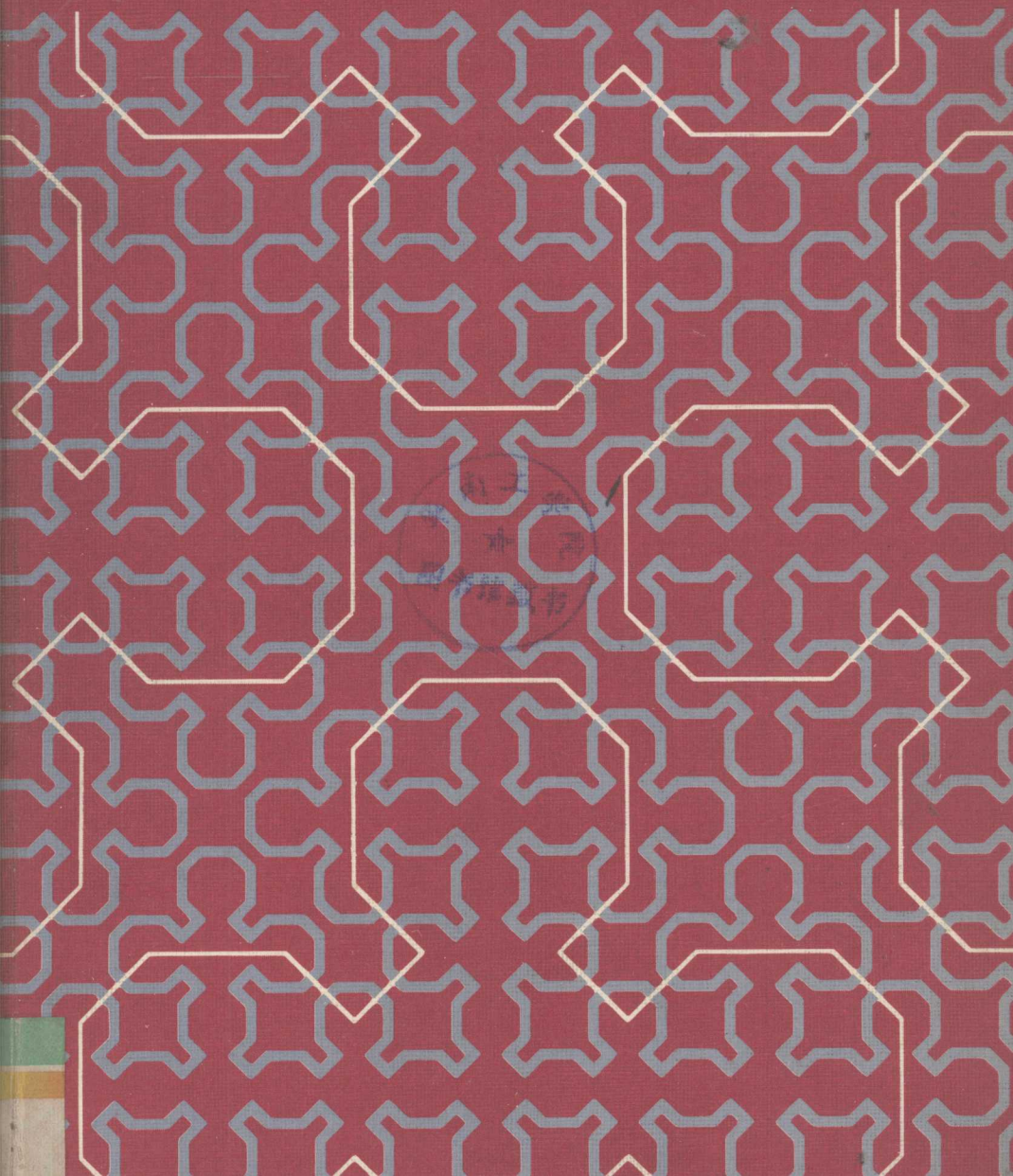
Cambridge Computer Science Texts · 16

Writing Pascal programs

J. S. ROHL



Cambridge Computer Science Texts. 16

Writing Pascal programs

J.S.ROHL

Department of Computer Science, University of Western Australia

CAMBRIDGE UNIVERSITY PRESS

Cambridge

London New York New Rochelle

Melbourne Sydney

Published by the Press Syndicate of the University of Cambridge
The Pitt Building, Trumpington Street, Cambridge CB2 1RP
37 East 57th Street, New York, NY 10022, USA
296 Beaconsfield Parade, Middle Park, Melbourne 3206, Australia

