANSITION TABLES

A program may be considered as an algorithm which progresses from its current state (i.e. what it is currently doing) to one of many other possible states, depending on changes in its operating environment. The arrival at any one particular state, or process, depends on the combination of the previous state and the occurrence of a particular event.

Program control is effected by means of

(a) a main driving module which performs as subroutines a series of processing modules;
(b) the processing modules, each of which addresses itself to one particular portion of the program's operation; and
(c) a state variable which is a data item the value of which is potentially changed by the occurrence of events and used by the driving routine to direct the path of the program to the next appropriate module.

The following example is that of a routine which has the task of examining a character string, as it may have been input from an operator at a terminal, checking it for correctness of format (or syntax) and separating it into its discrete components.

A stock part identifier in a motor accessories application has the general format:

A/B/C

where: A must be present and must be entirely numeric;
The 1st '/' is permitted only if B and/or C follow;
B may be omitted but if present it must be entirely numeric;
The 2nd '/' is permitted only if C follows;
C may be omitted but if present it must be either entirely numeric or entirely alphabetic (but may not contain blanks).

In addition to these rules, leading blanks (spaces) may occur before the start of *A*, and trailing blanks may occur after the last character, but blanks may not be embedded in the part identifier. A maximum of 30 characters is allowed on the screen for input of the entire identifier.

The following examples should illustrate the required operation of the program module in validating the format of the identifier and separating its components into their respective destinations:

Input text	*Valid?*	*A*	*B*	*Numeric C*	*Alpha C*
123/78/62	Yes	000123	000078	000062	blank
47/13	Yes	000047	000013	000000	blank
95//ABC	Yes	000095	000000	000000	ABC
46/XYZ	No				
46//145	Yes	000046	000000	000145	blank
/23	No				
57/13/AB96	No				
9481	Yes	009481	000000	000000	blank
376 42	No				

To embody the rules of syntax in a table, it is necessary to identify the states in which the algorithm may exist and the events which may result in a change from one state to another. The events in the example in Figure 11.5 will mainly consist of encountering various types of characters in the free-format input text string.

Algorithm States	*Current character in input string*					
	Digit	*Letter*	*Blank*	*'/'*	*30 Characters Processed*	*Else*
1. Leading blanks	2	9	1	9	9	9
2. Processing A	2	9	8	3	10	9
3. Just after 1st '/'	4	9	9	5	9	9
4. Processing B	4	9	8	5	10	9
5. Just after 2nd '/'	6	7	9	9	9	9
6. Processing Numeric C	6	9	8	9	10	9
7. Processing Alpha C	9	7	8	9	10	9
8. Trailing blanks	9	9	8	9	10	9
9. Error found, stop.	Next state to which algorithm moves					
10. Format correct						

Figure 11.5 State Transition Table for Stock Identifier Syntax

To implement these rules in an algorithm (and thence subsequently in a programming language) the following procedure is required:

(a) Build the state transition table as an array consisting of a row for each state other than the Error and Completed states and a column for each event required to be detected. The example in Figure 11.5 would produce the array shown in Figure 11.6.

Row	*Column* 1	2	3	4	5	6
1	2	9	1	9	9	9
2	2	9	8	3	10	9
3	4	9	9	5	9	9
4	4	9	8	5	10	9
5	6	7	9	9	9	9
6	6	9	8	9	10	9
7	9	7	8	9	10	9
8	9	9	8	9	10	9

Figure 11.6

It will be presumed that any component of the array in Figure 11.6 may be accessed by the name NEW_STATE and quoting appropriate values for ROW and COLUMN
e.g. NEW_STATE (ROW, COLUMN).
In the case where
ROW = 5 and COLUMN = 3
NEW_STATE (ROW, COLUMN) = 9

(b) Construct the algorithm of components which
(i) perform the initializing of variables and overall algorithm control;
(ii) select the appropriate processing module (in those cases where processing is actually required) depending on the Current State, and
(iii) detect the events which change the state.

The following algorithm implements the example above:

Validate Stock Identifier
Accept Stock Identifier into an array of 30 characters accessed by the name ID_CHARACTER using ID_INDEX as an index

```
A←0
B←0
NUMERIC_C←0
ALPHA_C←blanks
CURRENT_STATE←1
STOCK_INDEX←1
Perform until CURENT_STATE = 9 or 10
    Do Check Identifier Syntax
If CURRENT_STATE = 10
    .
    .
    .
    .
else
    .
    .
    .
    .
```

Check Identifier Syntax

```
Do Check Next Character
COLUMN←CHARACTER_TYPE
ROW←CURRENT_STATE
CURRENT_STATE←NEW_STATE (ROW, COLUMN)
Case of CURRENT_STATE:
2: A←10*A + ID_CHARACTER (ID_INDEX)
4: B←10*B + ID_CHARACTER (ID_INDEX)
6: NUMERIC_C←10*NUMERIC_C + ID-CHARACTER
   (ID_INDEX)
7: Append ID_CHARACTER (ID_INDEX) to current string of
    characters in ALPHA_C
else: no action needed
```

Check Next Character (Select appropriate array column)

```
If ID_INDEX > 30
    CHARACTER_TYPE←5
else
    Case of  ID_CHARACTER (ID_INDEX)
    Digit:   CHARACTER_TYPE←1
    Letter:  CHARACTER_TYPE←2
    Blank:   CHARACTER_TYPE←3
    '/':     CHARACTER_TYPE←4
    else:    CHARACTER_TYPE←6
Add 1 to ID_INDEX
```

As well as using this simple algorithm, the programmer benefits from being able to implement many of the likely changes to the syntax rules for the Stock Identifier by changing some of the values in the state array without needing to change the processing algorithm. For example, consider the following changes to syntax rules:

(a) It is permissible for the whole item to be blank, that is, a null response from the operator is acceptable, but any identifier keyed in must conform to the prescribed rules.

(b) The two '/' characters are not invalid if they appear without being accompanied by a *B* or *C* identifier component, that is, 123/ or 123// is now to be valid.

(c) The component previously known as an alphabetic *C* may now contain any character at all providing that its first character is not a numeric digit.

The state table would appear as in Figure 11.7 after incorporating these above amendments.

	Column					
Row	1	2	3	4	5	6
1	2	9	1	9	10	9
2	2	9	8	3	10	9
3	4	9	9	5	10	9
4	4	9	8	5	10	9
5	6	7	7	7	10	7
6	6	9	8	9	10	9
7	7	7	7	7	10	7
8	9	9	8	9	10	9

Figure 11.7

When implementing such a technique in a program, care must be taken that sufficient documentation is provided to enable the reader to understand the purpose of the table. If this is not done, the algorithm will appear to be driven by 'magic numbers' which have no significance to anybody subsequently charged with the maintenance of the program.

11.4 RUN-TIME INTERPRETED TABLES

This technique involves the storing in tabular form of information concerning the operation of a program. It is essentially the same approach as that taken using state transition tables, except that it is extended to encompass more of the program's functions and more of the internal data items used by the program.

It is an extremely useful technique with which to specify a set of procedures which may be subject to frequent alteration at the request of a user or which may have to exist in several versions to cater for a variety of users each of whom may require a degree of 'customizing' for his or her own environment.

Examples of situations suitable for such treatment are:

- Screen procedures for data input from source documents, the format of which may vary from one user to another. It is advantageous to be able to prompt for operator response in the sequence of the fields on the input document and to insert or delete fields for one of a number of users of a common system.
- Computer assisted learning programs which function on a question-and-answer basis. At any time the program is required to present a varying set of questions, check the responses against a supplied list of answers, allow a varying number of retries at any question, and so on.
- Test file loaders which are parameter driven to prompt an operator to supply data for a variety of record types with a variety of fields some of which may be subject to one of a number of edit constraints, for example, it must be numeric, must be a valid DDMMYY date, and so on.

The information contained in the table will be of three types:

(a) reference to data items held in other tables,
(b) references to procedural routines to be executed within the program, and
(c) references to other items within the interpreted table itself.

References to data items held in other tables, for example, the text to be displayed as a prompt to a screen operator when requesting the input of a data item, will normally constitute the subscripts of those data items in their respective tables.

References to procedural routines to be executed in the program, such as a validation/edit procedure to be applied to a data field on its receipt from a screen operator, will be control variables associated with Case Statements or Computed Branches.

References to other items in the controlling table will also be subscripts, this time to elements in its own structure, for example, the questions following and preceding a question to a screen operator.

Table Maintenance

One method of maintaining interpreted tables in a program is by the conventional method of a programmer altering the contents of the control table or of any of its associated tables and recompiling the program each time the user requests a change. This procedure, however, does not either:

(a) enable the maintenance to be effected independent of program recompilation by the programmer or, preferably, by the user, or
(b) allow for the use of a program by a number of users each of whom has a different requirement from his or her colleagues; for example, an order entry procedure via a screen from order forms used by a number of branches of an organization, each of which may have a slightly different form design from the others.

A preferable maintenance technique is to remove the tables from the program and store them in one or more files. The contents of the tables may now be amended either by:

(a) the use of a standard text editor to vary the file/record contents, or
(b) the use of a specially written editor provided by the data processing staff to enable the user to effect the desired changes.

The second approach will involve the production of a special software tool, but the time spent in its implementation will be recouped many times over in the lifetime of the system.

Figures 11.8 and 11.9 illustrate the tables required for a typical procedure whereby an operator at a screen can be prompted for a series of responses, those responses validated where appropriate, an error message displayed if necessary, and the data thus captured written to an output file.

The purpose of each of the tables is as follows:

Control Table
This is the primary table which governs the operation of the driving algorithm in managing the dialogue between the program and the screen operator.

- Question Details Pointer
  The subscript of the entry in the Question Details Table which relates to the question being currently asked of the operator.
- Reply Edit Pointer
  The subscript of the entry in the Reply Details Table which enables the operator's response to be accepted and validated.
- Output Format Pointer
  The subscript of the entry in the Output Format Table which enables the operator's response, if valid, to be assembled in an output record to be written to a file holding the data input by the operator.
- Next Question if Reply OK
  The subscript of the entry in the Control Table which contains details of the next question to ask the operator if the response to the current question is valid. Note that this need not necessarily be the adjacent question in the table. Different users may be asked the questions in different sequences depending on their operating environment.
- Next Question if Reply Not OK
  The subscript of the entry in the Control Table which contains the next question to be asked of the operator if the current response is invalid. This may be the repetition of the current question or any other suitable prompt.
- Prior Question
  The subscript of the entry in the Control Table which contains details of the question asked before the current question. This may be needed to allow the operator to return to previous questions to amend their responses.

Question Details Table
This table contains detailed information of each question referred to in the Control Table.

- Prompt Text Pointer
  The subscript of the entry in the Prompt Text Table which contains the prompt to be displayed to the operator for any given question.

- Prompt Position
  The line and column on the screen at which the text is to be displayed.
- Reply Essential
  A Yes/No indication of whether the operator is obliged to input a response to a given question.,

Prompt Text Table
This table holds the text of the prompts to be displayed to the operator.

Reply Details Table
This table contains information concerning the operator's response to each question.

- Maximum Size of Reply
  The maximum number of characters to be accepted from the operator as a response to a given question.
- Position to Accept Reply
  The line and column on the screen at which the operator's response is to be accepted.
- Validation Procedure
  An indicator which, if present, may be used in a Case statement or computed branch to select an appropriate processing routine to validate the operator's response.
- Position of Error Message
  The line and column at which an error message is to be displayed if the operator's response to a question is invalid.
- Error Message Pointer
  The subscript of an element in the Error Message Text Table to be displayed as an appropriate message in the event of an invalid response.

Error Message Text Table
This table contains the wording of error messages to be displayed to the operator.

Output Format Table
This table contains details of the positioning of the operator's valid responses in a record constructed from the entire data input from one complete set of questions.

- Output Format
  This indicates whether the operator's response is to be right-hand or left-hand aligned in its receiving field in the record.
- Position in Output Record
  This indicates the length of the field in the output record and its starting character position.
- Special Processing Routine
  This may be used, if present, to select by a Case statement or computed branch a processing module which may perform one or more operations on the output field before its inclusion in the record.

Question Number	*Question Details Pointer*	*Reply Edit Pointer*	*Output Format Pointer*	*Next Question if Reply OK*	*Next Question if Reply Not OK*	*Prior Question*
1	1	1	1	2	1	0
2	2	2	2	3	2	1
3	3	3	3	4	3	2
.						
.						
.						
50						

CONTROL TABLE

	Prompt Text Pointer	*Prompt Position*		*Reply Essential*
		Line	*Column*	
1	1	6	5	Y
2	2	7	9	Y
3	3	9	7	N

QUESTION DETAILS TABLE

1	Accession No.
2	Title
3	Date acquired (DD/MM/YY)
4	Author

PROMPT TEXT TABLE

	Maximum size of reply	*Position to accept reply*		*Validation Procedure*	*Position of Error Message*		*Error Message Pointer*
		Line	*Column*		*Line*	*Column*	
1	12	6	20	5	6	50	1
2	100	7	20		8	1	3
3	8	9	20	2	9	30	2

REPLY DETAILS TABLE

Figure 11.8

	Output Format	Position in O/P Record		Special Processing Routine
		Start	Length	
1	R	1	12	
2	L	13	100	
3	R	113	6	7

OUTPUT FORMAT TABLE

1	Reply must be numeric
2	Date must be of DD/MM/YY format
3	Blank reply is not acceptable
4	Number must not exceed 6 digits

ERROR MESSAGE TEXT TABLE

Figure 11.9

11.5 CONCLUSION

A large proportion of programming effort is spent on program maintenance. Alterations to programs are continually necessary to meet changing user requirements. Working from the premise that it is usually easier to change data than to change procedural code, it is worthwhile to incorporate as much as possible of a program's logic in a data structure. The foregoing approaches should provide an insight into the various means by which this may be accomplished.

Appendix A COBOL

A1 ALGORITHM IMPLEMENTATION

Implementation of the program control structures of sequence, repetition and selection is possible through syntax which is extremely close to that of the pseudocode used to express the algorithm from which a program will be coded.

Sequence

As COBOL instructions are executed in the order in which they occur in the source program, no further technique is required to implement a series of sequential operations, for example:

```
ADD 1 TO TOTAL.
MOVE TOTAL TO ANSWER.
DISPLAY "RESULT IS" ANSWER.
        .
        .
        .
```

Repetition

The pseudocode construct used was **Perform until** condition statement(s).

Because the PERFORM ... UNTIL ... feature in COBOL allows only for the execution of a separate module, or subroutine, the statement(s) to be executed must be placed in a separate Section within the Procedure Division. It must be remembered that the PERFORM ... UNTIL ... construct implies a leading decision technique. This means that the condition controlling the loop is evaluated before executing the performed routine for the first time and thereafter before each subsequent execution. Hence, if the condition is satisfied before the loop operation starts, there will be no execution of the instructions within the loop.

There are two basic methods of controlling a loop in COBOL, illustrated by a routine designed to examine each element in a table and to count the number of such elements containing the value zero.