First published 1983

Printed in Great Britain at the University Press, Cambridge

Library of Congress catalogue card number: 82-14591

British Library cataloguing in publication data

Rohl, J. S.
Writing Pascal programs. – (Cambridge computer science tests; 16)

1. PASCAL (Computer program language)
I. Title
001.64'24 QA76.73.P2

ISBN 0 521 25077 3 hard covers
ISBN 0 521 27196 7 paperback

DJ

WRITING PASCAL PROGRAMS

Also in this series

1 An Introduction to Logical Design of Digital Circuits
C. M. Reeves 1972

2 Information Representation and Manipulation in a Computer
E. S. Page and L. B. Wilson, Second Edition 1978

3 Computer Simulation of Continuous Systems
R. J. Ord-Smith and J. Stephenson 1975

4 Macro Processors
A. J. Cole, Second Edition 1981

5 An Introduction to the Use of Computers
Murray Laver 1976

6 Computing Systems Hardware
M. Wells 1976

7 An Introduction to the Study of Programming Languages
D. W. Barron 1977

8 ALGOL 68 - A First and Second Course
A. D. McGettrick 1978

9 An Introduction to Computational Combinatorics
E. S. Page and L. B. Wilson 1979

10 Computers and Social Change
Murray Laver 1980

11 The Definition of Programming Languages
A. D. McGettrick 1980

12 Programming via Pascal
J. S. Rohl and H. J. Barrett 1980

13 Program Verification using Ada
A. D. McGettrick 1982

14 Simulation Techniques for Discrete Event Systems
I. Mitrani 1982

15 Information Representation and Manipulation using Pascal
E. S. Page and L. B. Wilson 1982

CONTENTS

PREFACE

Teachers of introductory programming courses face an important problem: the ideas of programming and the constructs of programming languages must be introduced largely in a bottom-up fashion; while the design of a program generally has to be top-down. The authors of primers must deal with this dilemma, and, as well, consider issues of testing, efficiency and so on. In this multi-dimensional space, most authors choose to cover the language completely (as they are bound to do) and devote less than adequate space to other aspects especially program design. Thus the programs given tend to be both idealised and short; they are bereft of comments, and produce minimal output.

And yet for real programs these are the essence. A program that slights its user by producing unacceptable output will soon be replaced by another; a program that has no design documentation or comments will fare badly as others try to modify or extend it; a program that has not been adequately tested will undoubtedly fail at some point when it is sorely needed.

This book seeks to remedy this situation at least as far as Pascal is concerned. It consists of 15 chapters in each of which a complete program is designed. No attempt is made to teach Pascal - indeed it is assumed that the reader is following a traditional Pascal course in parallel. Thus the book should be a good companion to all current Pascal primers.

We assume that the student will have had perhaps six lectures before starting on this book. The early programs assume an understanding of simple forms of the for-statement and the while-statement, together with rudimentary input and output facilities. Thus these early chapters should help the student who understands all the lectures but does not know where to start when asked to write a program.

As the book advances, the programs use progressively more of Pascal's facilities. While there is no common agreement amongst teachers as to the order in which facilities should be introduced, this should not matter greatly. Reading of this book can be suspended for a week or two until the key facility is covered.

This latter part of the book should be of help to the student having difficulty with the grand structuring devices of Pascal such as the procedure and the record.

There is only one Pascal facility that does not feature in this book, dynamic storage. This decision was taken largely because it was felt that dynamic storage is best introduced in the context of data structures - a topic that needs a book of its own. All the problems covered here have simple data structures and simple algorithms. I think it fair to say that all readers of this book could do all the problems that we construct programs for, quite simply, if not completely accurately, by hand.

I should like to record my thanks to Mrs Joyce Fisher and Mrs Fiona Walker who typed the manuscript, to Mr Mike Palm who provided me with supporting system software and to Mrs Janet Brockman who ran the programs and checked by proof-reading.

Perth, March 1982 J.S.Rohl

1 THE DETERMINATION OF EASTER

Computers were created originally to perform arithmetic calculations and it is appropriate that our first program is in this category. The calculations were generally long, complex and tedious, being concerned with the determination of artillery tables, the cracking of codes and so on. Our problem is a much simpler one, and the calculations involved are neither long nor complex – just tedious.

1.1 The problem description

The problem is to write a program that will print out, for certain years read as data, the date of Easter Day. For reasons that will become obvious later we will restrict the years to be within the range 1900-2099 inclusive.

Thus the results of the program might be as shown in Fig. 1.1.

Fig. 1.1. The output from the Easter Day program.

```
                EASTER
YEAR              DAY

1900          15 APRIL
1901           7 APRIL
1902          30 MARCH
1903          12 APRIL
1904           3 APRIL
1905          23 APRIL
1906          15 APRIL
1907          31 MARCH
1908          19 APRIL
1909          11 APRIL
1980           6 APRIL
1981          19 APRIL
1982          11 APRIL
1983           3 APRIL
1984          22 APRIL
1985           7 APRIL
1986          30 MARCH
1987          19 APRIL
1988           3 APRIL
1989          26 MARCH

SOME EASTER DATES
=================
```

1.2 The broad outline

It is reasonable to assume that the data is presented as a sequence of years, preceded by the number of years, so that the data to produce Fig. 1.1 would be:

```
20
1900 1901 1902 1903 1904 1905 1906 1907 1908 1909
1980 1981 1982 1983 1984 1985 1986 1987 1988 1989
```

This leads immediately to the following outline program:

```
begin
write out the heading;
read(noyears);
for line := 1 to noyears do
   begin
   read and write year;
   if year not in range 1900-2099 then
      write out a suitable message
   else
      begin
      calculate the date of Easter;
      write out the date
      end
   end;
write out the title
end.
```

in which the English phrases represent the parts of the program we have yet to design.

Five of these are concerned with output - that is all those except *calculate the date of Easter.* Once we have decided on the form of output these are quite straightforward. In Fig. 1.1 we have a margin of 4 spaces and a gap of 6 spaces between the columns. The year column occupies 4 spaces and the date column 8. It is a fairly simple matter to ensure that the headings, the body of the columns and the table title are aligned.

However, we must always bear in mind that we may later wish to alter the layout by changing the width of the margin, the gap or the columns, or by adding extra columns. We should write the output-statements to facilitate these changes. Except for the title, we will use write-statements in which there is an explicit component written for each column and for the margin and the gap. For example the second line of the heading is:

```
writeln(' ':4, 'YEAR':4, ' ':6, 'DAY':8)
```

In this statement there are two further things worthy of comment. Firstly note the use of ' ':6 to get 6 spaces. Technically the computer prints out the character ' ' with a field width of 6, the first 5 being filled, as normal, by spaces. This avoids the necessity of typing a string of 6 (and generally many more) spaces. Secondly note the similar use of a field width with strings which also avoids having to type (and perhaps change later) a precise number of spaces. Of course as Fig. 1.1 demonstrates, this causes the strings to be right-justified. If this is not what is required then the spacing must be explicitly specified in the strings.

1.3 The Easter algorithm

Note that we have produced an outline design without yet considering how to calculate the date of Easter. We simply postponed the problem for later consideration, a technique we will continue to use. Of course ultimately we have to solve it - and for the Easter program that time has come.