The algorithm expressed in pseudocode would be:

```
Total←zero
Counter←1
Perform until Counter>size of table
   If Number (Counter)=zero
      Add 1 to Total
```

There are two basic means of implementing this loop in COBOL:

(a) Internal control of the loop:

```
PERFORM COUNT-ZEROS.
    .
    .
    .
COUNT-ZEROS SECTION.
    SET COUNTER TO 1.
    MOVE ZERO TO TOTAL.
CZ1.
    IF NUMBER (COUNTER) = ZERO
        ADD 1 TO TOTAL.
    IF COUNTER < TABLE-SIZE
        SET COUNTER UP BY 1
        GO TO CZ1.
```

(b) External control of the loop:

```
MOVE ZERO TO TOTAL.
SET COUNTER TO 1.
PERFORM COUNT-ZEROS
    UNTIL COUNTER > TABLE-SIZE.
    .
    .
    .
COUNT-ZEROS SECTION.
    IF NUMBER (COUNTER) = ZERO
        ADD 1 TO TOTAL.
    SET COUNTER UP BY 1.
```

or, alternatively:

```
MOVE ZERO TO TOTAL.
PERFORM COUNT-ZEROS
    VARYING COUNTER FROM 1 BY 1
    UNTIL COUNTER > TABLE-SIZE.
    .
    .
    .
COUNT-ZEROS SECTION.
    IF NUMBER (COUNTER) = ZERO
        ADD 1 TO TOTAL.
```

Of the two methods, external loop control provides a cleaner program and is in keeping with the general approach of structured programming. Internal loop control has the added disadvantage that it needs a branch (GO TO) instruction to implement it, and the prolific use of these instructions, and the labels

(paragraph names) which they necessitate, is extremely harmful to program clarity.

Selection

Selection in COBOL is basically implemented by the IF ... [ELSE] ... construction, for example:

```
IF TOTAL = ZERO
   DISPLAY ERROR-MESSAGE.
IF HOURS-WORKED > WORK-LIMIT
   PERFORM ERROR-PROCEDURE
ELSE
   PERFORM COMPUTE-PAY.
```

Multiple tests may be implemented by a series of IFs and ELSEs, for example:

```
IF SKILL-CODE = CARPENTER-CODE
   MOVE CARPENTER-RATE TO PAY-RATE
ELSE IF SKILL-CODE = FITTER-CODE
   MOVE FITTER-RATE TO PAY-RATE
ELSE IF SKILL-CODE = CLERK-CODE
   MOVE CLERK-RATE TO PAY-RATE
ELSE IF SKILL-CODE = SUPERVISOR-CODE
   MOVE SUPERVISOR-RATE TO PAY-RATE
      .
      .
      .
```

The concept of a Case statement is sometimes able to be implemented by a GO TO ... DEPENDING ... provided that the conditions being tested constitute a consecutive series of values held in a data-name. If such is the case, a GO TO ... DEPENDING ... may be able to provide a reasonably concise solution despite the need for a multitude of GO TO instructions, for example:

```
MULTIPLE-SELECTION SECTION.
   GO TO CODE-1 CODE-2 CODE-3 CODE-4
      DEPENDING ON CODE.
   PERFORM CODE-ERROR-ROUTINE.
   GO TO EXIT-POINT.
```

```
CODE-1.
    PERFORM CODE-1-ROUTINE.
    GO TO EXIT-POINT.
CODE-2.
    PERFORM CODE-2-ROUTINE.
    GO TO EXIT-POINT.
CODE-3.
    PERFORM CODE-3-ROUTINE.
    GO TO EXIT-POINT.
CODE-4.
    PERFORM CODE-4-ROUTINE.
    GO TO EXIT-POINT.
EXIT-POINT.
    EXIT.
```

This coding effectively provides a 'PERFORM ... DEPENDING ...' type of construct but, if used, should be constrained to a self-contained Section rather than embedded in the midst of a larger routine.

Program Modules

The pseudocode also introduced the concept of a module or subroutine comprising a free-standing task to be accomplished as part of a larger job and executed via a **Do** command.

Such modules in COBOL should comprise a SECTION and should be executed by a PERFORM verb. Examples of this have been given in the above illustrations.

A2 PROGRAM STRUCTURE

Programs should be written in modules, each of which should be a Section in the Procedure Division. The first Section should contain the mainline, or driving, logic of the program. Program termination, via the STOP RUN instruction, should occur at only one point in the program, that point being at the end of the first Section in the Procedure Division.

Execution of any but the mainline Section should only be by means of a PERFORM, never by a GO TO or by 'falling through' the program from a prior Section.

Each section should contain or control a separate task within the program and hence there should be no need for the use of the PERFORM ... THRU ... variant.

An EXIT verb within a paragraph at the end of a Section should not be necessary unless that Section contains an 'escape' branch (via a GO TO). Such a procedure should be regarded as an exceptional rather than a customary procedure.

If GO TO instructions are used they should not in any circumstances branch outside their own Section.

Care should be taken to separate 'managing' from 'working' Sections and the descent of the hierarchy of Sections within a program should reflect a transition from manager to worker instructions.

This progression is illustrated below by skeletal coding to implement the payroll algorithm in Chapter 2.

```
PROCEDURE DIVISION.
MAINLINE SECTION.
    PERFORM PROGRAM-INITIALIZATION.
    PERFORM OBTAIN-CORRECT-WORK-DETAILS
        UNTIL DETAILS-OK = TRUE
          OR END-OF-DATA = TRUE.
    PERFORM PROCESS-PAYROLL
        UNTIL END-OF-DATA = TRUE.
    PERFORM PRINT-FACTORY-TOTALS.
    PERFORM PROGRAM-FINALIZATION.
    STOP RUN.

*
*
PROCESS-PAYROLL SECTION.
    PERFORM PAY-CALCULATION.
    PERFORM UPDATE-HISTORY-RECORD.
    PERFORM PRINT-PAY-DETAILS.
    PERFORM OBTAIN-CORRECT-WORK-DETAILS
        UNTIL DETAILS-OK = TRUE
          OR END-OF-DATA = TRUE.

.
.
.
.
PAY-CALCULATION SECTION.
    IF HOURS-WORKED NOT > NORMAL-PERIOD
        MOVE HOURS-WORKED TO PAYABLE-HOURS
```

```
ELSE
   COMPUTE PAYABLE-HOURS
                                    = NORMAL PERIOD
                                    + OVERTIME FACTOR
                                    *(HOURS-WORKED -
                                    NORMAL-PERIOD).
COMPUTE GROSS-PAY = PAYABLE HOURS*
HOURLY-RATE.

PERFORM TAX-FORMULA.
COMPUTE NET-PAY = GROSS-PAY - TAX-AMOUNT.

PERFORM COINAGE-ANALYSIS.
ADD GROSS-PAY TO FACTORY-TOTAL-GROSS.
ADD TAX-AMOUNT TO FACTORY-TOTAL-TAX.
ADD NET-PAY TO FACTORY-TOTAL-NET.
```

Note that in this example, MAINLINE and PROCESS-PAYROLL Sections are managerial by nature because they only contain directives to other modules to perform certain tasks. The PAY-CALCULATION Section is partly managerial (i.e. IFs and PERFORMs) and partly working because it accomplishes several arithmetic calculations. We could infer that the COINAGE-ANALYSIS Section would entirely consist of working instructions.

A3 MODULE DESIGN

It should be a major consideration in program design that each functional task, consisting of one or more Sections, should operate as independently as possible of other tasks in the program.

Although data in a COBOL program is global by nature, it is possible to allocate a specific data area to the task of communicating with a procedural module. The allocated area will contain items which will be data and/or status variable parameters. The following is an example of the implementation of the Check Product ID Number algorithm in Chapter 3.

```
01 CHECK-PRODUCT-ID-WS.
   03 PRODUCT-TO-CHECK      PIC 9(6).
   03 PRODUCT-NAME          PIC X(30).
   03 PRODUCT-FOUND         PIC 9.
      .
      .
      .
```

```
    MOVE INPUT-PRODUCT-ID TO PRODUCT-TO-CHECK.
    PERFORM CHECK-PRODUCT-ID.
    IF PRODUCT-FOUND = TRUE
            .
            .
            .
        ELSE
            .
            .
            .
    CHECK-PRODUCT-ID SECTION.
        SET TABLE-INDEX TO 1.
        MOVE FALSE TO PRODUCT-FOUND.
        MOVE SPACES TO PRODUCT-NAME.
        PERFORM CHECK-PRODUCT-LOOP
            UNTIL PRODUCT-FOUND = TRUE
              OR TABLE-INDEX > TABLE-SIZE.
    CHECK-PRODUCT-LOOP SECTION.
        IF PRODUCT-TO-CHECK = TABLE-PRODUCT-ID
        (TABLE-INDEX)
            MOVE TRUE TO PRODUCT-FOUND
            MOVE TABLE-PRODUCT-NAME (TABLE-INDEX) TO
            PRODUCT-NAME
        ELSE
            SET TABLE-INDEX UP BY 1.
```

An alternative solution for the table search routine is given below, together with an illustration of suitable comments which would aid internal documentation.

```
    CHECK-PRODUCT-ID SECTION.
*   ------------------------------------------------------------
*       TO USE THIS PROCEDURE:
*       PLACE PRODUCT ID TO BE CHECKED IN
*       "PRODUCT-TO-CHECK"
*       IF PRODUCT ID IS MATCHED AGAINST A
*       PRODUCT IN THE TABLE OF VALID CODES
*           "PRODUCT-FOUND" IS SET TO "TRUE"
*           "PRODUCT-NAME" WILL CONTAIN THE
*           NAME OF THE RELEVANT PRODUCT
*       ELSE
*           "PRODUCT-FOUND" IS SET TO "FALSE"
*           "PRODUCT-NAME" WILL CONTAIN SPACES
*
        SET TABLE-INDEX TO 1.
```

```
SEARCH PRODUCT-TABLE VARYING TABLE-INDEX
WHEN PRODUCT-TO-CHECK = TABLE-PRODUCT-ID
(TABLE-INDEX)
   MOVE TRUE TO PRODUCT-FOUND
   MOVE TABLE-PRODUCT-NAME (TABLE-INDEX) TO
   PRODUCT-NAME
AT END MOVE FALSE TO PRODUCT-FOUND
   MOVE SPACES TO PRODUCT-NAME.
```

This example makes use of TRUE and FALSE as values for status variables. These are user-defined items and should be used for setting and testing all such variables. Avoid the use of 88-level condition names for this purpose because they provide less clarity to the reader than the use of TRUE and FALSE. The following example should illustrate this point:

The use of an 88-level status item implementation typically appears as:

```
01 CODE-STATUS              PIC 9.
   88 CODE-ERROR-FOUND      VALUE 1.
   88 CODE-CORRECT          VALUE 0.
```

Consider the instruction:

```
IF CODE-A IS NOT = CODE-B
   MOVE 1 TO CODE-STATUS.
```

Without cross-reference to the Data Division 88-level entry it is not apparent to the reader whether the status value of 1 represents the finding of an error or the finding of a correct situation.

The testing of the status item offers no help,

```
IF CODE-ERROR-FOUND
   PERFORM CODE-ERROR-PROCEDURE.
```

There is no apparent connection between the placing of a value of 1 in CODE-STATUS and the testing of the condition CODE-ERROR-FOUND.

The preferred method of implementing all status items within a program should be by declaring global condition names and values, for example:

```
01 STATUS-VALUES
      03 TRUE               PIC 9 VALUE 1.
      03 FALSE              PIC 9 VALUE 9.
```

The chosen PIC clauses and VALUEs are immaterial provided that all subsequent status variables have the same PIC clause, for example:

```
01 CODE-ERROR-FOUND          PIC 9.
```

The setting and testing of such status variables is now self-explanatory to the reader of a program:

```
IF CODE-A IS NOT = CODE-B
   MOVE TRUE TO CODE-ERROR-FOUND.
```

and

```
IF CODE-ERROR-FOUND = TRUE
   PERFORM CODE-ERROR-PROCEDURE.
```

A4 PROGRAMMING STANDARDS

The following are suggested as minimum standards for the writing of COBOL programs:

Program Function

The Identification Division should contain comments or a REMARKS section explaining the function(s) of the program and its interaction with any other program(s) in a system.

Coding Format

DIVISIONs and major SECTIONs must commence on a new page via a '/' character in column 7.

Related groups of statements in the Data Division and Procedure Division should be separated from one another by * lines for clarity of presentation.

Commence successive ASSIGN, PIC and VALUE clauses at the same column, for example:

```
      SELECT    X-FILE     ASSIGN TO "ABC".
      SELECT    Y-FILE     ASSIGN TO "PQR".

01    AAA       PIC X      VALUE "A".
01    BBB       PIC 9      VALUE 6.
```

Indent elementary items comprising a group item by 4 character positions and advance their level numbers by at least 2, for example:

```
01  A
    03  B                   PIC 9.
    03  C.
        05  D               PIC X.
        05  E.
            07  F           PIC 9(2)
            07  G           PIC 9(5)V9(2).
    03  H                   PIC X(4).
```

Use bracket notation for multiple characters in PICTURE clauses, for example:

PIC 9(3) not PIC 999

When using long alphanumeric literals, start the literal on a new line, for example:

```
01  MESSAGE          PIC X(40)          VALUE
    "..... ETC.....".
```

A VALUE clause should be used to define a constant not to initialize a variable, for example:

```
01  PAGE-SIZE        PIC 9(2)       VALUE 60.
```

correctly defines an item of data which does not alter during the execution of the program, whereas

```
01  ERROR-COUNTER    PIC 9(2).
```

and later

```
MOVE ZERO TO ERROR-COUNTER.
```

correctly initializes a variable the content of which may change many times as the program runs.

In the interests of efficiency, Index Variables should be used (and the related SET instructions) as subscripts to arrays (tables). In addition, each array should be accompanied by a size variable to be used in testing/controlling operations in the procedural code, for example:

```
01  TABLE
    03  ITEM                PIC 9(4)
                            OCCURS 20 TIMES
                            INDEXED BY ITEM-INDEX.
01  TABLE-SIZE              PIC 9(2) VALUE 20
```

and later

```
PERFORM 40-CLEAR-TABLE      TABLE-SIZE TIMES.
```

or

```
IF ITEM-INDEX IS NOT > TABLE-SIZE
    <action(s)>.
```

Heavily used arithmetic items should be held in COMP format for efficiency.

Program Structure

Programs should be written in modules, each of which will be a SECTION, the first module being the mainline or driving logic for the program.

Execution of any but the mainline SECTION may only be by a PERFORM, never by a GO TO or by 'falling through' the program from a previous SECTION.

Do not use the PERFORM ... THRU ... variant.

Indent IF/ELSE sequences to ensure that their function is clear, placing the IF<condition> and ELSE on lines by themselves, for example:

```
IF A=B
   ADD P TO Q
   MOVE R TO S
   IF C=D
      ADD M TO N
   ELSE
      NEXT SENTENCE
ELSE
   ADD V TO W.
```

Note: An acceptable format for multiple IFs is:

```
IF<condition>
   action-1
   action-2
ELSE IF<condition>
      action-3
      action-4
ELSE IF<condition>
      action-5
      action-6
  .
  .
  .
ELSE IF<condition>
      action-99
      action-100.
```

In the Procedure Division, write only one instruction per line and never break a data-name across two lines.

Naming Conventions

DATA DIVISION:

(a) All data names must be meaningful and hyphenation must be used to facilitate readability, for example, GROSS-PAY.