Few readers know how Easter is determined, and for those who do know the rule:

Easter is the first Sunday after the first full moon on or after 21st March

even fewer will have sufficient knowledge of astronomy to derive an algorithm.

Of course such algorithms already exist. Some are completely general, others are restricted to certain periods of time. The one we will use, due to T. H. O'Beirne (Table 1), is restricted to this century and the next.

Since the quotient and remainder can be implemented by means of Pascal's **div** and **mod** operators, the algorithm can be quite easily implemented.

Table 1. *To calculate Easter for any year y between 1900 and 2099 inclusive*

Subtract 1900 from y to find n, such that $0 \leqslant n \leqslant 199$; then use n as indicated below.

Step	Dividend	Divisor	Quotient	Remainder
(1)	n	19	–	a
(2)	$7a + 1$	19	b	–
(3)	$(11a + 4) - b$	29	–	m
(4)	n	4	q	–
(5)	$n + q + (31 - m)$	7	–	w

Between the years 1900 and 2099 inclusive, Easter Sunday is April $(25 - m - w)$: with an obvious interpretation of dates from April 0 to April (-9), inclusive.

1.4 The program

It is a simple matter now to fill out the broad outline to produce the program of Fig. 1.2.

Fig. 1.2. The Easter program.

```
program Easter(input,output);
{ This program prints out a table of dates for Easter Day.
   The algorithm, due to O'Beirne, is described in Chapter 1
   of "Writing Pascal Programs". This program is filed as
   WPP1.PAS on [55,0] }
var y,n,a,b,m,q,w,Easter,line,noyears:integer;
begin
page(output); writeln; writeln;
writeln(' ':4,' ':4,' ':6,'EASTER':8);
writeln(' ':4,'YEAR':4,' ':6,'DAY':8);
writeln;
read(noyears);
for line := 1 to noyears do
  begin
  read(y);
  write(' ':4,y:4);
  if (y<1900) or (y>2099) then
    writeln(' ':2,'is outside the range 1900-2099.')
  else { we have a valid year }
    begin
    n := y - 1900;
    a := n mod 19;
    b := (7*a + 1) div 19;
    m := (11*a + 4 - b) mod 29;
    q := n div 4;
    w := (n + q + 31 - m) mod 7;
    Easter := 25 - m - w;
    if Easter >= 1 then
      writeln(' ':6,Easter:2,'April':6)
    else
      writeln(' ':6,Easter+31:2,'March':6)
    end { of sequence dealing with a valid year }
  end { of loop on "line" };
  writeln;
  writeln(' ':4,'SOME EASTER DATES');
  writeln(' ':4,'================')
end.
```

1.5 Testing the program

How do we know that the program of Fig. 1.2 works? The answer is that we must test it using data for which we know the correct result. What data then is required for this program? Well, it covers only 200 years so that we could exhaustively test the program by running it with all 200 years as data. But how would we know whether the results are correct, given that the average diary covers only the current year together with the previous and perhaps the next years? As it happens, the Oxford Companion to English Literature gives the datea for years back to the Norman Conquest and the Book of Common Prayer gives

tables that enable the dates to be simply determined. Thus by a certain amount of effort we could find the correct dates for the two centuries involved and check our program completely.

However, this would be something of an overkill. With the exception of the final printing, the same statements are obeyed for every year. Thus we simply need to ensure that all statements are sufficiently exercised. The data of Fig. 1.1, together with that described in the next paragraph, is adequate (though the proof is a little tedious).

Of course we must also check that it recognises valid data, and handles invalid data correctly. Thus we must check data around the limits of validity. As it is all too easy to punch <= instead of < we would want to run with the years 1899, 1900, 1901 at the lower end and 2098, 2099 and 2100 at the upper end.