(b) Abbreviation of names is not a virtue, for example,

use GROSS-WEEKLY-WAGE
not GR-WK-WGE

(c) Always use the same basic name for an item of data wherever it is found in the DATA DIVISION. Duplication of data-names may be overcome by the use of Qualifiers or avoided by the use of identifying prefixes or suffixes on the names, for example:

```
FD MAG-TAPE-FILE
   .
   .
   .
01 MT-CUSTOMER.
   03 MT-CUSTOMER-NUMBER
   03 MT-CUSTOMER-NAME
   03 MT-CUSTOMER-ADDRESS
FD DISK-FILE
   .
   .
01 DISK-CUSTOMER.
   03 DSK-CUSTOMER-NUMBER
   03 DSK-CUSTOMER-NAME
   03 DSK-CUSTOMER-ADDRESS
```

but *never* the use of several variants on a name for what is essentially the same data item but appearing in several records, for example:

```
CUSTOMER-NMBR
CST-NO
CUST-NUMBER
   etc.
```

(d) Avoid the use of generalized global data items. Where possible, each procedural module should have allocated to it its own local data items.

PROCEDURE DIVISION:

(a) SECTION names must be meaningful and, unless the SECTION's purpose is self-explanatory, each must start with sufficient comment lines to make its purpose clear to the reader, for example:

```
READ-CUSTOMER-RECORD        SECTION.
    .
    .
    .
CALCULATE-NET-PAY           SECTION.
    .
    .
    .
FIND-CODE                   SECTION.
* THE TABLE OF VALID CODES IS
* SEARCHED VIA A BINARY CHOP TO
* MATCH THE CODE IN THE EMPLOYEE
* RECORD
```

(b) An EXIT paragraph for a SECTION should not be syntactically necessary unless that SECTION contains an 'escape' branch. Such paragraphs may, however, act as a documentation aid and their use is left to the individual programmer.

(c) A program should contain only one STOP RUN instruction and that should be in the mainline SECTION.

(d) The use of GO TO instructions and Paragraph names should be confined to the handling of exceptional circumstances and not considered to be part of normal control structures.
If GO TO instructions are used, under no circumstances may they reference paragraphs outside their own SECTION.

Further Points of Programming Style

Where possible, use 88-level condition names to avoid multiple testing by IFs, for example:

```
03 RECORD-TYPE              PIC 9.
   88 VALID-TYPE VALUES     1, 3, 4, 6
```

A program should contain only one OPEN, CLOSE, READ or WRITE for each file. These will normally be in a SECTION which may be PERFORMED whenever required.

Literals should not be used in the PROCEDURE DIVISION. Literal values should be given symbolic names and placed in the DATA DIVISION, for example:

```
MOVE CASUAL-EMPLOYEE-TYPE TO RECORD-TYPE.
```

rather than

```
MOVE 12 TO RECORD-TYPE.
```

In particular, subscripts should *never* be tested against literal values, for example:

```
use IF SUBSCRIPT < TABLE-SIZE
     ADD 1 TO SUBSCRIPT
          etc.
never
     IF SUBSCRIPT < 25
          etc.
```

Avoid complexity in compound and nested IF statements. In particular, avoid the use of compound tests linked by NOT; for example, many programmers have trouble in understanding why

```
IF CODE IS NOT = 1 OR NOT = 2
     PERFORM REJECTION-ROUTINE
```

rejects every code, including 1 and 2!

As a rule of thumb, if a compound/nested IF... ELSE... sequence is sufficiently complex that you feel the need to have somebody else check your logic, then it's too obscure! Break it up into smaller components and, if necessary, push some of the testing back into a subroutine.

Appendix B BASIC

Note A problem which arises in illustrating programs in BASIC is the variety of dialects of BASIC which are available. What has been chosen here is a very 'basic' BASIC which has none of the features which made the various forms of extended BASIC a much more elegant language. The reader should compare these features with those in the BASIC he or she is using and try to improve on these examples.

B1 ALGORITHM IMPLEMENTATION

BASIC by nature has few facilities to aid elegant program construction. Whatever structure is incorporated in the program must be the result of a discipline of coding style brought to bear by the programmer.

Sequence

As BASIC instructions are executed in the order in which they occur in the source program, no further technique is required to implement a series of sequential operations, for example:

```
100 READ A
110 LET B =2*A
120 LET C=3*A
130 PRINT A, B, C
```

Repetition

The pseudocode construct used was **Perform until.**

If a loop is to be executed a pre-determined number of times, a FOR ... NEXT ... construct may be used, for example: to set to zero each element in an array of N elements:

```
100 FOR K=1 TO N
110 LET E(K)=0
120 NEXT K
```

To implement a loop of an unknown number of interactions, a branch (GO TO) instruction is needed, for example, to read and print a series of numbers until a value of zero was encountered:

```
100 READ N
110 IF N=0 THEN 140
```

```
120 PRINT N
130 GOTO 100
140 ....
```

Note that in such a case the concept of a priming read may provide a neater solution, for example:

```
100 READ N
110 IF N=0 THEN 150
120 PRINT N
130 READ N
140 GOTO 110
150 ...
```

The latter solution is in closer conformity with the concepts of structured programming.

For the sake of program clarity, program loops of either of the above types should not span more than, say, 6 to 8 lines. If the task requires more than that amount of code, it should be committed to a subroutine, for example:

```
100 FOR K=1 TO N
110 LET P=E(K)
120 GOSUB 500
130 NEXT K
    .
    .
    .
500 REM   THIS SUBROUTINE PERFORMS A NUMBER
510 REM   OF CALCULATIONS ON "P" AN
520 REM   ELEMENT EXTRACTED FROM AN ARRAY
    .
    .
    .
690 RETURN
```

Selection

Selection is implemented by the IF command, but the construction in most BASICs allows for only a single instruction to follow the evaluation of the IF condition and allows for no ELSE construction.

For all but the most simple of situations, the program relies on a branching technique to effect the selection procedure, and it is essential to ensure that, to conform to the principles of structured programming, both the 'true' and 'false' branches of the IF resume processing at a common point, for example:

```
100 IF A=B THEN 160
110 REM   WHEN A<>B:-
120       LET P=Q+R
130       LET W=T*V
140       LET C=D/E
150       GOTO 200
160 REM   WHEN A=B:-
170       LET P=Q-R
180       LET W=T/V
190       LET C=D*E
200 PRINT P, W, C
.
.
.
```

Note the use of indentation to highlight the two sets of actions emanating from the IF and the REM statements to provide additional documentation for the reader.

If there are more than, say, six to eight instructions consequent on either branch of an IF, they should be placed in a subroutine and executed by a GOSUB instruction, for example:

```
100 IF A=B THEN 140
110 REM   WHEN A<>B:-
120       GOSUB 400
130       GOTO 160
140 REM   WHEN A=B:-
150       GOSUB 500
160 READ A, B, C
.
.
.
```

Multiple selection may be implemented by an ON instruction where the range of values being tested is consecutive. Once again, the various branches should all terminate at a common point before proceeding with the rest of the program, for example:

```
100 ON A GOTO 110, 150, 190
110 REM   A=1
120       LET ...
130       LET ...
140       GOTO 220
150 REM   A=2
160       LET ...
170       LET ...
```

```
180          GOTO 220
190 REM      A=3
200          LET ...
210          LET ...
220 PRINT A, B, C
      .
      .
      .
```

The use of REMs and indentation will again provide clarity for the reader.

Program Modules

The concept of a free-standing module or subroutine introduced by the pseudocode **Do** command should be implemented by a GOSUB. The module of code executed should always terminate by a RETURN, and under no circumstances should a routine executed in this manner do anything else other than, on completion, return to the instruction following the initiating GOSUB, for example:

```
100 IF A=B THEN 120
110    GOSUB 800
120
.
.
.
800 REM  THIS ROUTINE IS EXECUTED WHEN A<>B
810
    .
    .
    .
    .
    .
890 RETURN
```

B2 PROGRAM STRUCTURE

Programs should be written in logical sections, the first containing the mainline or driving logic of the program and the others each performing a self-contained task.

Execution of any but the mainline section should be by means of a GOSUB rather than by GOTOs or by 'falling through' the program from a previous section.

As a language, BASIC does not give much help to the programmer in maintaining a visible logical structure in a program. The success of the technique depends entirely on the discipline imposed by the programmer on his coding style.

Because sections of a program have no means of being assigned a meaningful procedure name, much more importance is thrust on internal comments statements than is the case in many other languages.

To further avoid confusion, care must be taken to separate 'managing' from 'working' modules, and there should be a hierarchy of such segments in the program.

This procedure is illustrated below by skeletal coding to implement portions of the Payroll algorithm in Chapter 2:

```
100 REM  OBTAIN CORRECT WORK DETAILS
110 GOSUB 300
120 REM  CHECK FOR END OF JOB
130 IF S1 = 1 THEN 250

140 REM  PAY PROCEDURE LOOP
150      REM  PAY CALCULATION
160      GOSUB ...
170      REM  UPDATE HISTORY RECORD
180      GOSUB ...
190      REM  PRINT PAY DETAILS
200      GOSUB ...
210      REM  OBTAIN CORRECT WORK DETAILS
220      GOSUB 300
230      REM  REPEAT FOR NEXT EMPLOYEE
240      GOTO 130
250 REM  PRINT FACTORY TOTALS
260 GOSUB ...
270 STOP
280 REM  ******************************************************
290 REM
300 REM  OBTAIN CORRECT WORK DETAILS :-
310 REM  ACCEPT EMPLOYEE NUMBER FROM OPERATOR
320 GOSUB 400
330 REM  CHECK FOR END OF JOB
340 IF S1 = 1 THEN 370
350 REM  ACCEPT HOURS WORKED FROM OPERATOR
360 GOSUB
370 RETURN
380 REM  ******************************************************
```

```
400 REM  ACCEPT EMPLOYEE NUMBER AND VERIFY
410 REM  BY RETRIEVING EMPLOYEE RECORD
420 REM  USING EMPLOYEE NUMBER AS KEY
430 REM  A ZERO INPUT AS EMPLOYEE
440 REM  NUMBER SIGNALS END OF JOB
450 PRINT "EMPLOYEE NUMBER PLEASE"
460 INPUT E
470 IF E<>0 THEN 500
480 LET S1=1
490 GOTO 590
500
  .
  .
  .
  .
  .
  .
  .
  .
590 RETURN
```

B3 MODULE DESIGN

It should be a major consideration in program design that each functional task should operate as independently as possible of other tasks in the program.

Although data in a BASIC program is global by nature, it is possible to allocate specific variables to the task of communicating with a procedural module. These designated variables will contain data or status parameters or both. The method by which the procedural module accomplishes its task is then immaterial to the remainder of the program.

This concept is amply illustrated by the intrinsic functions available in BASIC, for example:

```
LET X=SQR(Y)
```

This acknowledges the existence of a routine named SQR the task of which is to evaluate the square root of any non-negative variable passed to it. The user of the procedure may have no knowledge of the means by which the evaluation is effected. This independence of function should be sought in all program construction. Sufficient documentation must be included in the program to explain to a reader the means by which each program module may be used. The example below illustrates a procedure for calculating the area of a triangle from the lengths of its three sides. The algorithm is given in Chapter 1, Example 1.1:

```
500 REM  CALCULATE AREA OF TRIANGLE
510 REM  VARIABLES USED: S, S1, S2, S3, A, F
520 REM  TO USE THIS ROUTINE PLACE THE
530 REM  SIDE LENGTHS OF ANY TRIANGLE IN
540 REM  S1, S2 AND S3
550 REM  IF THE DIMENSIONS ARE IMPOSSIBLE
560 REM      F WILL BE SET TO ZERO
570 REM      A, THE AREA, WILL BE SET TO ZERO
580 REM  ELSE
590 REM      F WILL BE SET TO 1
600 REM      A WILL CONTAIN THE AREA
610 REM
620 LET S=(S1+S2+S3)/2
630 IF (S-S1)<0 THEN 700
640 IF (S-S2)<0 THEN 700
650 IF (S-S3)<0 THEN 700
660 LET F=1
670 LET S=S*(S-S1)*(S-S2)*(S-S3)
680 LET A=SQR(S)
690 GOTO 720
700 LET F=0
710 LET A=0
720 RETURN
```

A typical use of this routine would appear as:

```
100 LET S1=X
110 LET S2=Y
120 LET S3=Z
130 GOSUB 500
140 IF F=0 THEN 170
150      PRINT "AREA=", A
160      GOTO 180
170 PRINT "IMPOSSIBLE TRIANGLE"
180  .
     .
     .
```

B4 PROGRAMMING STANDARDS

The following are suggested as minimum coding standards for BASIC programs:

Program Function

Comment statements at the beginning of the program should clearly state the function of the program and its interaction with any other program in a system.

Dictionary of Variables

Comment statements should list all variables used in the program and explain what each represents. This should be done at the start of the program for global variables and at the start of each subroutine for local variables.

Coding Format

Where permitted by the BASIC system, use statement indentation to delineate sections of code which form a logical grouping. Illustrations of this are given earlier in this appendix.

Write one instruction per line.

Use comments statements liberally throughout the code as BASIC has little inherent self-documentation.

Program Structure

Construct the program using small task-oriented self-contained subroutines with local data items.

Execute the functional subroutines by means of an hierarchical structure of controlling modules, overall control being vested in a mainline module.

Keep all program modules small: no more than 20–30 executable statements.

Overall control should be by GOSUB calls to modules rather than GOTOs. If GOTOs are used they should never branch beyond their own module.

Each program module should have one entry point and one RETURN point.

It is important to remember that BASIC has no in-built facilities for handling structured programming. The elegance of a program depends entirely on the discipline brought to bear by the programmer.

Appendix C PASCAL

C1 ALGORITHM IMPLEMENTATION

As Pascal is a relatively modern programming language, it has been designed from the outset to possess features which facilitate the principles of structured program design.

The program control structures of sequence, repetition and selection are able to be implemented by syntax identical to the pseudocode used to formulate an algorithm.

Sequence

Pascal instructions are executed in the order in which they occur in the source program. No further technique is therefore required to implement sequential operations, for example:

```
read (a, b, c);
s := (a + b + c)/2;
temp := s*(s-a)*(s-b)*(s-c);
area := sqrt (temp);
writeln (area);
```

Repetition

The pseudocode construct used for repetition is:

```
Perform until condition
    statement(s)
```

This construct implies a leading decision-making strategy, that is, the governing condition is evaluated before the execution of each iteration of the loop, and if the condition is true when the statement is encountered, the loop will not be performed at all.

Pascal provides for such a repetition construct in the **while** ... **do** ... command, for example:

```
sum := 0;
read (number);
while number > 0 do
    begin
    sum := sum + number;
    read (number)
    end;
```

This code will read and accumulate the sum of a series of numbers until one (which may be the first) is read which is not greater than zero.

In addition to the **while** command, Pascal also provides a **repeat** ... **until** ... construct which implements a trailing decision-making strategy. A loop controlled by such a means would always be executed at least once. In this case, the following coding would not necessarily produce the same sum as that above:

```
sum := 0;
read (number);
repeat
    sum := sum + number;
    read (number)
until number <= 0;
```

In the latter case an initial *number* which was not greater than zero would be added to *sum* before the termination of the loop. Unless a particular reason exists for the use of a **repeat** in preference to a **while**, the **while** should be regarded as the normal means of implementing iteration.