1.6 Some observations

(i) Even though we have been careful to use write-statements that reflect the structure of the output, it is still an error-prone business to modify the program to alter the structure of the table. This is because, for example, the size of the margin, 4, appears in many places; and all of these must be found and altered if we wish to reduce it to, say, 3. The Pascal constant facility is of great use here. If we define constants as below:

```
const margin = 4;
      gap = 6;
      col1 = 4;
      col2 = 8;
```

and rewrite the output-statements to use the constants so that, for example:

```
writeln(' ':4, 'YEAR':4, ' ':6, 'DAY':8)
```

becomes

```
writeln(' ':margin, 'YEAR':col1, ' ':gap, 'DAY':col2)
```

then a change to the margin size (or the gap or the columns) requires a change to the constant definition only.

The robustness that the use of constants gives is so important that we shall use the facility extensively in subsequent chapters. We can go even further and define

```
const margin = '    ';
```

and then use *margin* instead of ' ':*margin* in the *writeln*-statement. We will often find that we need to use both ideas in a program.

(ii) Most of a programmer's time is spent in reading a program either because

it requires modification and extension (see Ex. 1.1 for example) or because the program as initially written does not work and requires correction. As an aid to reading programs we will always include comments. In *Easter* there are two sorts of comment, which we use in every program. Firstly, we include immediately after the program heading a comment that briefly describes the program, indicates where a fuller description may be found, and specifies where the program itself is stored. Secondly, we add a comment to each **end** indicating the construct that it terminates. For loops, it will simply refer to the control variable:

end {*of loop on "i"*}

For the alternatives of a condition a more expansive comment is given:

end {*of sequence dealing with a valid year*}

We will introduce further conventions as we proceed.

(iii) The computer on which these programs were run has only upper case characters, and hence all output will be in that form. The programs were prepared, however, on a machine with both upper and lower case and we have used these facilities to advantage, as can be seen from Fig. 1.2. All complete programs in this book are taken directly from lineprinter copy obtained after processing the text to put delimiter words in bold face, and to replace some rather ugly punching conventions.

Some related exercises

1.1 O'Beirne has also given a general algorithm for determining Easter:

Table 2. *To calculate Easter for any year of the Gregorian calendar*

Step	Dividend	Divisor	Quotient	Remainder
(1)	x	19	–	a
(2)	x	100	b	c
(3)	b	4	d	e
(4)	$8b + 13$	25	g	–
(5)	$11(b - d - g) - 4$	30	θ	–
(6)	$7a + \theta + 6$	11	ϕ	–
(7)	$19a + (b - d - g) + 15 - \phi$	29	–	ψ
(8)	c	4	i	k
(9)	$(32 + 2e) + 2i - k - \psi$	7	–	λ
(10)	$90 + (\psi + \lambda)$	25	n	–
(11)	$19 + (\psi + \lambda) + n$	32	–	p

In the year x AD of the Gregorian calendar, Easter Sunday is the pth day of the nth month.

Extend the Easter program of Fig. 1.2 using this algorithm.

1.2 The Easter programs reflect not only the motion of the moon but also the nature of the current (Gregorian) calendar, in which a year has 365 days unless it is a leap year in which case it has 366. A year n is a leap year if it is divisible by 4, but not by 100, unless it is also divisible by 400. Given that 1 Jan 1900 was a Monday, write a program that will tabulate the day on which New Year's Day falls for years read in as data.

1.3 Extend the New Year program of Ex. 1.2 so that it tabulates the day of other important festivals whose dates are fixed such as Christmas Day (25 December), Bastille Day (14 July), Anzac Day (25 April) and so on.

1.4 Write a program to tabulate the dates of festivals whose occurrences are a little more variable such as Melbourne Cup Day (the first Tuesday in November), Thanksgiving Day (the fourth Thursday in November) and so on.

2 PRODUCING A BLANK CLASS TIMETABLE

In many programs, the creation of the output is a significant part of the problem. Furthermore it is a very important part of any program, since it is one of the primary interfaces between the program and its users. In this chapter we will consider a problem that consists almost entirely of creating the output.

This problem illustrates two important points:

(i) The output of a program determines, to some extent or other, the structure of that program. Here, of course, it determines it completely.

(ii) Lineprinters were designed for the production of text. When constructing more diagrammatical output we have to operate within the consequent limitations such as the fixed-width characters and the discrete, left-to-right, top-to-bottom printing process.

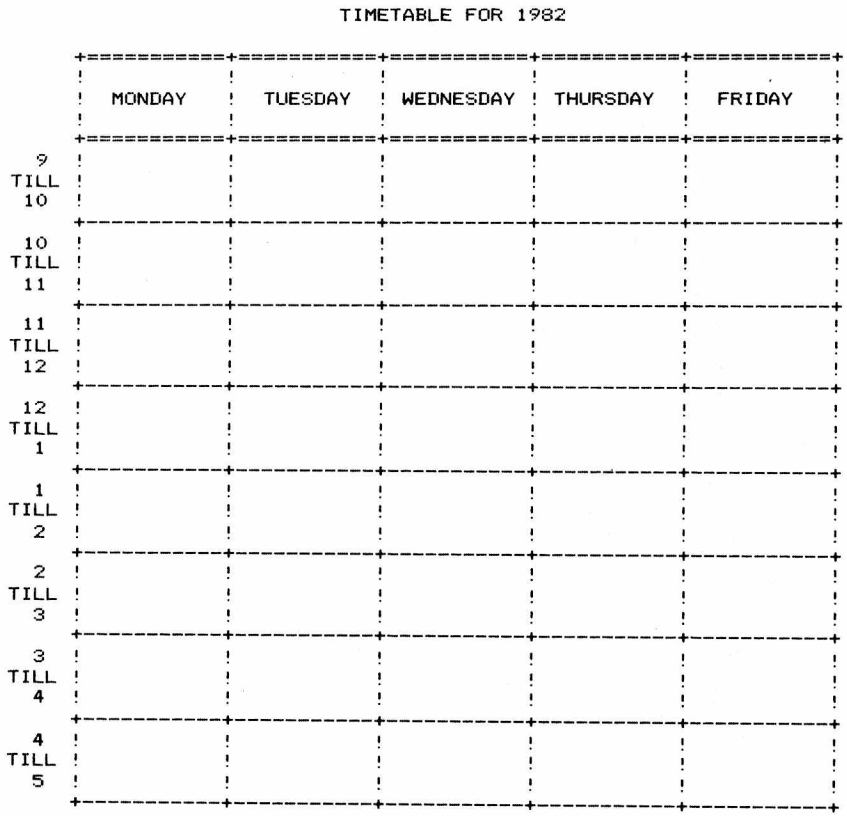
2.1 The problem description

The problem is simply to write a program to print out a blank class timetable for university or college use such as the one shown in Fig. 2.1. The timetable is to cover only the five weekdays, each of which is divided into a number of one-hour periods. The number of periods is variable (but not greater than 12) and the start time together with the number of periods is provided as data, the start time being in the usual 12-hour clock system. Thus, bearing in mind that the year is also read in, the data to produce Fig. 2.1 is:

```
1982    9    8
```

The timetable is to be no larger than A4-sized paper so that it can be stuck inside a notebook or file. In the first instance it should be no wider than A4 and we do not expect significant changes. However should we later decide that we want a timetable that is wider and shallower (such as would fit sideways inside an A4 notebook) then this should be possible without major reorganisation.

Fig. 2.1. A blank timetable for 1982.



Let us suppose that the consequent decisions about the shape of the timetable are as shown in Fig. 2.1. That is the boxes are 11 characters wide and 3 lines deep and so on. We assume too the choice of characters, =, !, − and + for constructing the boxes.

2.2 The development of the program

It is clear that a timetable is made up of vertical and horizontal lines together with text describing the days and the periods covered. The lines divide the timetable both horizontally and vertically into boxes. This structure should be reflected in the structure of our program. We must, however, concentrate on the horizontal lines since the lineprinter operates a line at a time.† Thus we can

† If we were using a graph plotter, of course, we would not be constrained in this way.