A variation on the **while** is the **for** command which also implements a leading decision, for example:

```
sum := 0;
for n := 1 to limit do
    sum := sum + n;
```

If the initial value of *limit* was less than 1 the *for* loop would not have been executed.

Selection

Simple selection in Pascal is implemented by the **if** ... **then** ... **(else)** command, for example:

```
if number < target then
    begin
    number := number + 1;
    total := total + number
    end;
if counter > 0 then
    average := total/counter
else
    writeln ('invalid data');
```

Multiple selection is implemented by the **case** statement, for example:

```
range := grosspay div 10000;
if range > 6 then
    range := 6;
case range of
0, 1, 2 : percentage := 10;
3, 4, : percentage := 25;
5, 6 : percentage := 35
end;
tax := grosspay*percentage/100;
```

Program Modules

The concept of a program module as executed by the **do** construct in pseudocode may be implemented in Pascal by a **begin** ... **end** block of in-line commands, by the execution of a free-standing subroutine by the use of a named procedure or by the declaration of a function with the attendant transfer of parameters at the time of its invocation.

Unless a task can be expressed in no more than six to eight lines of code, it should be constructed as a function or procedure rather than a **begin** block; **begin** blocks which comprise lengthy amounts of code obscure the overall structure and readability of programs.

C2 PROGRAM STRUCTURE

Programs should be written in modules, each of which should be a declared procedure. Overall control of these procedures should be the task of a mainline or driving module which, by the nature of a Pascal program, will be the last procedure to be declared.

Care should be taken to avoid placing too much coding in any one procedure and to separate 'managing' from 'working' procedures.

This procedure is illustrated below by skeletal coding to implement the payroll algorithm in Chapter 2:

```
begin
    paycalculation;
    updatehistoryrecord;
    printpaydetails;
    while (notendofdata) or (not truedetails) do
      obtaincorrectdetails;
end;
```

```
    procedure printfactorytotals
            .
            .
            .
    end;
begin
    (*mainline program procedure *)
    while (not endofdata) or (not truedetails) do
       obtaincorrectdetails;
    while not endofdata do
       processpayroll;
    printfactorytotals;
end
    (* end of program *)
```

C3 MODULE DESIGN

It should be a major consideration in program design that each functional task, consisting of one or more *procedures*, should operate as independently as possible of other tasks in the program.

Pascal provides the facility for both local and global data and strict rules encourage their correct usage.

Data may be exchanged between a procedure and its controlling module (or block) by means of global variables or parameters to the procedure.

Parameters in the procedure declaration (or heading) may be any of four kinds, namely:

value parameter
variable parameter
procedure parameter
function parameter

Here we are concerned only with value parameters and variable parameters.

```
program payroll;
var   .
      .
      .
      .
    procedure processpayroll;
```

```
var  .
     .
     .
   procedure paycalculation;
   var  .
        .
        .
      procedure calculatetax;
            .
            .
            .
      end;
      procedure analyzecoinage;
            .
            .
            .
      end;
   begin
      if hoursworked not > normalperiod
         payable hours : = hoursworked
      else
         payable hours : = normalperiod
                           + overtime factor *
                           (hoursworked - normalperiod);
      calculatetax;
      netpay : = grosspay - taxamount;
      analyzecoinage;
      factory total gross : = factory total gross + grosspay;
      factory total tax   : = factory total tax + taxamount;
      factory total net   : = factory total net + netpay;
   end;
   procedureupdate history record;
        .
        .
        .
   end;
   procedure printpaydetails;
        .
        .
        .
   end;
   procedure obtaincorrectdetails;
        .
        .
        .
   end;
```

A *value* parameter (sometimes known as an input parameter) is one

which is local to the procedure and is used to pass a value to the procedure. Although the value of a parameter may change during the execution of the procedure (although this is to be discouraged), it will return to its original value following the execution of the procedure.

A *variable* parameter (sometimes known as a throughput or output parameter) is one primarily used to pass a value back to the controlling module. Its value may be altered during the execution of the procedure and the parameter will retain the newly calculated value after the procedure has been executed.

C4 PROGRAMMING STANDARDS

The following are suggested as minimum coding standards for Pascal programs:

- Comments should appear in each procedure where an explanation of the function would assist the reader.
- Indentation of embedded procedures, if/else, case and while/do constructs should follow the example of the pseudocode.
- Meaningful names should be used for all variables.
- Constants used within the program should be assigned to variables with meaningful names and those variables used within the procedural statements rather than the constants themselves.
- Boolean values of true and false are provided in the syntax of the language and should be used in operations involving status variables.

Appendix D PROGRAM OPERATION AT MACHINE LEVEL

D1 INTRODUCTION

When a program is running it is resident in memory, wholly or in part, and is being executed by the Control Unit of the central processor. Although it is no longer necessary for programmers to write programs in machine code, an understanding of the principles involved in program operation at machine level assists programmers in understanding why certain features of some languages should be used or avoided in the interests of efficiency of execution. Such an understanding also helps to remove the mystique surrounding programs and exposes them as collections of numbers which have special significance when interpreted by the Control Unit of the computer.

In this section we will review briefly the nature of an internally stored program and clarify the principles involved in its translation from a source language and subsequent execution.

D2 INTERNAL MEMORY

The internal memory of any computer functions as though it were a very large collection of pigeon holes or locations which are capable of holding nothing other than a binary number. A binary system, based on 2, is chosen in preference to the more familiar decimal system, based on 10, because of the engineering simplicity which the binary system permits. The size of the memory locations in any particular computer governs the ease with which data may be stored and manipulated and with which instructions may refer to, or address, a range of such locations in the course of executing a program.

Binary Storage

For a given number, N, of binary digits (bits) the range of combinations of the 1s and 0s is 2^N. These combinations may represent the magnitude of a numeric quantity or codes by which numeric or non-numeric data is represented in the computer system.

Hence a group of 8 bits may represent the combinations ranging from 00000000 to 11111111. In a binary system, the values of the digits ascend in powers of 2, hence the respective decimal equivalents of the above binary numbers can be calculated as follows:

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	
(128)	(64)	(32)	(16)	(8)	(4)	(2)	(1)	
0	0	0	0	0	0	0	0	= 0
1	1	1	1	1	1	1	1	=255

By conventions adopted for the computer, these bits could represent the decimal values 0 to 255, the decimal values -128 to $+127$ or a series of 256 separate characters representing numeric, alphabetic (both upper and lower case) and many special characters (e.g. $ * / = < > ; : etc.). The value thus represented could be interpreted as the whole or part of an item of data or a program instruction.

D3 STORAGE OF DATA

Depending on the size of the locations (i.e. their number of bits) in the memory of a computer, data items may be held within a single location or within a group of locations operated on as a unit or individually, depending upon the nature of the data. The types of data items manipulated by programs were outlined in Chapter 7.

Integers and Fixed Point Numbers

These numbers will be held in pure binary format (i.e. as a true binary number rather than in a binary coded form) usually in a unit of at least 16 bits. This will enable a value to be held within the range of -32768 to $+32767$. Larger units of bits may be used with consequently larger values being able to be held. A computer with a 32-bit location would be able to hold integer values within the range $\pm 2^{31}$ which is approximately ± 2 thousand million.

Negative values are held in complementary format, usually as 2s complements. The 2s complement of a positive number is obtained by changing the 1s to 0s and the 0s to 1s and adding 1 to the result. The following is an example of the range of numbers able to be represented by a group of 8 bits using 2s complements for negative values:

Binary number	*Decimal equivalent*
01111111	127
01111110	126
01111101	125
⋮	⋮

.	.
00000011	3
00000010	2
00000001	1
00000000	0
11111111	−1
11111110	−2
11111101	−3
.	.
.	.
.	.
10000011	−125
10000010	−126
10000001	−127
10000000	−128

Floating Point Numbers

Floating point numbers will normally be allocated a storage unit of at least 24 to 32 bits, of which typically 8 bits would hold the exponent and the remainder would hold the mantissa. Both the exponent and the mantissa are capable of being either positive or negative. This format allows for the storage of values of larger magnitude than those held in integer format but with a loss of accuracy in numbers which do not convert to an exact binary fraction.

Character Data

Character, or string, data may consist of any alphabetic letters, numeric digits or special symbols each of which will normally be allocated one 8-bit unit of storage usually referred to as a byte. The characters will be represented in a coding system such as ASCII (American Standard Code for Information Interchange, a 7-bit code) or EBCDIC (Extended Binary Coded Decimal Interchange Code, an 8-bit code). Numeric data held in this format will normally require conversion to binary integer format for the purpose of any mathematical operations. This will make the program larger because of the additional machine code operations needed to perform the conversion and slower because of the time taken for those instructions to be executed each time the data item is referenced.

Packed Decimal

Packed decimal numbers, also referred to as BCD (Binary Coded Decimal), are a variation on character storage. Each decimal digit is represented by a 4-bit binary code, thus allowing two decimal digits per byte of memory. Such a representation is economic in its use of storage but, unless the computer is able to perform BCD arithmetic, will still need conversion to pure binary for computational purposes.

Boolean Data

Boolean variables are typically held in integer format.

D4 STORAGE OF INSTRUCTIONS

Depending on the size of a computer's memory location, instructions are held one per location, several per location or one may occupy several locations. There may be several formats of instructions in the repertoire of any one computer. The format of an instruction will typically contain:

an Operation Code, and
one or more Operands.

Operation Codes

The operation code indicates the function to be carried out by the Control Unit in executing the instruction. Typical instructions at machine code level which are common to most computers are:

Load and Store operations to transfer the contents of locations from one to another or to or from working registers;
Arithmetic operations to add, subtract, multiply and divide;
Compare operations to test the relative magnitudes of the contents of locations or registers;
Branch operations to transfer program control unconditionally or conditionally on the result of a test on the contents of a location or a register, to transfer control to a subroutine and arrange for the storage of a return address to the calling routine;
Shift operations to move the pattern of bits in a location or a register to the right or left by a specified number of positions;

Logical operations to perform Boolean AND and OR functions between the contents of locations and/or registers; and
Input/Output operations to transfer data between the internal memory and peripheral devices.

In addition to indicating the function to be performed, the operation code usually provides the addressing mode of the instruction. The addressing mode indicates the way in which the instruction interprets the operand. Typical addressing modes are:

Memory Address mode: the operand is interpreted as a memory location the contents of which are to be operated on by the instruction;
Constant mode: the operand is interpreted as a literal or constant value to be manipulated by the instruction;
Indirect mode: the operand is interpreted as a location in which the instruction will find the address of the actual operand to be manipulated.

Operands

An operand in any instruction is a constant, a memory location or a register which is the object of the instruction's attentions. There may be none, one or several operands in an instruction depending on the format employed by a particular computer. This in turn is typically governed by the size of the computer's memory location. Larger locations contain more bits and may therefore be used to represent more than one operand in any instruction.

D5 EXECUTION OF INSTRUCTIONS

Instructions are executed sequentially by the Control Unit with the aid of the Program Counter. The Program Counter is primed with the memory address of the first instruction when the program is loaded into memory and is thereafter incremented automatically as each instruction is decoded and executed by the Control Unit. The operation of branch instructions is to replace the contents of the Program Counter with the address of the instruction to be branched to.

Execution of an instruction involves the fetching of the instruction from the location or locations in which it is stored, decomposing it into its

component operation code and one or more operands, and enabling the mechanism by which the function is effected. This latter stage will involve opening the various electronic gates which enable the circuitry of the processor to execute instructions. It may also involve the execution of microcode, a series of very primitive machine operations which collectively perform one of the instructions in the computer's machine code repertoire. Such microcode is often held in read-only memory (ROM). Some computers are able to have their ROM changed by replacing one of their circuit boards and are thus able to emulate the mode of operation of an entirely different model of computer.

D6 THE PROCESS OF COMPILING A PROGRAM

Programs written in high-level languages such as COBOL, BASIC and Pascal are not in a form executable by the computer's control unit. To enable them to be executed they must be translated into the computer's machine code. This translation may be carried out in a single process referred to as compiling, as is typical with COBOL, or in two processes referred to as interpreting, as is usually the case with BASIC.

Compiling

Compiling is the process of converting a source program written in a high-level language to an object program in machine code. The tasks of the compiler program are to:

(a) break up the source text into the statements comprising the syntax of the language;

(b) check each statement for correctness of format and report any error found for subsequent correction by the programmer;

(c) print (if possible) a listing of the source program together with any syntax errors discovered; and

(d) if the program has no syntax errors, generate an object program consisting of

  (i) memory locations allocated to the symbolic names used by the programmer for variables and constants;

  (ii) machine code instructions to accomplish the functions of the procedural statements in the source program, this process

requiring the generation of many machine code instructions for each source language instruction;

(iii) library subroutines included in the object program to carry out complex functions such as input/output to and from peripherals, mathematical operations such as logarithms, and square roots and trigonometric functions such as sines, cosines and tangents.

The program is thereafter stored in its machine code version and only needs to be recompiled when amendments are made to the source program.

Interpreting

Interpreting is the process of translating a source program into a format which is an intermediate stage between the syntax of the source language and the machine code of the computer. This will involve tasks (a), (b) and (c) of the compilation process but will stop short of the detailed machine code generation in task (d). This reduces the translation time, provides the programmer with a report of any syntax errors with less of a delay than is inherent in full compilation and results in a smaller object program because of the elimination of the necessity to generate voluminous machine code.

Interpreting, however, results in programs which will be slower to execute than those which have been fully compiled. This is brought about by the necessity to generate machine code instructions (task (d) of the compiler) to accomplish the operation of the source instructions as the program is executing. Hence, a source instruction in a loop which is executed 100 times may be converted to machine code 100 times each time that loop is encountered in the running of the program.

As a general principle, interpreted programs will occupy less memory than compiled programs but will execute at a slower rate.

D7 CONCLUSION

This is a brief summary of the background of knowledge needed by a programmer when faced with the choice of using:

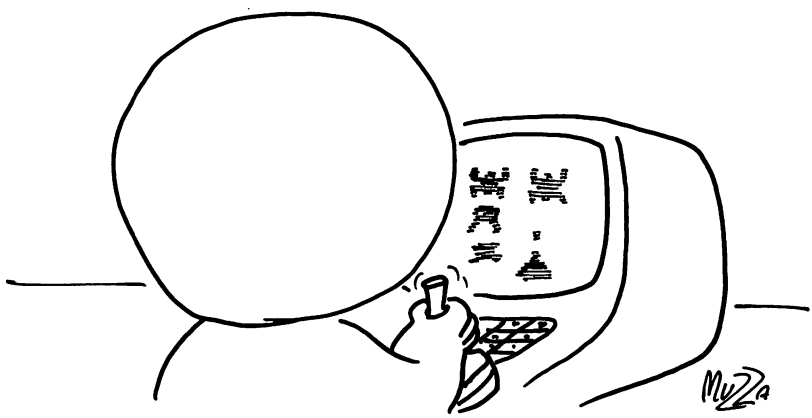
(a) integers as opposed to floating point variables in BASIC or Pascal

or the choice between DISPLAY and COMPUTATIONAL variables in COBOL, or

(b) a fully compiled as opposed to an interpreted version of a programming language, or

(c) a microcomputer with replaceable ROM enabling it to run software originally written for a different computer.

High-level languages have removed much of the tedium otherwise associated with writing programs at machine level, but they have also removed the intimacy between programmer and computer which promotes a fuller understanding of what is actually happening when a program executes.



INDEX

INDEX